Analyzing the Worst-Case Behavior of Multi-Level Caches in Concurrent Real-Time Systems

Vom Promotionsausschuss der
Technischen Universität Hamburg

zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften (Dr. rer. nat.)

genehmigte kumulative Dissertation

von
Thilo Leon Fischer

aus
Hamburg

2025

DOI: 10.15480/882.16335
https://orcid.org/0000-0002-6309-8979

Gutachter: Prof. Dr. Heiko Falk
Prof. Dr. Sibylle Schupp

Tag der mündlichen Prüfung: 11. Dezember 2025

Summary

Real-time systems are a class of embedded computing systems that require timely output. Consequently, if a deadline is missed the system fails to operate correctly. In applications where failure can lead to catastrophic consequences, such as harm to humans, the system is classified as a *hard* real-time system. Due to these consequences, it is essential that the timing behavior of a hard real-time system is verified prior to its deployment. This verification process is performed using worst-case analysis tools, which inspect the application's code and its interaction with the hardware platform to determine the worst-case timing behavior.

Advances in computer engineering have increased the complexity of hardware architectures used in real-time systems. This thesis focuses on enhancing the capabilities of static worst-case analysis tools to better support these complex architectures. In particular, two trends which pose significant challenges for static worst-case analysis are tackled: the use of multi-level caches and multi-core architectures.

The first contribution of this thesis is a novel analysis approach for shared caches in multi-core real-time systems. The key insight leveraged in the analysis is that inter-core interference does not manifest instantly after new data is stored in the shared cache. Rather, it takes time for the interfering cores to issue conflicting accesses that create contention in the shared cache and ultimately evict the data. This interference behavior is quantified using event-arrival curves. The analysis classifies individual accesses to the shared caches into definite cache hits and potential cache misses. Compared to previous analyses, the presented approach improves the classification precision and worst-case execution time estimation.

The second contribution of this thesis is an analysis of *cache-related preemption delay* (CRPD) in a two-level non-inclusive cache hierarchy. CRPD occurs if the currently executing task is interrupted in favor of another task. Naturally, the preempting task can evict data of the preempted task from the caches. When resuming the preempted task, its execution is delayed as it has to reload data from memory into the caches. In multi-level cache hierarchies, two distinct interference effects occur: *direct* and *indirect* interference. The work in this thesis shows that the previous state-of-the-art method to analyze CRPD in non-inclusive two-level caches was flawed in its estimation of these interference effects. Thus, it may produce unsafe results. This thesis provides formal foundations for the analysis of interference effects. Furthermore, a novel and safe analysis for non-inclusive two-level cache hierarchies is presented. The presented analysis improves upon the precision of previous analyses, leading to increased system schedulability.

Finally, the presented CRPD analysis is applied in an optimization to improve system schedulability. The optimization uses cache bypassing to reduce context-switching costs and intra-task interference. Caches are bypassed by allocating parts of the application to uncached memory during compilation. Allocation decisions are made using metaheuristic algorithms. While bypassing caches can increase a program's execution time, the evaluation results show that the reduction of cache interference leads to improved system schedulability.

Acknowledgments

I want to dedicate this work to my late father, who sparked my curiosity in mathematics and engineering.

I extend my deepest gratitude to my mother and sister for their unwavering support throughout these years. Without you, this thesis would not have been possible.

A heartfelt thank you to Arne, a friend since our Bachelor's studies, for the many unforgettable moments we shared in Hamburg and Lübeck.

A special thank you to my friend Johannes for all the thought-provoking discussions on the nature of the universe and the meaning of life.

Finally, I would like to thank my advisor Prof. Dr. Heiko Falk for providing me with the opportunity and the support to create this thesis.

Contents

Introduction

1

Over recent decades, computers have become deeply integrated into our daily lives, appearing in devices like smartphones, smartwatches, and laptops. Furthermore, many devices with which we interact are not directly recognizable as computers but rely on computers under the hood to operate. For instance, when driving a car we are not interacting with a purely mechanical machine but with many different computers that react to our inputs and communicate with each other. These computers are called *embedded systems*, because they are embedded into a surrounding product, which renders them invisible to the user. The advances in performance, cost efficiency and miniaturization have driven the adoption of embedded systems in numerous application areas.

In particular, this thesis focuses on *real-time systems*, which are a special class of embedded computing systems designed to produce their output within time constraints. Hence, the correct operation of a real-time system depends not only on the functional correctness, meaning the computed result, but also on the time required to produce it. A delayed result can be equally critical to an incorrect or completely missing result. The deadlines of a real-time system are dictated by the requirements of the larger product in which the system is embedded. These requirements mean real-time systems are found in applications such as aerospace and automotive systems, as well as robotics and medical devices.

Real-time systems are grouped into two categories: *soft* and *hard* real-time systems. In soft real-time systems, missing a deadline degrades the quality of the service. An application type that can be categorized as a soft real-time system is video streaming. Missing a deadline while decoding video frames leads to a degradation of the video playback quality. In contrast, in hard real-time systems meeting all deadlines is crucial, as otherwise the complete system can fail — with potentially catastrophic consequences, such as endangering and causing harm to humans. An example for a hard real-time system is the airbag control in a car. When a crash occurs, the system needs to detect the collision and calculate which airbags should be deployed. This process has a fixed deadline due to the interaction with the physical environment. If the system reacts too late, the passengers will not be protected by the airbags.

Given these severe consequences of a delayed result in hard real-time systems, it is critical to rule out that a deadline could be violated in the worst-case scenario. The methods needed for this verification process are found in the discipline of worst-case timing analysis. Worst-case timing analysis aims to compute bounds on the timing behavior of a system. It can be performed *dynamically*, by executing the program and measuring its behavior, or *statically*, by applying analytical methods to the application's code.

The requirements placed on embedded processors by system designers and consumer demand have increased steadily. To meet these escalating demands, the structure of the embedded computers has become more and more complex over time: single-core processors

have given way to multi-core architectures to increase the computational bandwidth; and simple memory modules have been augmented by multi-level caches to reduce the average memory access latency. This trend in architecture design complicates the worst-case timing analysis as the interaction of different components becomes increasingly difficult to predict during the system design phase.

In particular, the utilization of caches contributes to the unpredictability of the timing behavior of a system. Caches are used to reduce the average latency of a memory access. However, as detailed above, in real-time systems, the worst-case behavior is critical. In order to guarantee tight bounds on the worst-case behavior of an architecture containing caches, a detailed analysis of the cache behavior is required. Otherwise, if a precise analysis of the timing behavior is impossible, the architecture is unsuitable for use in hard real-time applications.

The analysis of the worst-case cache behavior has been a focus of research for over 25 years. However, the increasing complexity of modern hardware architectures means that the methods used in existing cache analyses are not sufficient anymore. The inadequacy of the existing cache analyses techniques manifests itself in two dimensions: 1. the results obtained by existing analyses lack the necessary precision to be of use in system verification, or 2. the analyses impose restrictions that are not fulfilled due to the complexity of modern architectures.

Consequently, the capabilities of timing analysis methods need to evolve to cover the advances made in the realm of computer engineering. Bridging this gap is essential to allow modern hardware architectures to be used safely in hard real-time applications.

The research question underlying this thesis is whether it is possible to construct precise analyses of cache behavior to allow for the utilization of modern hardware architectures in hard real-time systems. In particular, the contributions portrayed in this thesis aim to close the gap between modern hardware architectures containing multiple, connected caches and the capabilities of static worst-case analysis. A discussion of the contributions contained in this thesis to the state-of-the-art of static worst-case analysis is given in the following section.

1.1 Contributions

This thesis advances the state-of-the-art of timing analysis for systems containing caches organized into multiple cache levels.

The first research problem addressed by the contributions of this thesis is the precision with which shared caches in multi-core systems can be analyzed. Data stored in shared caches is subject to interference from programs running in parallel on other cores. Due to the complexity of cache accesses from multiple cores interacting at the shared cache, previous analysis approaches were imprecise or had prohibitive computational overhead. By incorporating timing information into the analysis of cache accesses, we achieve a precise analysis with manageable analysis time overhead.

The second research problem tackled by the contributions of this thesis is the interaction of tasks in multi-level cache hierarchies under preemptive scheduling. Preemptive schedul-

ing allows for the interruption of a task in favor of a higher priority task. When using preemptive scheduling, the state of the caches can be modified during a preemption: the preempting tasks issues accesses to the cache, evicting data of the preempted task. This interference disturbs the execution of the preempted task when it resumes execution. The effects on the cache require a precise analysis to determine an upper bound on the delay incurred by the preempted task. The preemption effects have been studied extensively for single-level caches in the past. However, the analysis of multiple cache levels is more complex due to the way the cache levels interact with each other. Only few publications have previously addressed this interaction in multi-level cache hierarchies. The contributions in this thesis highlight issues in the state-of-the-art, provide a formal foundation, and present a novel analysis approach, which outperforms previous analyses.

Finally, an application of the CRPD analysis for system optimization is illustrated. The optimization employs cache bypassing to improve the system schedulability. While bypassing caches can increase a program's execution time, it can also decrease context-switching costs and intra-task interference. Two metaheuristic algorithms, *simulated annealing* and the *strength pareto evolutionary algorithm*, are explored for the optimization problem. The presented results demonstrate that the tradeoff between execution time and cache interference can be performed such that system schedulability is improved.

The following list contains a detailed description of the presented contributions:

- A novel perspective on inter-core interference for shared caches based on the concept of event-arrival curves and an ILP model to derive these interference curves from the machine code of an interfering task.

- An algorithm to determine the cumulative interference curve of multiple tasks that are scheduled *non-preemptively* on an interfering core. The algorithm is shown to compute safe results.

- An extension of the shared cache analysis to *preemptively* scheduled task sets. The approach is based on a novel formal language that encapsulates the behavior of fixed-priority preemptive scheduling.

- A data-flow analysis which utilizes timing information and interference curves to classify accesses to a shared cache as definite cache hits and potential cache misses.

- An analysis of the previous state-of-the-art technique for cache-related preemption delay in two-level non-inclusive cache hierarchies, which shows that the technique is unsafe and pessimistic.

- A formal foundation for the analysis of indirect interference effects after a preemption in non-inclusive cache hierarchies.

- A safe analysis of the cache-related preemption delay for non-inclusive cache hierarchies, which consists of multiple data-flow analyses and algorithms that are shown to compute safe results.

- An optimization to improve schedulability for systems containing a two-level instruction cache hierarchy. By allocating parts of the code in uncached memory, cache interference is reduced. Allocation decisions are made using metaheuristic algorithms.

1.2 Structure

This thesis is structured as follows: Chapter 2 highlights foundational concepts of hardware architectures for real-time systems. In particular, the focus lies on multi-core architectures and caches as they are key contributors to the worst-case timing behavior, and are the focus of the timing analyses presented in this thesis. Chapter 3 introduces the methods used in timing analysis of an individual task. The provided tools allow for the derivation of an upper bound on the execution time, under the restriction that no interference from other tasks occurs. Chapter 4 continues by extending the timing analysis to systems containing multiple tasks. It also provides an overview over the *real-time calculus*: an analysis technique viewing a system as a stream of events arriving at, and being processed by, different components. Chapter 5 explores the analysis of caches in real-time systems in greater detail as the behavior of caches is a key component in the latency incurred by accesses to the system memory.

Chapter 6 contains the first of the publications, which constitute the core of this thesis. Shared caches are analyzed in non-preemptive multi-tasking systems by employing techniques from real-time calculus and static code analysis. The second publication by the author, reproduced in Chapter 7, extends the analysis of shared caches from the previous chapter to preemptively scheduled systems. The third publication in Chapter 8 shifts the focus towards the analysis of context-switching costs incurred by preemptive scheduling for systems containing multiple cache levels. Finally, Chapter 9 contains the fourth publication, which highlights an optimization to improve system schedulability by strategically employing cache bypassing.

Chapter 10 concludes this thesis by providing a summary of the contributions, discussing the results and giving an outlook on open questions, which need to be answered in future work.

Hardware Architecture

2

In order to determine the timing behavior of a system and verify that all deadlines are met, it is necessary to consider which components exist in the system and how these components interact with each other. The aim of this chapter is to introduce foundational concepts of hardware architectures commonly utilized in modern real-time systems. In particular, we explore two defining architectural characteristics with significant impact on the worst-case timing behavior. First, Section 2.1 motivates the utilization of multiple processing cores in a single system and introduces the phenomenon of inter-core interference. Second, Section 2.2 introduces the concept of caching and discusses its realization in hardware.

2.1 Multi-Core Systems

In this section we explore the motivation leading to the adoption of multi-core processors and highlight a challenge arising due to the utilization of multi-core processors for real-time systems.

The computational power of a system containing a single core is the result of the computational power of that individual core and of the connected components. The expected performance of a processor can be measured by the number of instructions that are processed in a given time frame. More precisely, the system performance can be estimated by considering the average *cycles-per-instruction* (CPI) value of the architecture as well as the clock frequency of the particular processor. The CPI value of a processor can be determined by measuring or simulating how many clock cycles are needed to process a single instruction on average. Using the CPI value, we can estimate the time a processor needs to complete a program:

$$\text{CPU time of program} = \frac{\text{Instruction Count} \cdot \text{CPI}}{\text{Clock Frequency}}. \tag{2.1}$$

Suppose we want to reduce the average time required to execute a given program. Assuming the instructions contained in the program are fixed, there are only two ways to achieve this goal: either the CPI value of the processor has to be reduced or the clock frequency has to be increased.

Improving the CPI value is possible by exploiting instruction level parallelism, which allows the system to process multiple instructions of the program simultaneously [HP11]. However, this approach requires architectural changes to the processor core.

The second way to improve the average system performance is to increase the clock frequency. However, in the mid 2000's processor development hit the so-called *power-wall* [PH12].

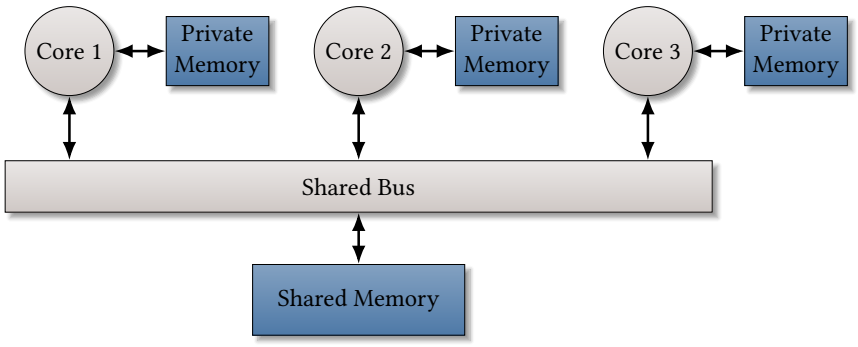


Figure 2.1. – A multi-core architecture with three cores. Each core has its own private memory. The cores are connected via a shared bus to a larger shared memory.

The power consumed by a processor is proportional to the clock frequency and the square of the voltage used to operate it. This means that increasing the clock frequency is quickly limited by the required power and resulting waste heat, which necessitates elaborate cooling of the processor. Historically, this limitation of uni-core processors manifested in the mid 2000's, and caused multi-core architectures to increase in popularity.

The idea behind using a multi-core processor is to utilize multiple slower but more efficient cores instead of a single high performance core. This architectural design choice can increase the throughput of the processor, under the condition that the executed application can be divided into multiple smaller pieces that can be processed in parallel on the different cores.

An example for a multi-core architecture is shown in Figure 2.1. The shown architecture contains three processor cores. Each core can execute a program functionally independent of the computation that is performed on the other cores. In this example, each core possesses a private memory. Additionally, every core is attached to a shared bus, which offers access to the shared memory.

The speedup for the execution of a program on a multi-core processor, compared to a single-core processor, is limited by *Amdahl's law* [HP11, p.46]:

$$\text{Speedup}_{\text{overall}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}. \tag{2.2}$$

The value $\text{Fraction}_{\text{enhanced}}$ is the fraction of the program that can be transformed to execute in parallel. The value $\text{Speedup}_{\text{enhanced}}$ is the speedup factor that a parallelized architecture can provide. Finally, $\text{Speedup}_{\text{overall}}$ is the resulting speedup that can be expected by executing the transformed program on a multi-core processor. For example, a program of which 80% can be parallelized to four cores can expect a speedup factor of 2.5×.

This example shows that adopting a multi-core architecture is a viable avenue to improve the system performance. However, using an architecture with multiple independent processing cores creates new challenges; especially in the domain of real-time systems, where predictability is paramount.

These challenges arise due to *resource contention* between the multiple cores. Typically, resources shared among cores impose exclusive access, i.e., they may only be accessed by a

single core at any point in time. Hence, conflicts occur when multiple cores request access to a shared resource at the same time.

As can be seen in the example of Figure 2.1, all three cores have access to the shared bus. However, as the underlying hardware implementing the bus is shared among all cores, only a single core may be granted access at any point in time. Suppose two cores desire to access a shared resource. If one of the cores has exclusive access to such a shared resource and the other core desires access to that resource, one of the cores has to wait until it is granted access to the resource. This means that while the cores are functionally independent, the timing of the execution of one core depends on the behavior of all other cores in the system. This effect on the timing behavior is known as *inter-core interference*. Inter-core interference needs to be considered in the design and verification of real-time systems. This thesis tackles the inter-core interference occuring in shared caches in Chapters 6 and 7.

Bus Arbitration Schemes

The nature and extent to which inter-core interference manifests itself in the timing behavior depends on the decision-making process used to grant access to shared resources. This process is called *arbitration* [Dal04]. In the following, we introduce selected arbitration approaches for shared buses.

The *fixed-priority* arbitration approach operates by assigning a static priority value to each processor core. Access is granted to the core with the highest priority, which has requested access. The fixed-priority approach is *un-fair*, because a high priority core could continuously use the bus and consequently starve lower priority cores of access to the bus.

An approach to arbitration, which prevents starvation, is called *time-division multiple access* (TDMA). In TDMA, the access to the resource is structured using time slots. Each core is assigned one or multiple slots in which it may access the resource. The time slots are arranged in a periodically repeating schedule. A core may only access the bus if it is assigned the current slot of the schedule. Consequently, the maximal delay to access the bus is bounded, which allows for the analysis of the worst-case timing behavior [KFM$^+$11, KFM$^+$14, SCT10]. However, TDMA arbitration has the drawback of leaving the bus idle if the owner of the current time slot has no interest in accessing the bus, even if other cores are waiting to access to bus.

This drawback is eliminated by *round-robin* (RR) arbitration. In RR arbitration, the priorities assigned to each core are rotated after each access to the shared bus. This means that at some point, each core has the highest priority, preventing resource starvation. More precisely, the maximal waiting time to access the shared bus is bounded by $\#Cores - 1$ accesses of other cores, where $\#Cores$ is the number of cores connected to the bus. Additionally, RR arbitration is *work conserving*, i.e., the bus is not left idle while there are active requests, which can occur in TDMA arbitration.

2.2 Caches

Over the past decades, the computational power of processors has grown tremendously. In contrast, the access latency and bandwidth of memory modules have not improved at

the same pace. The comparatively slow memory introduces a significant bottleneck, which limits overall system performance. This discrepancy between the processor and memory performance is known as the *memory wall* [WM95, McK04].

To overcome this gap between the speed of processors and the comparatively high latency of memory accesses, caches have been introduced in modern system architectures. Caches are small memories, which are located close to the processor and thus offer low access latencies. Caches improve the average-case performance by transparently storing data predicted to be needed in the near future. This process is known as caching.

Caches can be configured to store either *instructions*, *data*, or they can be *unified* to store both data and instructions. In the following we refer to the cache's contents as "data", but the presented concepts are general and apply to all types of caches.

When employing a cache, the memory address space is partitioned into *cache blocks*, which encompass multiple bytes.

Definition 1 (Cache Block [PH12])**.** *A* ***cache block*** *is the smallest unit of information that may be either present or not present in the cache.*

The size of the cache blocks is a power of 2, commonly ranging between 32 and 64 bytes. Blocks are aligned in memory according to their size. A cache block may also be referred to as a *cache line*.

Note that the address space partitioning into cache blocks is particular to the considered cache. Consider a system featuring two caches, e.g., separate instruction and data caches with differing cache line sizes. Due to the change in line size, the address space partitioning is different when considering instruction or data accesses.

When a data request can be served from a cache, the access is said to result in a *cache hit*, otherwise it is called a *cache miss*. The latency of a cache hit is orders of magnitude smaller than a cache miss. Consequently, by intelligently placing (copies of) data in a cache, the performance of a system can be increased significantly.

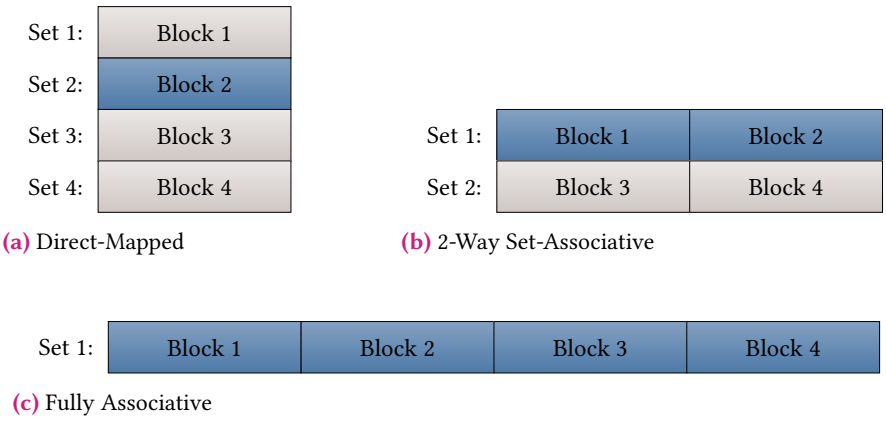
The remainder of this section introduces the technical background needed to understand caching techniques, which are relevant in the context of this thesis.

Locality

Caches generate performance improvements in the average case by exploiting the locality of memory accesses [JNWR08]. There are different types of locality, which are discussed in the following paragraphs.

Temporal locality denotes the observation that the data that is used currently, will likely be needed again in the near future. To leverage this phenomenon, caches store recently used data, reducing the latency of future memory accesses to the same data. The limiting factor to exploit temporal locality is the overall size of the cache.

The second locality effect is called *spatial* locality. Spatial locality denotes the observation that data, which is located closely together in memory, is often used together. When the memory is accessed at a particular location, it is likely that the data located next to the initial target location is accessed in the near future. Caches exploit spatial locality by not only caching the data at the requested memory address but also surrounding memory locations.



(a) Direct-Mapped

(b) 2-Way Set-Associative

(c) Fully Associative

Figure 2.2. – Organization of four cache blocks for (a) direct-mapped, (b) 2-way set-associative, and (c) fully associative cache structures. Example locations in which a new cache block could be placed are highlighted blue.

This is realized by grouping multiple bytes into a single cache block. When a memory location is requested by the processor and the request results in a cache miss, the cache will fetch the complete cache block associated to the requested location. Limiting factors to exploit spatial locality are the size of a cache block and the program behavior.

Internal Cache Organization

Caches are able to simultaneously store multiple cache blocks. The number of cache blocks that fit into the cache is the quotient of the total cache size and the cache block size. When performing a line fill, the cache needs to decide in which part of the cache the fetched block should be stored. This decision is made using the memory address of the block.

Cache blocks can be organized in three different ways inside a cache: If a cache block can be stored in exactly one location, the cache is called *direct-mapped*; when a cache block can be stored in any location, the cache is *fully-associative*. Otherwise, when a cache block can be stored in a restricted number of locations, the cache is called *set-associative*. The set of locations in which a particular block can be placed form a *cache set*. The number of sets a cache is divided into is a key characteristic of the cache. The number of locations that belong to a cache set is called the *associativity*.

Examples for different cache configurations containing four cache blocks are shown in Figure 2.2. The figure visualizes: (a) a direct mapped cache, (b) a 2-way set-associative cache, and (c) a fully associative cache. Suppose a new cache block should be stored in each of these configurations. In a direct-mapped cache, there is only a single place where this block could be stored according to its address. In this example, this is highlighted by the blue color of block 2. In the set-associative configuration, the block may be placed in the first set, but this set contains two individual cache blocks: block 1 and block 2. The block may be placed in either of these locations. In the fully associative cache, the block may be positioned in any of the available four locations. The procedure used to derive the corresponding cache set for a particular block using its starting address is described in the following.

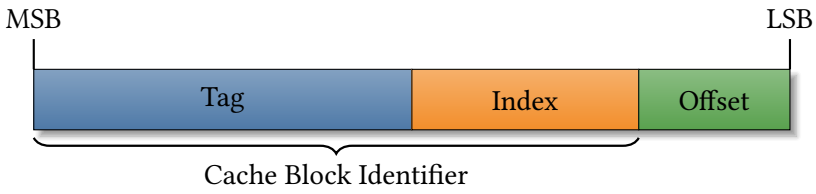


Figure 2.3. – Decomposition of an address into offset, index, and tag. The most-significant bit (MSB) is on the left-hand side; the least-significant bit (LSB) is on the right-hand side. The tag and index form the cache block identifier.

Cache Lookup

When a memory request is passed to a cache, the cache needs to determine whether the requested data is present in the cache or whether a cache line fill has to be performed from the backing store before the request can be answered. This process of determining the cache set corresponding to a cache block, which is performed on every cache access, is explained in this section.

Upon accessing a cache, the target address is split into three components. Starting from the most-significant bit towards the least-significant bit, the address is decomposed into the *tag*, the *index*, and the *offset* [HP11]. This decomposition is visualized in Figure 2.3. If the cache is fully-associative, there is only one cache set that encompasses all cache lines and there is no need for an index.

The offset is used to decide which part of the cache block is requested. This means that the size of the offset is determined by the size of a cache block. Specifically, for a byte-addressable architecture, the offset size is the binary logarithm of the cache block size. The offset bits occupy the least-significant portion of the target address. The bits following the offset are used as the *index*. The index is used to decide which cache set is responsible to handle the cache block. The number of bits required for the index is thus dependent on the number of cache sets. The remaining bits of the target address are used as the *tag*, which distinguishes different cache blocks inside a cache set.

Together, the tag and the index uniquely identify each cache block. The *cache block identifier* is highlighted in Figure 2.3. Hence, the cache block identifier consists of the most-significant bits, excluding all bits of the offset, which are required to reference the desired data inside the cache block.

Figure 2.4 illustrates the process of performing a cache lookup in a direct-mapped cache. Using the index, the correct cache set is selected and activated. Then the tag of the stored cache block is retrieved and compared to the tag derived from the address. If there is a match between the stored tag and the address tag, the access has resulted in a cache hit and the cache can provide the requested data. Finally, the offset is used to determine which part of the cache block was requested.

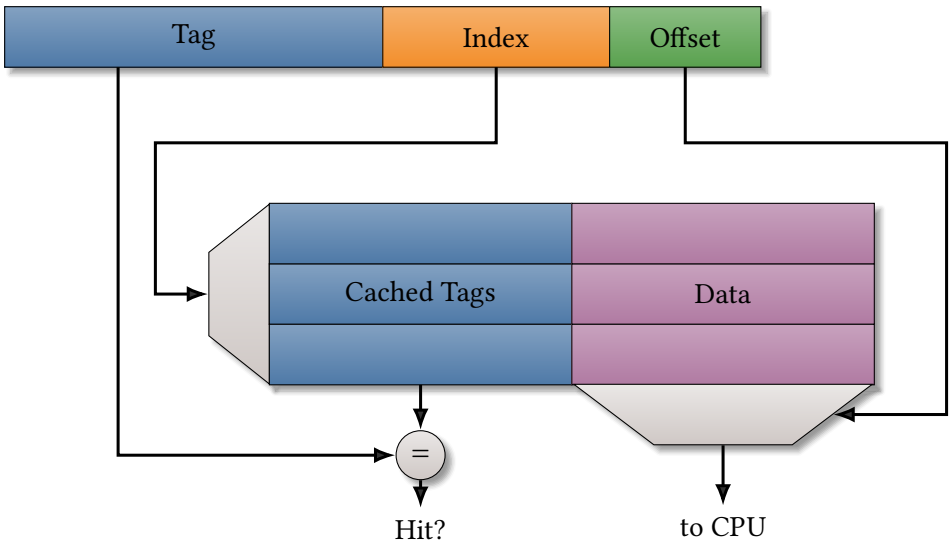


Figure 2.4. – Schematic cache access procedure for a direct-mapped cache. The index is used to determine which cache set is accessed, and the tag is used to check for a cache hit. If the access results in a hit, the offset is used to select the targeted byte from the cache line.

Cache Lookup with Virtual Memory

Modern architectures often feature *virtual memory*. Virtual memory disconnects the address space used by the process from the physical address space used to access the memory [HP11].

Disconnecting the address space into a virtual address space and a physical address space has the benefit of isolating the address spaces of different processes as these processes can have different virtual address spaces. Additionally, this separation allows location independent code, i.e., a process can be placed at an arbitrary location in physical memory, without needing to modify the (virtual) address values in the application's code.

In systems with virtual memory, virtual addresses must be translated to physical addresses on a memory access. The variable mapping from virtual to physical memory locations is handled by the memory management unit (MMU) during run time, by translating addresses from the virtual space to the physical space.

This translation procedure introduces a critical design decision for a cache implementation: Should the virtual or physical address be used to perform the cache lookup? As we will discuss in the following paragraphs, this decision directly impacts the latency and predictability of the cache.

There are two steps in accessing a cache: indexing the cache to select the correct set where the desired data may reside, and comparing the tags present in the selected cache set to the tag of the targeted cache block. Both of these two steps can be performed using either the virtual or physical address. Thus, there are four different ways to caches can be accessed when using virtual memory: 1. virtual index and virtual tag (VIVT), 2. virtual index and physical tag (VIPT), 3. physical index and virtual tag (PIVT), 4. physical index and physical tag (PIPT) [CD97a, CD97b].

A fully-virtual, or VIVT, cache has the advantage that the virtual address does not need to be translated to the physical address before performing the cache lookup. However, as the virtual address spaces of different processes are not related, two processes can use the same virtual address to refer to a different physical memory location. This address sharing phenomenon is known as a cache *homonym*.

In order to prevent processes accidentally using cache contents from another process, the cache lines can be extended so that an *address space identifier* is used in addition to the cache block tag. Another way to prevent homonyms is to invalidate the cache on a context-switch to a different process.

In contrast to fully-virtual addressing, the cache may be accessed by fully-physical, or PIPT, addressing. The PIPT configuration inherently prevents the existence of homonyms. Thus, the cache requires neither address space identifiers nor invalidation on a context-switch. However, for a PIPT cache, the virtual address needs to be translated to the physical address in the MMU before the lookup process in the cache can begin. This increases the cache access latency.

The PIVT addressing mode, does not offer any benefits over PIPT caches, because the physical address is needed for the indexing, which is the first step in the cache lookup procedure. Hence, when the physical address is already available during the indexing, it may also be used for tagging.

A beneficial hybrid of VIVT and PIPT caches is the VIPT cache configuration. The activation of the correct cache set can be performed immediately using the virtual address. While this action is performed, the virtual address is translated in parallel by the MMU. Then, the comparison of the tags is performed using the physical address. By performing the address translation in parallel, the latency penalty of the PIPT configuration can ideally be avoided.

However, VIPT caches can contain *synonyms*. A synonym occurs when the same physical address is translated to different virtual addresses. A VIPT cache may thus run into the situation that the same physical memory gets mapped to different cache sets, depending on the virtual address space. Hence, two processes can effectively cause two copies of the same data to be stored in the cache. As a result, cache synonyms cause data coherency issues.

Cache synonyms can be avoided by ensuring that the address bits used to form the index are identical for the virtual and physical address. This is the case when the index bits are part of the page offset from the virtual memory system. In this configuration, the VIPT and PIPT approaches are identical. Alternatively, the operating system can perform page coloring to prevent this problem. Page coloring is a technique that ensures that the virtual and physical address bits for the cache index are identical by setting the page number accordingly [CD97a].

Replacement Policy

When there is no space left to store a cache block on a cache line fill, another cache block needs to be removed from the cache to make room for the new data. The process of clearing a cache line is called an *eviction*. In case of fully-associative and set-associative caches, there are multiple potential locations to place the new cache block, as visualized in Figure 2.2. Thus, a particular block needs to be selected for eviction from the targeted cache set.

There are different heuristics to decide which cache block should be evicted. The goal of these heuristics is to achieve the highest possible system performance by minimizing the number of cache misses. However, because the target addresses of future memory accesses are not known, the heuristics have to make an educated guess. The heuristic implemented by a cache is called its *replacement policy*. We will shortly introduce different cache replacement policies in the following paragraphs.

The *optimal* replacement policy is a theoretical policy that cannot be implemented in a real system. The optimal replacement policy assumes that there exists an oracle which can determine the cache block that will be not needed for the longest amount of time. Although it is impossible to implement, it is sometimes used to compare the performance of a given replacement algorithm to the theoretical optimal behavior.

The *random* replacement policy randomly selects a cache line to evict. Random replacement is featured in modern processors designed for real-time systems [Arm14]. Analyzing the worst-case behavior of a random cache requires statistical analysis, which computes a probability that data is contained in the cache. Consequently, the timing analysis is also only probabilistic, resulting in some probability with which the deadlines may be violated [KAQC13, SKA+14].

The *first-in-first-out* (FIFO) replacement policy orders cache blocks in a cache set so that on an eviction the oldest cache block is replaced. The *least-recently-used* (LRU) policy is similar to FIFO replacement, with the modification that an access to a cache block which is already stored in the cache refreshes that block, i.e., the accessed block is moved to the youngest position in the set. The LRU policy has the highest predictability of cache replacement strategies [RGBW07]. Its high predictability allows a cache analysis to gather precise information on the possible cache states during runtime. This is contrasted to random replacement, which is less predictable [Rei14]. Due to its high predictability, LRU replacement is recommended for real-time systems [WGR+09, CFG+10]. Consequently, this thesis focuses on caches using the LRU replacement policy.

Multiple Cache Levels

Multiple caches can be arranged to form a hierarchy of caches. The idea behind a cache hierarchy is to have multiple caches organized in sequence. Going from the processor towards the memory, the caches increase in size but lose some of their low latency in return. It is common to find up to three cache levels in contemporary architectures. The cache levels are numbered according to their proximity to the processor. The cache closest to the processor is called the first-level, or L1, cache. The cache following the L1 cache is called the second-level, or L2, cache. The final cache level in front of the memory is commonly referred to as the last-level cache (LLC).

In a multi-level cache hierarchy, a memory access is first passed to the L1 cache. If the L1 cache does not contain the requested data, the request is passed on the L2 cache. This procedure is repeated the hierarchy until a cache hit occurs, or the final cache level misses and the memory is accessed to service the request. Figure 2.5 visualizes the memory hierarchy of a single-core processor with a two-level cache hierarchy.

Figure 2.5. – Example architecture of a single-core system with a two-level cache hierarchy.

A novel challenge that arises from cache hierarchies is that of the *inclusion* policy. The choice for the inclusion policy determines whether data is forced or allowed to be duplicated between cache levels. There are three different possible inclusion policies. These policies are called *inclusive*, *exclusive*, and *non-inclusive* [BJ19].

In an *inclusive* cache hierarchy the higher cache levels are inclusive of the lower cache levels. This means that if a datum is contained in the L1 cache, it must also be contained in the L2 cache. In an *exclusive* cache hierarchy the higher cache levels are exclusive from the lower cache levels. Hence, if data is contained in a particular cache level, it must not be contained in any other cache level. The restrictions from inclusive and exclusive caches are lifted for *non-inclusive* caches, where duplication of data is neither prohibited nor enforced in different cache levels.

Maintaining the inclusivity or exclusivity constraints during run time incurs maintenance overheads. For example, consider the scenario that in an inclusive two-level hierarchy, data is evicted from the L2 cache. As the caches are inclusive this same data is also contained in the L1 cache. However, due to its eviction from the second-level cache, this data has to also be evicted from the L1 cache. Notifying the L1 cache to evict this data creates additional latency. Furthermore, inclusivity effectively wastes cache space as data is forced to be duplicated in multiple cache levels.

The context-switching costs occuring in a non-inclusive cache hierarchy, as shown in Figure 2.5, are analyzed in Chapter 8 and a memory layout optimization for the shown architecture is presented in Chapter 9.

Shared Caches

In multi-core architectures, the last-level cache is often shared between cores. Figure 2.6 visualizes an example for such an architecture. The shown architecture contains four cores, each with its own private L1 cache. The cores are connected via a shared bus to the shared L2 cache and main memory. This means that all cores effectively share the space of the L2 cache. Consequently, inter-core interference occurs, as introduced in Section 2.1. Suppose the program executing on core 1 stores data in the shared L2 cache. Then, another program running on core 2 makes memory requests, which are also cached in the shared L2 cache. If a sufficient amount of data is stored in the cache by the program running on core 2, the data initially cached by core 1 is going to be evicted. This means that the execution of the program on core 1 is slowed down by the inter-core interference of the programs running in parallel. Consequently, the potential inter-core interference needs to be quantified in order to derive safe bounds on the worst-case timing behavior. In Chapters 6 and 7, we present an analysis of the shared cache behavior for systems instantiating the architecture shown in Figure 2.6.

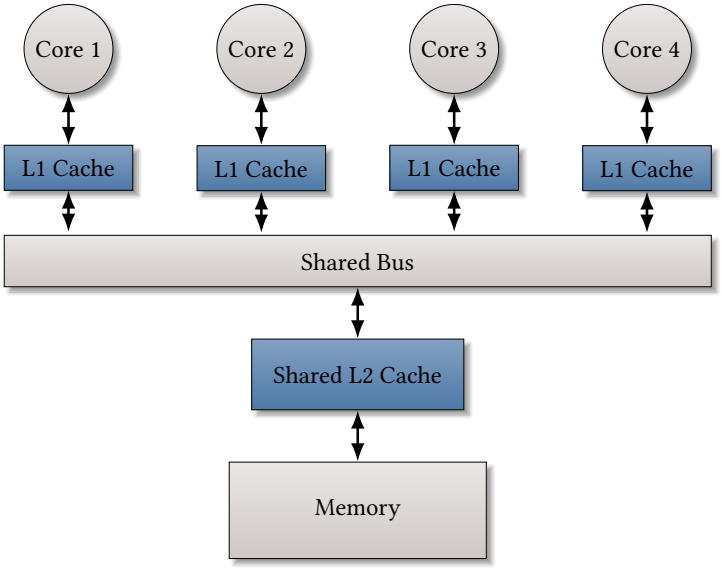


Figure 2.6. – Example multi-core architecture with two-level caching. Each core has a private L1 cache. Accesses resulting in L1 misses are passed via the shared bus to the L2 cache, which is shared among all cores.

Timing Analysis

3

To determine whether a system satisfies its real-time requirements, the timing behavior of the system needs to be analyzed. Specifically, the timing of the system needs to satisfy all deadlines even in the worst-case scenario. Naturally, the worst-case timing behavior of a system is induced by the characteristics of the utilized hardware architecture. It is thus crucial to correctly model the hardware architecture in the timing analysis.

In this chapter, the timing behavior of a program executing in isolation is analyzed. Section 3.1 clarifies the goal of the timing analysis. Furthermore, the complexity of timing analysis in general is highlighted. Section 3.2 introduces the timing analysis workflow employed in this thesis.

3.1 The Worst-Case Execution Time Problem

Verifying the timing behavior requires knowledge of the longest processing duration until the analyzed program terminates. This worst-case processing duration is called the *worst-case execution time*.

Definition 2 (Worst-Case Execution Time)**.** *The **worst-case execution time** (WCET) of a program on a particular hardware platform is the **maximal** duration required for a complete execution of the program for any input arguments and initial hardware state, under the restriction that no other program interferes with its execution, i.e., the program under analysis is not interrupted during its execution and there are no delays due to the concurrent execution of other programs.*

In real-time systems, it can also be helpful to have information on the behavior of the system in the best-case scenario. The *best-case execution time* is defined similarly to the WCET:

Definition 3 (Best-Case Execution Time)**.** *The **best-case execution time** (BCET) of a program on a particular hardware platform is the **minimal** duration required for one complete execution of the program for any input arguments and initial hardware state, under the restriction that no other program interferes with its execution.*

Determining whether a program will terminate within a given time frame is a special case of the *halting problem* [Tur37]. The halting problem asks whether a given program will terminate and has been shown to be undecidable. Consequently, it is, in general, impossible to determine the WCET of a program. Otherwise, we could use this WCET analysis to solve the halting problem by checking whether the WCET is finite.

However, for a restricted subset of all programs, it is possible to determine an upper bound on the WCET. This is the case if upper bounds on the numbers of loop iterations and

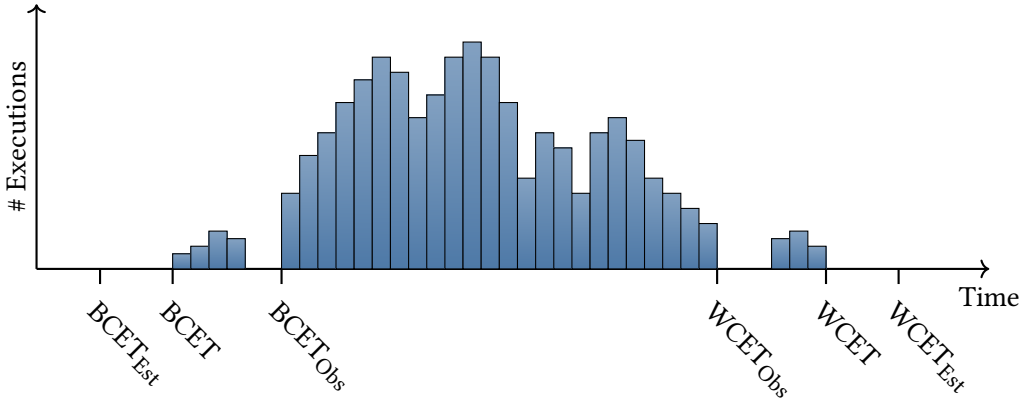


Figure 3.1. – Hypothetical distribution of a program's required execution time for different initial states. Upper and lower bounds on execution time observed from a subset of initial states are marked by $WCET_{Obs}$ and $BCET_{Obs}$, respectively. The actual worst- and best-case are marked by WCET and BCET, whereas the estimations derived by static analysis are marked as $WCET_{Est}$ and $BCET_{Est}$. Adapted from [WEE+08].

the maximal recursion depths are known, for example due to a limited range of possible input parameters. Combining this information with timing information of the hardware architecture allows us to establish an upper bound on the WCET. This approach is called *static* worst-case execution time analysis. We denote the upper bound on the WCET derived using static analysis by $WCET_{Est}$.

The counterpart to static analysis is the *dynamic* analysis approach, which aims to determine the WCET experimentally by running the program on the hardware and measuring the elapsed time [WEE+08]. However, for hard real-time systems dynamic analysis is not suitable due to the vast state space of possible input arguments and feasible initial hardware states. Exhaustively testing all combinations of input variables and hardware states to determine the WCET is infeasible. Furthermore, when measuring only a subset of all possible initial configurations, it is impossible to know whether the worst case is contained in the sample runs. Thus, dynamic analysis is too unreliable for hard real-time systems, where guaranteed timing behavior is required. Hybrid approaches, which combine aspects of static and dynamic timing analysis, have been proposed also [SCR+14, AHQ+15].

A distribution of execution times for different input arguments and initial hardware states is visualized in Figure 3.1. In the example, the worst-case observed in empirical measurements underestimates the actual worst-case. It is thus not safe to determine whether a real-time system meets all its deadlines simply based on the observed execution times. It can also be seen that the $WCET_{Est}$, which is derived by static analysis, overestimates the actual worst-case. Consequently, a goal of static analysis methods is to get *tight* estimates, i.e. to minimize the overestimation:

$$WCET_{Est} - WCET \rightsquigarrow 0, \tag{3.1}$$

while ensuring that the determined WCET bound is safe, meaning that it never underestimates the actual worst-case:

$$\mathrm{WCET}_{\mathrm{Est}} \geq \mathrm{WCET}. \tag{3.2}$$

The constraints on determining BCET estimates are analogous: the actual best-case should never be over-estimated and the estimation should be as close to the actual best-case as possible.

If a certain failure probability is acceptable for the analyzed system, it is also possible to perform a *probabilistic timing analysis* (PTA) [AHQ+15]. To differentiate probabilistic from non-probabilistic analysis methods, we refer to the latter as *deterministic timing analysis* (DTA).

In probabilistic timing analysis, the constraint given in Equation (3.2) is relaxed so that the bound only holds with a certain probability. The actual execution time of the program may thus exceed the computed probabilistic WCET. However, this shall occur only with a known probability. In contrast to DTA, the aim of PTA is not to derive a single WCET value but a probability distribution of execution times. The tail of the distribution function is then cut so that the failure probability of the WCET bound is sufficiently low for the analyzed application. In particular, the tolerated failure rate of the timing analysis is configured so that it is lower than the rate of hardware failures and lower than the maximal failure rate allowed by the safety certification, e.g., less than 10^{-9} failures per hour [KAQC13]. Probabilistic timing analysis can be performed as an instance of static analysis or dynamic analysis, in which case it is called *measurement-based probabilistic timing analysis*.

In this thesis, we focus on static deterministic timing analysis (SDTA). This is motivated by the fact that the research goal pursued in this thesis is the derivation of upper bounds on the worst-case timing behavior that are guaranteed to be safe.

A commercial tool for static deterministic WCET analysis is *aiT* by AbsInt [FH04]. Research projects on WCET analysis tools include *Chronos* [LLMR07], *Heptane* [HRP17], and *LLVMTA* [HJH+22]. The evaluations in this thesis were implemented in the *WCC* compiler [FL10], which includes an internal WCET analyser that implements concepts presented by Kelter [Kel14].

3.2 Static Deterministic Timing Analysis

In this section, we present the general workflow of static deterministic timing analysis. The presented analysis steps correspond to the workflow of the commercial *aiT* analysis tool [LM11]. A schematic presentation of all required analysis steps is given in Figure 3.2. In the figure, analysis steps are represented by blue rectangles, whereas data is represented by purple trapezoids. The analysis procedure is divided into multiple steps, which are laid out in the following sections.

The first step is to create an analyzable program representation, which is discussed in Section 3.2.1. This step is denoted by *CFG Construction* in the figure. It takes the program to be analyzed as input and creates an analyzable representation of that program. Before presenting the analysis steps following the CFG construction, we introduce the concept of

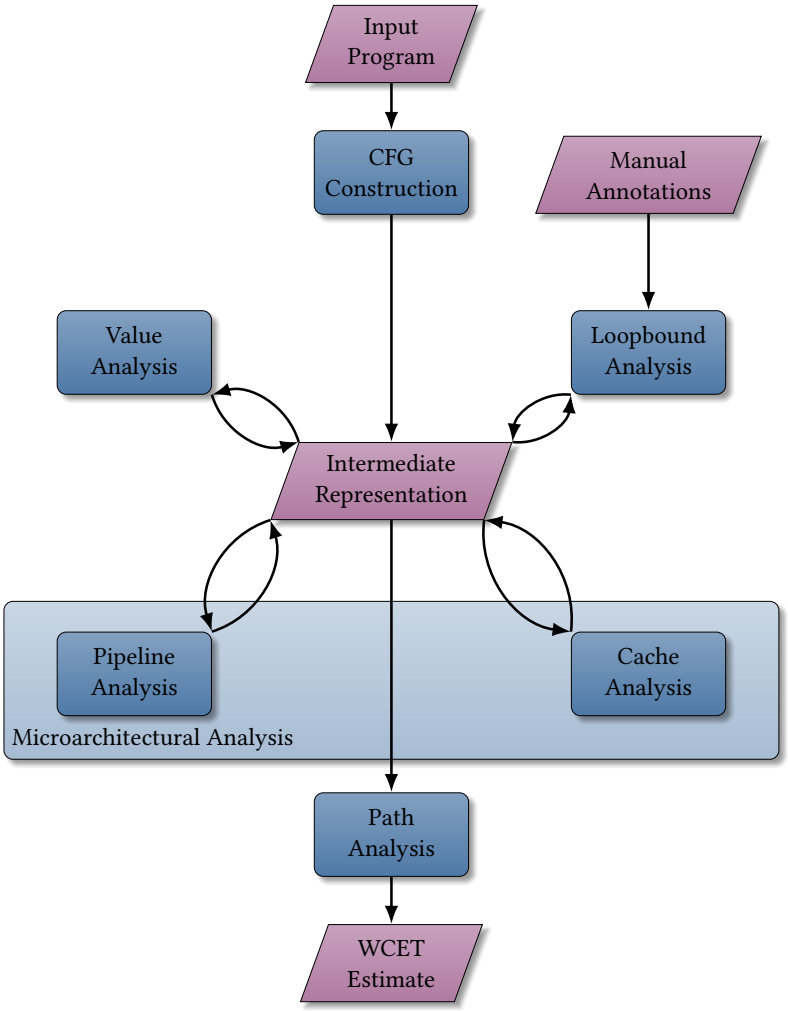


Figure 3.2. – Analysis steps performed in a static deterministic timing analysis. Data is represented by purple trapezoids, whereas analyses are shown as blue rectangles. Analysis steps are performed from top to bottom. The pipeline and cache analyses are substeps of the microarchitectural analysis.

data-flow anlaysis in Section 3.2.2. In Section 3.2.3, the value and loopbound analyses are introduced. These analyses steps derive information on the possible values variables and registers might hold and how many times loops may iterate. In the third step, in Section 3.2.4, the potential hardware states at different points in the program are determined to calculate WCET estimates for individual sections of the program. This step is called the *microarchitectural analysis* and consists of two analysis sub-steps: the pipeline analysis and the cache analysis. Finally, the total WCET estimate for a complete program execution is determined by the *path analysis*. In this step, the timing estimates for the individual program sections are accumulated, while considering the maximal iteration counts of loops and recursion depths of functions. A path analysis technique is described in Section 3.2.5.

3.2.1 Control-Flow Reconstruction

To perform static WCET analysis, an analyzable program representation is required. This representation can be constructed from the binary executable file of the program [LM11, LLMR07] or another low-level program representation [FL10, HJH+22]. Basing the timing analysis on a low-level program representation is critical to produce a safe WCET estimate. This is the case as the exact machine instructions used to implement high-level constructs and the memory layout of these instructions are only known after compilation. The timing behavior of high-level language constructs can vary significantly based on their implementation, which makes it difficult to analyze them without knowledge of the final binary representation.

In the analyzed program representation, multiple machine instructions are grouped together as *basic blocks*.

Definition 4 (Basic Block [Muc97])**.** *A **basic block** is a maximal sequence of instructions that can be entered only at the first of them and exited only from the last of them.*

Grouping instructions in basic blocks reduces the analysis overhead because certain properties only need to be computed at the entry and exit of a basic block (see [AM11]). Let $\mathscr{B}$ denote the set of all basic blocks in the program. To capture the possible execution order of the basic blocks, they are connected to form a directed graph called the *control-flow graph*:

Definition 5 (Control-Flow Graph)**.** *A **control-flow graph** (CFG) is a directed graph that represents all feasible execution paths of the analyzed program. A node in the CFG represents a fragment of the program under analysis. A node b_1 is connected to another node b_2 via a directed edge (b_1, b_2), iff b_2 can be executed immediately after b_1.*

Nodes in the CFG typically correspond to basic blocks. However, a correspondence between nodes and smaller program fragments, such as singular instructions, is also possible. In the following, we focus on CFG with basic blocks as nodes.

If a basic block $b_1 \in \mathscr{B}$ can be directly followed by basic block $b_2 \in \mathscr{B}$ in the task's execution, then there is an edge from the node representing basic block b_1 to the node representing basic block b_2. In the following we denote the CFG over the basic blocks by $(\mathscr{B}, E)$, with the edges being contained in the set $E \subseteq \mathscr{B} \times \mathscr{B}$.

We introduce two functions to denote the predecessors and successors of a particular basic block:

$$pred : \mathscr{B} \to 2^{\mathscr{B}},\ pred(b) = \left\{ b_p \in \mathscr{B} \,\middle|\, (b_p, b) \in E \right\}, \tag{3.3}$$

$$succ : \mathscr{B} \to 2^{\mathscr{B}},\ succ(b) = \left\{ b_s \in \mathscr{B} \,\middle|\, (b, b_s) \in E \right\}. \tag{3.4}$$

To enhance the precision of analyses performed on the CFG, functions and loops can be virtually duplicated within the CFG to support the analysis of different *execution contexts*. Informally, an execution context is defined as the previously executed program fragments that led up to the execution of the considered fragment. There are two types of contexts: the current callstack, and the current iteration count of enclosing loops. Due to the nature of these two context types, this refinement is known as *virtual-inlining and virtual-unrolling* (VIVU) [MAWF98]. Note that applying this refinement does not modify the program under analysis. Only the analysed representation of the program is changed by differentiating between multiple execution contexts. VIVU allows for a more precise analysis of the hardware and program state and, consequently, tighter bounds on the WCET.

A common scenario were VIVU, and loop unrolling in particular, is helpful are memory accesses in a loop. Often a memory access will miss in the cache for the first loop iteration but hit in the cache for subsequent iterations. This behavior can be captured efficiently using VIVU without increasing the mathematical complexity of the other analysis steps.

3.2.2 Data-Flow Anaysis

Information on the program behavior can be derived from the CFG using *data-flow analysis* (DFA). Different types of DFA can be instantiated depending on the type of the desired information. DFA derives information about the program from the semantics of the nodes in the CFG and the structure of the graph, i.e., how the nodes are connected with each other. During the DFA, the derived *data-flow information* is annotated onto the nodes of the control-flow graph.

An annotation of a CFG is a function which maps the locations of the program, i.e., the positions before and after executing a node in the CFG, to the domain of the particular analysis. This information is denoted by $in[b]$ and $out[b]$, for each node $b \in \mathscr{B}$ in the CFG.

Data-flow analysis can be performed to derive different types of information. Examples of DFA include *liveness* analysis and *reaching-definition* analysis [ALSU06]. In the following, we introduce the DFA technique on the example of deriving possible hardware states.

Since multiple distinct feasible states may exist, the annotated information is an element of the powerset of all possible hardware states. This *concrete domain* is ordered by set inclusion to form a partially ordered set. Consequently, the empty set and the set containing all states are the least and greatest element, respectively. Together with the set union and intersection operation, this domain forms a *complete lattice*. We denote the concrete domain by D_C. In the following, we use the subscript C to refer to this concrete domain.

The possible hardware states are propagated through the CFG using two mechanisms. Firstly, the hardware state is transformed according to the semantics of the instructions contained in the basic blocks. This is realized by a *transfer function* $F_C^b : D_C \to D_C$, as is shown in Equation (3.5). The transfer function models the impact of instructions on the

hardware state. Secondly, the information is propagated along the edges in the CFG. For this, the *out* information from predecessor nodes is aggregated to determine the feasible states at the beginning of a node, as shown in (3.6).

$$out[b] = F_C^b(in[b]) \tag{3.5}$$

$$in[b] = \bigcup_{p \in pred(b)} out[p] \tag{3.6}$$

The possible hardware states can be determined by solving Equations (3.5) and (3.6). As the equations are recursive for CFGs containing loops, the equations have to be evaluated iteratively until the results stabilize. However, in practice, this approach is infeasible as the state space is too large to efficiently evaluate all possible hardware states. This presents a significant challenge for the verification of real-time systems, as considering all possible states is crucial to identify the worst-case scenario.

The solution to this problem is found in the theory of *abstract interpretation* [CC77, CC04]. Instead of performing the analysis on the concrete domain of hardware states, in abstract interpretation, a simplified abstraction of the concrete domain is introduced. This *abstract domain* enables efficient analysis while preserving critical information, which is required to determine the worst-case execution time. In the following, we denote the abstract domain by D_A and use the subscript A to refer to operators and elements belonging to this abstract domain.

Compared to the concrete domain, the abstraction sacrifices analysis precision but increases the efficiency, which makes the analysis computable in practice. The concrete and abstract domains are related through *abstraction* and *concretization* functions, commonly called $\alpha : D_C \to D_A$ and $\gamma : D_A \to D_C$. A single abstract state can represent a collection of multiple concrete states, enabling the evaluation of multiple concrete states in a single computation.

The abstract domain is ordered by the partial order $\sqsubseteq$. The *join* operator $\sqcup$ determines an upper bound for two elements of D_A, whereas the *meet* operator $\sqcap$ determines a lower bound. Additionally, the abstract domain contains the least element, called $\bot$, and the greatest element $\top$. $\bot$ signals that no information has been collected yet, whereas $\top$ serves as an upper bound for all other values. Together, these operators and elements form a *complete lattice*.

To compute data-flow information in the abstract domain, an abstract transfer function $F_A^b : D_A \to D_A$ is required for every basic block b. This set of functions encodes the program semantics in the abstract domain for each basic block b according to instructions contained in b. In the abstract domain, the data-flow equations (3.7) and (3.8) use the abstract transfer function and join operator:

$$out[b] = F_A^b(in[b]) \tag{3.7}$$

$$in[b] = \bigsqcup_{p \in pred(b)} out[p] \tag{3.8}$$

Given that the functions F_A^b are monotone, i.e., $a_1 \sqsubseteq a_2 \implies F_A^b(a_1) \sqsubseteq F_A^b(a_2)$, the Knaster-Tarski fixpoint theorem guarantees the existence of a fixpoint [Tar55]. Moreover,

for finite domains, or infinite domains without infinite ascending chains, the analysis terminates in finite time. An ascending chain is a sequence $(a_1, a_2, \ldots)$ with $a_i \sqsubseteq a_{i+1}$. To satisfy the ascending chain condition, it is required that such a chain eventually stabilizes: $\exists\, i : \forall\, j \geq i : a_i = a_j$.

Beyond termination, soundness is a critical property of a data-flow analysis. Soundness ensures that the results of the analysis can be depended upon; without it, the analysis procedure is ineffective as it can not be depended upon. In order for an abstraction to be sound, it is required that the abstraction process does not lose information on the feasible concrete states [Alt12]:

$$\forall d \in D_C : d \subseteq \gamma(\alpha(d)). \tag{3.9}$$

In addition to the conservative abstraction, the abstract transfer function must correctly propagate data-flow information. This property is called local consistency in [Alt12].

To solve the data-flow equations (3.7) and (3.8), initial values are assigned to the *in* and *out* mappings. Typically, an analysis-specific value is assigned to the entry node of the CFG, while other nodes are initialized to $\bot$. The equations are then evaluated iteratively until a fixpoint is reached. An efficient approach to this procedure is the *worklist* algorithm, which is shown in Algorithm 1 [AG04].

The algorithm begins by initialyzing the worklist W to the set of all nodes in the CFG. In this case, this is equal to the set of basic blocks $\mathscr{B}$. The algorithm then enters a loop until the worklist has been emptied completely (line 2). In each iteration a single node is taken from the worklist and analyzed (line 3). The order in which nodes are removed from the worklist W in line 3 is not specified. For acyclic graphs, removing the nodes in topological order is optimal, as every node is evaluated only once [AG04]. In line 4, the previous value of $out[b]$ is saved in the variable *old*. Then, in line 5, the new incoming information from all predecessor nodes $pred(b)$ is determined. By applying the transfer function F_A^b to the newly computed $in[b]$, the novel value for $out[b]$ is determined in line 6. If the outgoing value has changed, all successor nodes of b need to be re-evaluated. This is realized by adding them to the worklist in lines 7 – 9.

Algorithm 1: Worklist Algorithm [AG04]

```
1  W ← B
2  while W ≠ ∅ do
3  |   b ← remove node from W
4  |   old ← out[b]
5  |   in[b] ← ⨆_{p∈pred(b)} out[p]
6  |   out[b] ← F_A^b(in[b])
7  |   if old ≠ out[b] then
8  |   |   W ← W ∪ succ(b)
9  |   end
10 end
```

A DFA can be performed in either the *forward* or *backward* direction on the CFG. A forward analysis operates in the conventional sense of the control flow: information is prop-

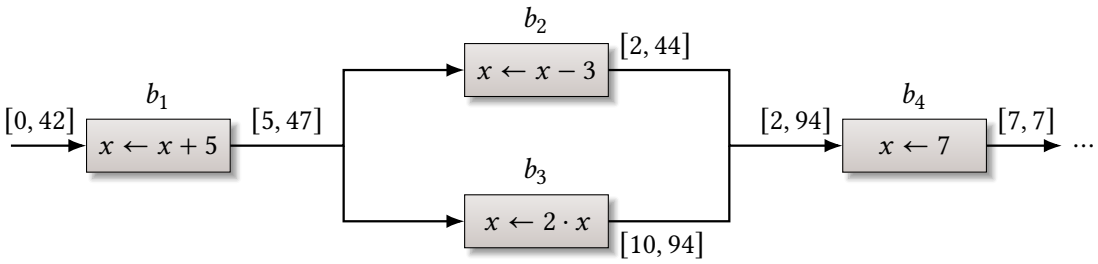


Figure 3.3. – Excerpt of a control-flow graph with annotated value information for the variable x. The feasible values of x are expressed in the abstract interval domain.

agated from the beginning of the program towards the end of the program. In a *backward* analysis, this flow is reversed: the roles of *in* and *out* are switched, and *pred* is exchanged for *succ* in the data-flow equations and the worklist algorithm.

3.2.3 Value & Loopbound Analysis

In this section, we present analyses which are performed as part of the complete WCET analysis procedure. First, the *value analysis* is presented and aftwards the *loopbound analysis* is presented.

One of the steps performed in the WCET analysis is the *value analysis* [Kel14], which determines the values that a variable or register may potentially hold during the program execution. This is especially useful for identifying memory locations accessed indirectly through pointers. Knowledge of the accessed memory locations is vital to facilitate cache analysis, to determine which function is invoked upon calling a function pointer.

To clarify the concepts introduced in the previous section, we present a simple instance of *value analysis* in this section in which we analyze the value of a single 32-bit variable. Interpreted as an unsigned number, the concrete domain consists of sets containing integer numbers between 0 and $2^{32} - 1$. Let D_C^V denote this concrete domain of the example value analysis.

In our example, we define the abstract domain for the value analysis as the set of intervals ranging from 0 to $2^{32} - 1$. The interval domain also includes the empty interval $\emptyset$. Formally, we define the interval domain as:

$$\mathbb{I} = \left\{ [a,\ b] \,\middle|\, 0 \leq a \leq b \leq 2^{32} - 1 \right\} \cup \left\{ \emptyset \right\}. \tag{3.10}$$

Elements of $\mathbb{I}$ are ordered by inclusion. The join operator, which determines the least upper bound of two elements, is defined as follows:

$$[a_1,\ b_1] \sqcup [a_2,\ b_2] = [\min(a_1,\ a_2),\ \max(b_1,\ b_2)], \tag{3.11a}$$

$$\emptyset \sqcup [a_1,\ b_1] = [a_1,\ b_1] \sqcup \emptyset = [a_1,\ b_1] \tag{3.11b}$$

The concretization and abstraction functions are given intuitively by:

$$\gamma_{\mathbb{I}}([a,\ b]) = \{\, c \mid a \leq c \leq b \,\},\ \gamma_{\mathbb{I}}(\emptyset) = \emptyset, \tag{3.12a}$$

$$\alpha_{\mathbb{I}}(d_C^V) = [\min_{c \in d_C^V}(c),\ \max_{c \in d_C^V}(c)],\ \alpha_{\mathbb{I}}(\emptyset) = \emptyset. \tag{3.12b}$$

Figure 3.3 shows an example segment of an annotated CFG. The value of the variable x in the abstract interval domain is annotated at the beginning and end of each block b_1 through b_4. The value at the beginning of b_1, i.e., $in[b_1]$, is $[0, 42]$. This value captures all possible initial values for x. In a complete program, this could be the result of an input parameter dependency. The value is transferred across the block b_1 and consequently gets updated to $[5, 47]$. Afterward, the value of x is propagated to b_2 and b_3, where the block semantics transform the value into $out[b_2] = [2, 44]$ and $out[b_3] = [10, 94]$, respectively. To compute the value $in[b_4]$ both of these values are joined using the $\sqcup$ operator: $in[b_4] = [\min(2,\ 10),\ \max(44,\ 94)] = [2,\ 94]$. Finally, b_4 performs an assignment to x, resulting in the certainty that x contains the value 7. As the shown CFG segment is loop-free there is no need to recompute values due to changed input values.

The goal of the *loopbound analysis* is to determine an upper bound on the number of iterations that are performed by loops in the program. This is a critical step of WCET analysis, because unbounded loop iteration counts can create an arbitrarily large WCET estimate. For simple loop structures, bounds on the iteration count may be derived automatically [ESG+07, LCFM09, GLSB03]. However, loopbound analysis is not complete, in the sense that it may fail to derive some bounds automatically. In this case, manual annotations from the programmer are required. This is visualized in Figure 3.2 by the dependence of the loopbound analysis not only on the analyzed program representation but also additional data, which has to be provided manually [LM11].

3.2.4 Microarchitectural Analysis

In this section, the *microarchitectural analysis* is discussed. The goal of this analysis step is to deduce upper bounds on the worst-case execution time of individual basic blocks.

To determine an upper bound on the execution time of an individual basic block, a timing model of the processor is required. The timing model is constructed using detailed knowledge of the processor's internal workings, as it models all relevant components that influence its timing behavior. Examples for components that impact the timing behavior include the pipeline, branch predictors, caches, and bus arbiters.

At each program location, information about the state of these components is required because different states can result in varying execution times for the same basic block. For instance, on a memory access, the required data may be located in a fast cache, or conversely, may need to be fetched from slow main memory, leading to different execution times due to the different hardware states.

The microarchitectural analysis consists of two analysis steps, as is shown in Figure 3.2. Combining the information collected by the sub-steps of the microarchitectural analysis makes it possible to derive an upper execution time bound for the individual basic blocks.

First, we discuss *pipeline analysis*, which is concerned with the analysis of the behavior of the processor pipeline. Second, we provide a brief motivation of *cache analysis*. Cache analysis is discussed in detail in Chapter 5.

Pipeline Analysis

Modern architectures feature multi-stage pipelines, in which multiple instructions are processed at the same time. To facilitate pipelining, the execution of an instruction is divided into multiple steps [HP11]. In the pipeline analysis, the state of the processor's pipeline is examined to determine the time required to process the program's instructions [The04, Wil07, Kel14]. For example, the ARM Cortex M0 processor features a three stage pipeline [Arm20]. The pipeline stages are 1. *fetch*, 2. *decode*, and 3. *execute*. While the precise definition of the work performed in each stage of the pipeline is not required for the scope of this thesis, the key characteristic of pipelined architectures is that an instruction is processed in the *Execute* stage, while the following instruction is already being processed in the *Decode* stage. In the pipeline analysis, this behavior of the pipeline is determined for the given control-flow graph. As a result, timing information for the processing of instructions and the issuing of memory accesses to fetch instructions and access data is produced.

Cache Analysis

If the analyzed architecture contains caches, the microarchitectural analysis requires another vital step, which is the analysis of the cache behavior. The information produced by the pipeline analysis is used to determine in which order accesses to the cache are issued. This allows the cache analysis to approximate the potential cache states using abstract interpretation. The result of the cache analysis is the latency of individual or sets of memory accesses. As cache analysis is the focus of this thesis, we devote the full Chapter 5 to the introduction of foundational concepts in cache analysis.

3.2.5 Path Analysis by Implicit Path Enumeration

After determining an upper bound on the execution time of each node in the CFG, the final step is to determine the maximal duration of all feasible paths. A path corresponds to a sequence of nodes $(b_1, \ldots, b_n)$ in the CFG, where adjacent nodes are connected $(b_i, b_{i+1}) \in E$, and the path duration corresponds to the sum of the individual WCETs of nodes in the path. A path with maximal duration realizes the worst-case execution time and is called a *worst-case execution path* (WCEP).

The problem of finding a path with maximal duration can be formulated as a linear optimization problem, and solved using *integer linear programming* (ILP). This technique of determining the WCET was developed by Li and Malik [LM97], and is known as the *implicit path enumeration technique* (IPET).

The target function to maximize in the IPET ILP is the sum of the execution duration of all basic blocks. The execution duration of an individual basic block $b \in \mathscr{B}$ is given by the

execution count of the block, denoted by $ec(b)$, multiplied by the WCET estimate derived in the microarchitectural analysis, denoted by WCET(b):

$$\text{maximize} : \sum_{b \in \mathscr{B}} ec(b) \cdot \text{WCET}(b). \tag{3.13}$$

In order to model the program's control flow, bounds on the execution count of the basic blocks are required. Otherwise, the sum in Equation (3.13) would be unbounded, as the execution count of the individual basic blocks could take arbitrary values, leading to an unbounded WCET

The key idea behind the IPET formulation is to transform the flow of the original program into linear constraints on the number of times that the control enters and leaves nodes in the CFG. For regular nodes, i.e., nodes that are neither the start nor the exit of the program, the number of times the control enters a node is equal to the number of times that the node is exited. This constraint resembles Kirchhoff's first law on current in electrical systems.

To formalize this constraint, we utilize the execution count of each edge $e \in E$. Let this execution count be denoted by $ec(e)$. For a basic block b, this flow constraint is given by:

$$\sum_{(b_p,b) \in E} ec((b_p, b)) = \sum_{(b,b_s) \in E} ec((b, b_s)). \tag{3.14}$$

This means that the number of times control is passed from a block $b_p \in pred(b)$ to b is equal to the number of times control is passed from b to one of its successors $b_s \in succ(b)$. The execution count of a block, which is not the program start, is $ec(b) = \sum_{b_p \in pred(b)} ec((b_p, b))$. Calculating the execution count of the first node requires an additional virtual incoming edge to model the starting of the program [OSF18].

While the constraint in Equation (3.14) limits the execution count of individual basic blocks using the structure of the analyzed program, the solution derived from these equations would still be unbounded, as the loops in the model may iterate infinitely many times.

To impose restrictions on the times loops in the program may iterate, the results of the previously performed loopbound analysis are utilized. For example, for a head-controlled loop, the number of times the back edge may be taken in the loop is equal to the loopbound multiplied by the number of times the loop head is entered.

The resulting system of equations can be solved to determine a path with maximal length, and consequently determine the WCET.

To clarify the IPET technique, we construct the ILP for an example CFG. Figure 3.4 shows the exemplary CFG, which consists of 5 basic blocks $b_1, \ldots, b_5$. The basic blocks are connected to form a loop that iterates 10 times from `i` $= 0, \ldots, 9$. After `i` is incremented to the value 10, the loop condition is false and the program returns the value `x`.

The target function to maximize is the execution time of all basic blocks $b_1, \ldots, b_5$:

$$\text{maximize} : ec(b_1) \cdot \text{WCET}(b_1) + \ldots + ec(b_5) \cdot \text{WCET}(b_5).$$

As there is only one incoming and outgoing edge from the block b_1, it follows that: $ec(e_1) = ec(e_2)$. Analogously, it follows that: $ec(e_3) = ec(e_4) \wedge ec(e_4) = ec(e_5)$. For b_2, we know that the

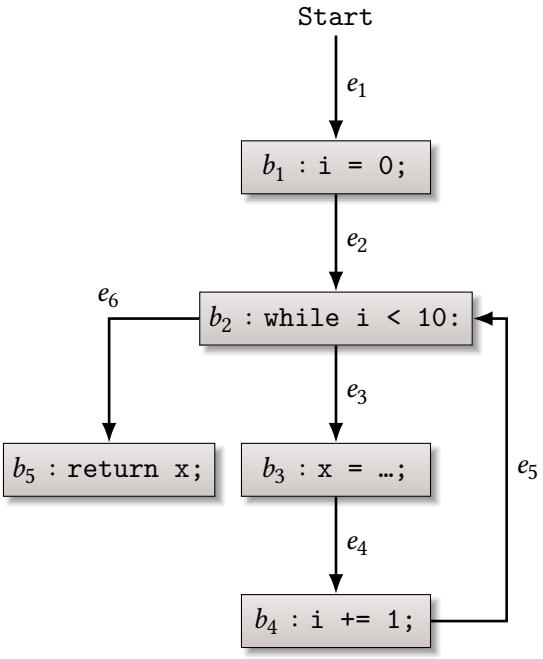


Figure 3.4. – Example CFG for path analysis. The CFG contains 5 nodes, which are connected to form a simple `while` loop.

number of incoming flow is equal to the outgoing flow: $ec(e_2)+ec(e_5) = ec(e_3)+ec(e_6)$. From the loopbound analysis, we know that the loop iterates 10 times. This allows us to add the following constraint on the back-edge e_5: $ec(e_5) = 10 \cdot ec(e_2)$. Consequently, the execution count of the loop-header b_2 is given by: $ec(b_2) = ec(e_2) + ec(e_5)$. This system of equations can be solved to determine the duration of the longest path and compute the WCET of the example CFG.

Thus, by combining the techniques presented in this chapter it is possible to determine the WCET of a program that executes without interruption on a single-core processor.

System-Level Analysis

4

While the previous chapter showed how the worst-case execution time of a single program can be determined using a static deterministic timing analysis, the assumptions of the approach as presented so far are too restrictive for most applications. In particular, a system is typically not only required to execute a single program but has to process multiple programs on a single core.

In this chapter, the analysis scope is broadened from the timing behavior of a single program to a system executing multiple programs. The chapter is structured as follows: In Section 4.1, requisite definitions are given. The order in which programs are executed, which is known as scheduling, is discussed in Section 4.2. The response time, which is the required execution time including interference from other programs, is analyzed in Section 4.3. Finally, we introduce the real-time calculus in Section 4.4, which further zooms out and views the system in terms of resource requests and service volume over time.

4.1 Task Model

From an application-level perspective, the code executed in a real-time system can be divided into one or multiple *tasks*, which collectively enable the system to serve its intended purpose. For example, in an automotive system, these tasks may include speed calculations, obstacle detection, and communication with other subsystems.

Definition 6 (Task)**.** *A **task** is a unit of work that needs to be performed in order for the system to fulfill its intended purpose. The set of tasks in a system is denoted by* $T = \{\tau_1, \ldots, \tau_N\}$.

Each task can be executed multiple times. To distinguish between the individual instances of a task, we introduce the notion of a *job*.

Definition 7 (Job)**.** *Each execution of a task is called a **job**.*

We say that a task gets *released* once another job of the task is ready to be performed. Depending on the release schedule of a task, we can differentiate between three types of tasks: *periodic*, *sporadic*, and *aperiodic* tasks. The categories are defined as follows:

Definition 8 (Periodic Task [Mar21])**.** *Tasks that are released repeatedly, with a fixed time interval between releases, are called **periodic** tasks. The period is denoted by the mapping* $Period : T \to \mathbb{N}$.

Definition 9 (Sporadic Task [Mar21])**.** *Tasks that are released repeatedly, with a minimal time interval between releases, are called **sporadic** tasks. The minimal inter-arrival time is also denoted as* $Period : T \to \mathbb{N}$.

For instance, a periodic task may execute every 50*ms*, while a sporadic task may run repeatedly with at least 50*ms* between jobs but at no fixed interval.

Definition 10 (Aperiodic Task [Mar21])**.** *Tasks that are neither periodic nor sporadic are called* ***aperiodic*** *tasks.*

Aperiodic tasks, in contrast to periodic and sporadic tasks, have unbounded processing demands, as there is no upper limit on their execution frequency. As a result, timing requirements cannot be verified using static analysis for aperiodic tasks. For this reason, this thesis focuses on task sets consisting of periodic and sporadic tasks.

In a real-time system, we can derive a deadline for each time-critical task from the application context.

Definition 11 (Relative Task Deadline)**.** *The* ***relative deadline*** *of a task is the time frame within which a job has to be completed after being released. If the deadline equals the task period, the deadline is called* ***implicit****.*

The system utilization is a measure of the load that is placed on the processor. It provides an indication whether the processor will be mostly idle or will (likely) be overloaded and miss deadlines. Using the periods and WCET estimates for all tasks, we can determine the system utilization of a task set:

Definition 12 (System utilization)**.** *The system* ***utilization*** *U for a task set T is given by the sum of the fractional utilization values of the individual tasks:*

$$U = \sum_{\tau \in T} \frac{WCET(\tau)}{Period(\tau)}. \tag{4.1}$$

4.2 Task Scheduling

When multiple jobs are ready to be executed, the system needs to decide which job should be executed first. This decision-making process is known as *scheduling*. The goal of scheduling is to find an order to process all jobs, such that no deadline is violated. The ability of a scheduling approach to determine such a schedule is called the *schedulability* of the system.

Definition 13 (Schedulability)**.** *A system is* ***schedulable****, if and only if there exists a schedule, i.e., an order of processing jobs, in which no deadline is violated.*

Schedulability can be determined using *schedulability tests*. Schedulability tests can be divided into three categories: 1. *exact* tests, which produce results equivalent to the schedulability of the system; 2. *necessary* tests, which are required to be satisfied for a system to be schedulable; and 3. *sufficient* tests, which are sufficient to guarantee schedulability but may give a false-negative result for some systems.

The remainder of this section is structured as follows: Section 4.2.1 introduces different approaches to scheduling jobs on a single processor core, and Section 4.2.2 expands the scheduling problem to multi-core systems, where not only the ordering of tasks to execute is required but also an assignment of tasks to processor cores is necessary.

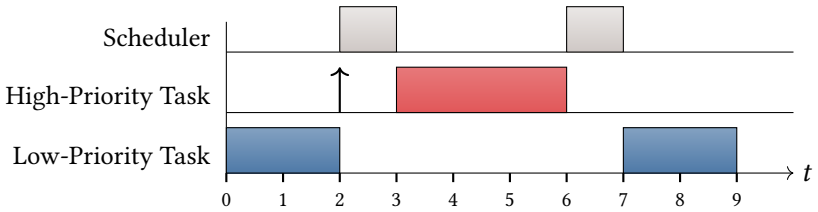


Figure 4.1. – Example of preemptive scheduling. A high-priority task becomes ready while a low-priority task is executing. The scheduler preempts the low-priority task and performs a context switch. After the high-priority task is complete, the low-priority task resumes execution. The preemption allows for a low response time for the high-priority task but necessitates context switches, which incur unproductive overhead.

4.2.1 Single-Core Scheduling Algorithms

In this section, we explore different approaches that can be employed to schedule tasks in a real-time system on a single core.

Scheduling approaches can be divided into two broad categories: *offline* and *online* scheduling. In offline scheduling, the scheduling decisions are made at the system design stage. During runtime, the system executes tasks based on the precomputed schedule [MH21]. The algorithm used to construct the schedule must guarantee that no deadlines are violated during runtime.

The alternative to generating a fixed schedule in advance is to incorporate a scheduling algorithm into the system and to dynamically decide which of the *ready* jobs to execute. This approach has the downside of incurring additional computational overhead during runtime, but it also provides greater flexibility as the exact schedule does not need to be determined when designing the system.

Scheduling approaches can further be categorized by their behavior as either *non-preemptive* or *preemptive*. In non-preemptive scheduling, the scheduler waits until a job is completed before executing a different job. In contrast, in preemptive scheduling the scheduler is permitted to interrupt the execution of the current job, in favor of executing a different task. This process of interrupting the currently executing task is called *preemption*.

Preemptive scheduling offers the advantage of high responsiveness as the scheduler does not have to wait for a job to finish executing to make another scheduling decision. Instead, it can directly react to a high-priority event. Figure 4.1 illustrates a scenario containing two tasks: a low-priority and a high-priority task, while allowing preemptions. It can be seen that the high-priority task becomes ready at $t = 2$. At the same point in time, the scheduler re-evaluates which task should be executed at this point in time. It notices that the high-priority task is ready and performs a context switch at $t = 3$. After completing the high-priority task, the scheduler decides to resume the low priority task because there is no other ready job with higher priority.

The same scenario is shown in Figure 4.2 with a non-preemptive scheduler. The high-priority task becomes ready again at $t = 2$. The scheduler, however, waits until the low-priority task is finished before starting to process the higher priority task. This means that

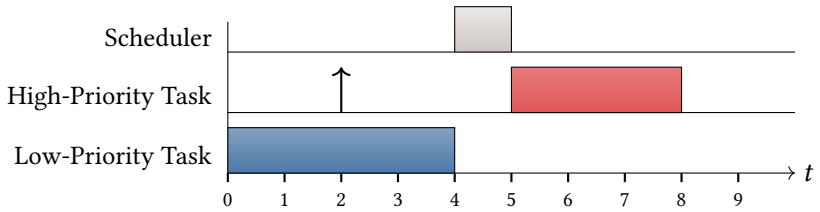


Figure 4.2. – Example of non-preemptive scheduling. The high-priority task becomes ready while the low-priority task is executing. The processing of the high-priority task is delayed until the low-priority task has finished execution. This causes a high response time for the high-priority task.

the time between the release of the high-priority task and its completion is 6 time units compared to 4 time units when using preemptive scheduling.

A drawback of preemptive scheduling is the creation of *context-switching costs*. Suspending the current job and starting another job is not a free operation: the state of the preempted job needs to be saved and, when resuming its execution, restored. A preemption thus delays the execution of the applications' workload due to context-switching overhead.

An online scheduling algorithm must decide which of the currently ready tasks should be processed next. A common approach to make this decision involves assigning each tasks a *priority*, as in Figure 4.1 and 4.2. Priority-based algorithms can either assign a *fixed priority* to each task or vary the priority *dynamically* during runtime. The scheduling decisions are then made to ensure that the job with the highest priority value is being executed.

In fixed-priority preemptive scheduling, the tasks that may preempt another task τ, i.e., are of higher priority, are collected in the set $hp(\tau)$; conversely, the set $lp(\tau)$ contains tasks with lower priority. The set $aff(\tau, \phi)$, for two tasks $\tau, \phi \in T$, is defined as $aff(\tau, \phi) = hp(\tau) \cap lp(\phi)$. $aff(\tau, \phi)$ contains the tasks that may have preempted τ before a preemption by ϕ occurs. I.e., when considering the worst-case impact of a preemption of τ by from ϕ, the tasks in $aff(\tau, \phi)$ may "collaborate" with ϕ to cause additional delays for τ.

An example for fixed-priority preemptive scheduling is *rate-monotonic* (RM) scheduling [LL73]. In RM scheduling each task is assigned a priority according to its period. This creates an ordering of tasks, where the task with the lowest period has the highest priority and the task with the highest period has the lowest priority. The ordering of tasks is fixed, as the period of each task is determined during the system design stage.

In a seminal publication by Liu and Layland [LL73], the following assumptions are made for the time-critical tasks in the system:

1. the tasks are periodic,
2. the deadlines are implicit, i.e. equal to the period,
3. tasks are independent of each other, and
4. the run time of tasks is known and constant.

The final assumption can be relaxed to a known upper bound on the run time, i.e., knowledge of a safe WCET estimate, under the condition that the total context-switching costs

can be bounded or are negligible. This is the case as the delay of the scheduler and the context-switching costs may then be subsumed into the individual task runtimes.

Using these assumptions Liu and Layland showed that RM scheduling is able to schedule all tasks in T without violating a deadline if the processor core utilization is bounded by:

$$U \leq |T| \cdot \left(2^{\frac{1}{|T|}} - 1\right). \tag{4.2}$$

The utilization bound approaches $\ln(2) \approx 0.693$, when the number of tasks tends to infinity. This schedulability test is a sufficient schedulability test. In a practical case study, it has been shown that RM scheduling is able to successfully schedule task sets with an average utilization of 0.88 [LSD89].

Rate-monotonic scheduling is an optimal scheduling strategy for fixed priorities. This means that under the assumptions listed above, if there exists a fixed assignment of priorities that does not cause the violation of a deadline, RM will produce a valid schedule without deadline misses.

A related scheduling approach is *deadline-monotonic* (DM) scheduling [ABRW91]. DM scheduling relaxes the second requirement listed above. Deadlines are no longer required to be implicitly given by the tasks period but may be shorter. Task priorities are derived from their deadline: the smaller the relative deadline, the higher the priority of the task.

However, there are scenarios for which it is possible to find a schedule, but not possible to find a schedule using fixed task priorities. This motivates the utilization of dynamic task priorities.

An example for an online scheduling algorithm with dynamic priorities is the *earliest-deadline-first* (EDF) approach [Hor74]. In EDF scheduling the priorities are assigned dynamically during the execution of the system. The deciding factor for the priority of a task is the remaining time until the deadline passes. The task with the earliest deadline is assigned the highest priority and executed.

Under the assumption that there are no context-switching costs, Horn [Hor74] showed that EDF is optimal with respect to the maximal lateness, i.e., it minimizes the maximal deadline overshoot. This means that if there exists a schedule without any lateness, EDF will produce a valid schedule without any missed deadlines.

4.2.2 Scheduling in Multi-Core Systems

The previous section has looked at the problem of scheduling tasks on a single processor core. However, if the considered architecture features multiple cores that can process a given task, the scheduling problem becomes more complex. Not only the execution order of tasks but also the core that should process the task has to be determined. This additional complication of scheduling for multi-core systems is also known as *task mapping*.

Scheduling approaches for multi-core systems can be divided into *global*, *semi-partitioned*, and *partitioned* approaches. These categories differ in their approach to solve the mapping problem.

In *partitioned* scheduling, the mapping of tasks to cores is determined at design time and is not changed during run time. While the system is running, an individual core executes

a conventional scheduling algorithm to decide which of the tasks that have been assigned to it should be executed next. In general, the mapping of tasks to cores is similar to the bin-packing problem, which is NP-hard [Leu04, p.30-15]. However, many different approaches have been presented to solve the mapping problem for particular hardware architecture configurations [YHZ+09, MTK+11, MIM16].

In contrast, in *global* scheduling a task may be executed on a set of multiple processor cores [LLF+15, ZGW+17]. This means, that the mapping problem has to be solved during runtime, which creates additional overhead and potentially reduces system performance. For example, GEDF is a variant of EDF for multi-core processors using global scheduling. In GEDF, a single priority queue is maintained for all tasks. The GEDF scheduler ensures that at all times the first m entries of the global priority queue are being executed, where m is equal to the number of processor cores.

A case study of an implementation of partitioned EDF and global EDF showed that using partitioned EDF scheduling lead to higher system schedulability compared to global EDF scheduling [GFPF13].

The third category of scheduling approaches is called *semi-partitioned* scheduling [ABD05, AB08, BBA11, BG16]. Semi-partitioned scheduling is a hybrid approach combining aspects of global and partitioned scheduling. In semi-partitioned scheduling, some of the tasks are assigned to fixed cores during the system design phase, while other tasks are allowed to migrate between different cores.

4.3 Response Time Analysis

For a static priority assignment and tasks with deadlines constrained by their period, we can determine the schedulability of the task set using a *response-time analysis* [ABR+93, BMSO+96]. The goal of a response time analysis is to determine an upper bound on the worst-case response time, which is the maximal duration from a task's release to its completion.

Definition 14 (Worst-Case Response Time)**.** *The **worst-case response time** (WCRT) of a task on a particular hardware platform is the maximal duration required for a complete execution of the program for any input arguments and initial hardware state, including all delays and interferences from other tasks.*

The WCRT of a task $\tau \in T$ consists of three components: 1. the worst-case execution time of τ, 2. the blocking time created by other tasks due to resource contention, and 3. the delays caused by higher priority tasks.

Let $\mathrm{WCRT}(\tau)$ denote the worst-case response time of a task $\tau \in T$, let $\mathrm{Wait}(\tau)$ be the cumulative time τ is waiting because another task is executing, and let $\mathrm{Block}(\tau)$ be a bound on the blocking time due to other tasks. The response time of τ is given by the sum of these three components:

$$\mathrm{WCRT}(\tau) = \mathrm{WCET}(\tau) + \mathrm{Block}(\tau) + \mathrm{Wait}(\tau). \tag{4.3}$$

Blocking time occurs when another task has exclusive access to a resource and τ is not granted access to that resource immediately. In particular, in multi-core systems, resource contention from the other cores has to be considered. Thus, an access request to, e.g., a shared bus may not be granted instantly. This delay is accounted for in the blocking time. Moreover, in non-preemptive scheduling, when a low-priority task is being executed, i.e., has exclusive access to the processor core, τ has to wait until the low-priority task has finished executing, thus blocking τ. Waiting time occurs due to higher priority tasks. τ has to wait while these tasks are processed. The waiting time is given by:

$$\text{Wait}(\tau) = \sum_{\phi \in hp(\tau)} \left\lceil \frac{\text{WCRT}(\tau)}{\text{Period}(\phi)} \right\rceil \cdot \text{WCET}(\phi). \tag{4.4}$$

Equation (4.4) accumulates the execution time of all higher priority tasks, which may run during the time frame given by the WCRT of τ. The WCET of each higher priority task is multiplied by the number of times this task may be activated. The activation count of a higher priority task ϕ is derived from its period $\text{Period}(\phi)$ and the response time of τ: $\left\lceil \frac{\text{WCRT}(\tau)}{\text{Period}(\phi)} \right\rceil$.

This definition of the response time is circular, because determining the response time of a task τ requires knowing how many jobs of higher-priority tasks will interfere with τ. However, the number of jobs of higher-priority tasks that may preempt τ depends on the response time of τ, which creates a dependency loop. This circularity can be resolved by computing the response time iteratively. We indicate iterates of $\text{WCRT}(\tau)$ using a superscript $\text{WCRT}(\tau)^{(i)}$:

$$\text{WCRT}(\tau)^{(1)} = \text{WCET}(\tau), \tag{4.5a}$$

$$\text{WCRT}(\tau)^{(i+1)} = \text{WCET}(\tau) + \text{Block}(\tau) + \sum_{\phi \in hp(\tau)} \left\lceil \frac{\text{WCRT}(\tau)^{(i)}}{\text{Period}(\phi)} \right\rceil \cdot \text{WCET}(\phi). \tag{4.5b}$$

The iteration starts at the WCET of τ and can be stopped once the value has converged, or exceeds the deadline of τ. The system is schedulable iff the maximal response time is less-or-equal to the deadline for each task.

Equation (4.5b) assumes that the context-switching costs are negligible or are bounded by a constant so that the costs can be integrated into the task run time. However, the delays incurred from switching between tasks can be substantial and impact the system performance such that deadlines are missed. To account for context-switching costs, they need to be integrated into the response time analysis. Let $\text{CS}_{\tau,\phi}$ represent the context-switching costs when the task τ is preempted by the task ϕ. We can account for the context-switching costs explicitly in the response time analysis by adding them to the execution time of the preempting task [BMSO+96]:

$$\text{WCRT}(\tau)^{(i+1)} = \text{WCET}(\tau) + \text{Block}(\tau) + \sum_{\phi \in hp(\tau)} \left\lceil \frac{\text{WCRT}(\tau)^{(i)}}{\text{Period}(\phi)} \right\rceil \cdot (\text{WCET}(\phi) + \text{CS}_{\tau,\phi}). \tag{4.6}$$

Note that this approach requires *timing compositionality*. A decomposition of a system into multiple components is *timing compositional* if the timing behavior of the complete system can be determined from the timing behavior of its individual components [HRW15]. In this case, the execution time of a task is determined from its WCET without preemptions plus the additional context switching costs.

The overhead of executing the scheduling algorithm itself also has an impact on the response time. The first effect of the scheduler itself is that it requires some time to be executed, as visualized in Figures 4.1 and 4.2. If the scheduler is executed periodically, it can be viewed as the highest-priority task in the response time analysis. Otherwise, its execution time overhead can be incorporated into the execution time bound WCET(τ) for each task in $\tau \in T$. The second effect of the scheduler on the response time of tasks is the delay between a task becoming ready and the scheduler noticing that the task is actually ready to execute. For a periodically executed scheduler, the delay is equal to the period of the scheduler. This delay can be viewed as a component of the blocking time.

Using the techniques presented in this and previous sections, it is possible to determine the worst-case response time of tasks in a multi-tasking system and thus decide whether a set of tasks is schedulable on a hardware platform using a particular priority assignment.

4.4 Real-Time Calculus

Whereas the previous sections focused on the timing analysis of individual tasks based on their control-flow graph, in this section, we take on a different analysis perspective, which can be used to characterize the timing behavior of events occurring in a real-time system. Here, the word *event* can be used to refer to a multiplicity of things, e.g., the release of tasks, the arrival of packets in a network interface, or accesses performed on a shared resource. First, in Section 4.4.1, we introduce the technique in a general context. Then, in Section 4.4.2 we highlight an example of its application in a microarchitectural analysis, which serves as the foundation for the shared cache analysis presented in Chapters 6 and 7.

4.4.1 Foundations of Real-Time Calculus

Real-Time Calculus (RTC) is a discipline in which performance characteristics of real-time systems are determined by analyzing the amount of incoming events and the amount of events that can be handled by the system over time. RTC was proposed by Thiele et al. in [TCN00] to analyze the schedulability of a real-time system and generalized by Chakraborty et al. in [CKT03]. It is a specialization of *network calculus*, which is used to analyze the traffic in networks. Network calculus was developed by Cruz in two articles published in 1991 [Cru91a, Cru91b].

In real-time calculus, the demands placed on the system are modelled using functions of time. The computational demand is quantified using the notion of events. Events can be any discrete state change of interest, e.g., the release of a task, or as in Section 4.4.2, accesses over a shared memory bus.

The occurrence of these events over time form an event stream. From such an event stream, the cumulative demand can be derived. This is shown in Figure 4.3. The individual

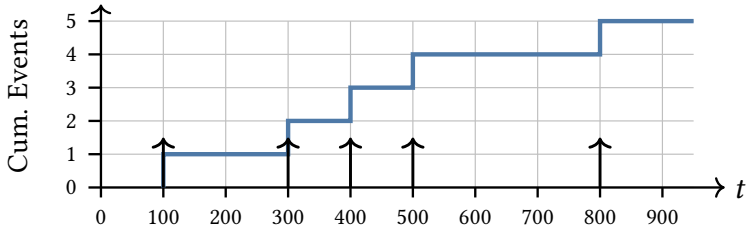


Figure 4.3. – Event stream and cumulative event curve. The x-axis shows absolute time. The occurrence of events is marked by arrows. The blue curve shows the total number of events up to this point.

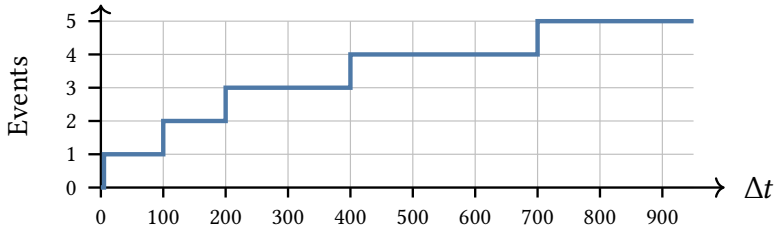


Figure 4.4. – Event-arrival curve for the event stream from Figure 4.3. The x-axis shows the duration of an observation window. The blue curve shows the maximal number of events that can potentially occur for a particular time frame.

events are shown as upwards pointing arrows, while the blue step function keeps track of the total number of events that have occurred so far. Note that the x-axis represents the absolute passing of time. The y-axis shows the number of events that have occurred so far.

Suppose the cumulative number of events occurring at a system is given by a function $R : \mathbb{N}_0 \to \mathbb{N}_0$. Note, we assume that the domain of the function R is discrete, i.e., $\mathbb{N}_0$, representing discrete time steps. However, working with a continuous time domain $\mathbb{R}^+$ is also possible. As we are focusing on computer systems that operate on discrete clock cycles, we are focusing on the discrete domain in this thesis.

Using the cumulative load function of the system, we can analyze how quickly new events may occur in the system using an *event-arrival curve*. The upper event-arrival curve, denoted by η^+, gives an upper bound on the requests that may arrive during any time frame of length Δt:

$$\eta^+(\Delta t) = \max_{t \geq 0} \Big\{ R(t + \Delta t) - R(t) \Big\}. \tag{4.7}$$

Figure 4.4 shows the upper event-arrival curve corresponding to the cumulative demand from Figure 4.3. Note that the x-axis corresponds to the duration of a time frame, rather than absolute time. The curve indicates that a single event may be observed immediately, as even the smallest non-empty time frame can contain a single event. To observe two events, the observed time frame duration must be at least 100 time units. To observe five events, the absolute time frame must contain the interval $[100, 800]$, resulting in the final step of the event arrival curve at $\Delta t = 700$.

Analogously to the upper event-arrival curve, the lower event-arrival curve η^- gives a lower bound on the incoming requests:

$$\eta^-(\Delta t) = \min_{t \geq 0} \Big\{ R(t + \Delta t) - R(t) \Big\}. \tag{4.8}$$

The same perspective can be applied to the system's processing capabilities. The number of serviceable requests in a time frame of Δt cycles are given by the *service curves* β^+ and β^-. Real-time calculus uses these concepts to analyze the timing behavior of the system, such as the backlog of requests or the maximal delay of processing an event.

In this thesis, real-time calculus is employed to model the maximal amount of interference occurring in shared resources. Consequently, only upper event arrival curves are required.

Real-time calculus makes use of a special algebra: the *max-plus algebra* [Lie17, BCOQ01]. In the max-plus algebra, the addition operator is replaced by the maximum operator, and the multiplication operator is replaced by addition: $(\mathbb{N}_0 \cup \{-\infty\}, \max, -\infty, +, 0)$. I.e., the neutral element of the max operator is $-\infty$, whereas the neutral element of addition is 0.

An important operation that can be modelled using RTC is the multiplexing of two event streams into a single combined stream. The behavior of multiplexed streams can be determined by computing the convolution of the two input streams. The operation is performed in the max-plus algebra. Thus, the convolution of two event-arrival curves is a function that determines the maximum number of events for a given time interval Δt. In the convolution, the interval duration Δt is split between both input streams and the maximal number of events for any distribution between both streams is determined. We represent the max-plus convolution by the $\circledast$ operator:

$$(\eta_1 \circledast \eta_2)(\Delta t) = \max_{0 \leq \delta \leq \Delta t} \Big\{ \eta_1(\delta) + \eta_2(\Delta t - \delta) \Big\}. \tag{4.9}$$

4.4.2 Derivation of Event-Arrival Curves

To apply RTC in the analysis of a real-time system, suitable event-arrival curves need to be determined. The curves used for system analysis must be safe in the sense that an upper event-arrival curve must not underestimate the number of potentially occurring events. Several different approaches to estimate event-arrival curves and service curves based on measurements have been proposed [Fid10, FGGH23]. Measurement-based derivation techniques observe the execution of the system and capture relevant information in so-called *execution traces*. From these traces, the curves are derived. However, as with measurement-based WCET analysis, such approaches are vulnerable to underestimation, as the actual worst-case may not be identifiable from a limited set of execution traces.

Oehlert et al. [OSF18] formalized a technique, which can be used to derive event-arrival curves from the task representation used in the microarchitectural analysis. The analysis is static, i.e., it does not require the execution of the task and observation of its behavior. Consequently, the derived curves are known to be safe.

The technique creates an ILP model from the CFG of the analyzed task, similarly to the IPET path analysis described in Section 3.2.5. However, instead of maximizing the execution time contribution of each basic block, the target function of the optimization model is the

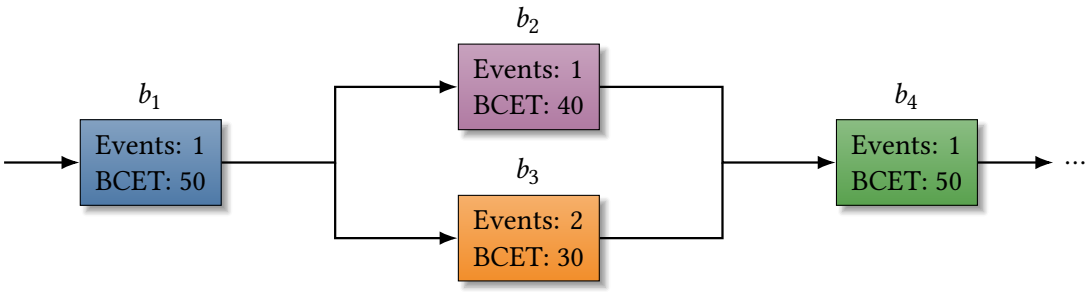


Figure 4.5. – Example CFG annotated with number of events caused per node execution and best-case execution times.

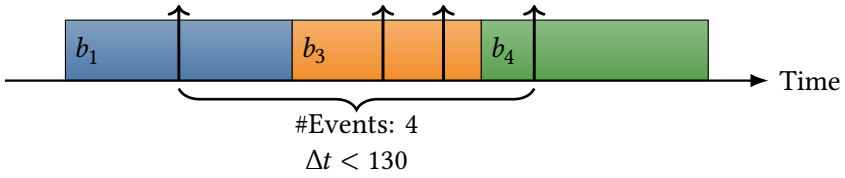


Figure 4.6. – Example execution trace of the CFG shown in Figure 4.5. Occurrences of events are marked by an upward arrow. Events can happen anytime during the execution of a node. This means the time required to observe 4 events is lower than the sum of the node BCETs.

number of events that may occur in a particular time frame. The analysis technique is general and can be applied to various types of events, such as memory accesses, or function calls. It derives the event-arrival curve by annotating the number of events that are triggered by each execution of a basic block to the nodes in the CFG.

A key difference to the IPET analysis is that partial paths in the control-flow graph are considered as well. The IPET analysis aims to determine the maximal path duration of all paths starting at a task's entry node and ending at a sink node, i.e., nodes without outgoing edges. In contrast, for the derivation of event-arrival curves, paths may start and end at any node in the CFG.

Figure 4.5 shows an example of a CFG, annotated for the derivation of an upper event-arrival curves. Each node in the CFG has an associated event count and a best-case execution time. The event count is an upper bound on the number of events that may occur due to the execution of the respective node. Note that the BCET of nodes is used and not the WCET, as the upper bound on the number of events in a particular time frame is realized by quick execution, i.e., the best-case timing, of the nodes in the CFG.

Figure 4.6 shows an example execution trace of the CFG given in Figure 4.5. Note that there is no restriction on the distribution of the events inside the nodes. The event triggered by b_1 happens somewhere in the middle of its execution, while the event triggered by b_4 happens near the beginning of the node. For this reason, it is not sufficient to accumulate the best-case timing of nodes. Consider the scenario that 4 events should be generated. This is satisfied by the (partial) path (b_1, b_3, b_4). The sum of the BCETs of these nodes is 130. However, the four events may be generated more quickly, as can be seen in Figure 4.6, where

the time frame duration Δt is strictly smaller than 130. Consequently, the time contributions of the first and last node in the partial path are reduced to a single cycle to generate safe upper event-arrival curves.

The solution of a single instance of the model gives a single point on the desired event-arrival curve. For the example in Figure 4.4, solving a single instance would, e.g., provide the information that 3 events can occur in an interval of 200 time units. To derive the complete event curve, multiple instances of the model need to be solved.

This approach to derive event-arrival curves was used by Oehlert to model the bus contention occurring in a multi-core system, where multiple tasks want to access the shared bus simultaneously [Oeh21]. In Chapter 6, we extend the derivation technique to analyze interference in a shared cache. We utilize a novel event definition in which the number of events triggered by the execution of a node is not independent of the previously executed nodes. This is a critical feature to capture the inherent behavior of the analyzed cache structure.

Foundations of Cache Analysis

5

This chapter explores the timing behavior exhibited by caches within the context of real-time systems. Understanding the cache behavior is a critical component of the microarchitectural analysis described in Section 3.2.4, as the cache access latency significantly influences the worst-case execution time of individual instructions and basic blocks.

Section 5.1 presents two foundational data-flow analyses: the *Must* and the *May* analyses. These analyses determine which data will definitely, or may potentially, be contained in the analysed cache.

The Must and May analyses determine the cache contents based on the access sequence of a singular task. Thus, these analyses are not sufficient to adequately capture the cache behavior of systems in which multiple tasks execute concurrently. Concurrent task execution occurs in multi-core systems, as well as in preemptively scheduled multi-tasking systems. The concurrent execution of multiple tasks causes mutations of the cache state, which cannot be captured by the Must and May analyses. We refer to these mutations as *interference*. Interference can alter which data is contained in the cache and consequently impact the timing behavior of the system.

We highlight two different types of interference in this thesis. First, Section 5.2 describes inter-core interference, which occurs in caches shared among multiple processor cores, as introduced in Section 2.1. Second, Section 5.3 discusses inter-task interference, which occurs in preemptively scheduled systems, as discussed in Section 4.2.

5.1 Static Analysis of Caches

To analyze the cache behavior statically, i.e., without executing the program, it is essential to know which accesses will occur at run time. To formalize the analysis, we introduce several definitions:

Let Acc denote the set of cache accesses. We assume that each access in Acc targets exactly one cache block from the set of cache blocks $\mathscr{C}$. (We address this restriction in Section 5.1.3.) The function $cb : Acc \to \mathscr{C}$ maps each access to the targeted cache block. We annotate these accesses to the control-flow graph. Let $acc : \mathscr{B} \to \bigcup_{n \in \mathbb{N}_0} Acc^n$ be the function that associates a sequence of cache accesses to each basic block in the CFG.

As discussed in Section 2.2, virtual memory creates a virtual address space, which is separate from the physical address space. In order to perform static cache analysis, it is required to know before run time which cache blocks conflict in the cache. I.e., the mapping of cache blocks to cache sets has to be known. This is the case if the cache is physically indexed, or the virtual address space is predictable, i.e., the index bits of the virtual address are identical to the physical address index bits. The latter can be achieved through careful cache con-

figuration or by employing page coloring, which ensures identical index bits in virtual and physical memory at run time.

The goal of the timing analysis is to derive precise bounds on the latency of cache accesses. A naïve approach might assume that every cache access leads to the worst-case behavior (e.g., a cache miss). While this approach provides a safe WCET estimate, the resulting estimate is overly pessimistic, rendering it unusable in practice. Consequently, a detailed analysis of the cache contents is required to achieve a more precise estimate.

In general, there are two types of cache analyses: *qualitative* and *quantitive* analyses. Qualitative analyses aim to determine specific patterns of cache behavior for individual cache accesses. I.e., the properties of each element in *Acc* are determined, which allows the analysis to draw conclusions on the timing of the containing CFG node. An example for this would be the observation that a particular memory access always results in a cache miss in the first iteration of a loop but will always hit in subsequent iterations.

In contrast, quantitive analyses seek to establish an execution time bound on a set of memory accesses. An example of this would be the observation that there is at most a single cache miss for memory accesses repeated in every iteration of a loop. This result provides no information about when that cache miss will occur—whether it happens in the first or any subsequent iteration of the loop.

The quantitive approach is advantageous in the given example, if the loop is entered multiple times. Due to the bound of a single cache miss, we have a tighter bound on the execution time compared to the qualitative approach, which would have to account for a cache miss for each complete execution of the loop. On the other hand, qualitative information is desirable because it provides deeper insight into the behavior of individual accesses, which is required for further analysis steps like the interference analyses.

5.1.1 May and Must Analysis

In this section we present the *May* and *Must* analyses of Ferdinand and Wilhelm [FW99], which are foundational for qualitative cache analyses. The analyses are applicable to a single cache level; the implications of a multi-level cache hierarchy are discussed in Section 5.1.3. Before defining the May and Must analyses themselves, we first formalize the state of an LRU cache.

Without loss of generality, we assume that the analyzed LRU cache is fully associative, meaning it contains only a single cache set. For caches with multiple cache sets, the analysis can be performed independently for each cache set.

An LRU cache set orders the contained cache blocks based on the recency of the last access to each block. The most recently accessed cache blocks are positioned at the beginning of the sequence, whereas the less recently accessed blocks are positioned near the end. The position of a cache block in this sequence is referred to as the block *age*. The LRU policy replaces the least-recently used block on an eviction.

Consequently, the concrete state of an LRU cache set can be represented by the age of each cache block. Each block is assigned a value according to its position in the sequence of

cached blocks or the value $\mathscr{A}$, which is the associativity of the cache and represents uncached blocks. The set of possible concrete cache states is given by:

$$D_C^{Must} = \left\{ d \,:\, \mathscr{C} \to \{\, 0, 1, \ldots, \mathscr{A} \,\} \;\middle|\; d(b) = d(b') \neq \mathscr{A} \implies b = b' \right\}. \tag{5.1}$$

D_C^{Must} consists of functions mapping the cache blocks to their age, and are injective for all values excluding $\mathscr{A}$.

In the following, we utilize *λ-notation* to concisely define functions. E.g., in *λ-notation*, a function to compute the square of the provided value can be written as $\lambda x.x^2$. The variable x between the λ and the dot is the function parameter, whereas the expression following the dot is the function definition.

An access to a cache block causes a mutation of the concrete cache state. The update of the state mapping d due to an access to b is defined by the higher-order update function $\mathscr{U}_C^{Must}$:

$$\mathscr{U}_C^{Must} \,:\, (D_C^{Must} \times \mathscr{C}) \to D_C^{Must}, \tag{5.2a}$$

$$\mathscr{U}_C^{Must}(d, b) = \lambda b'. \begin{cases} 0 & \text{if } b = b' \\ d(b') & \text{else if } d(b) < d(b') \\ d(b') + 1 & \text{else if } d(b) > d(b') \wedge d(b') < \mathscr{A} \\ \mathscr{A} & \text{otherwise.} \end{cases} \tag{5.2b}$$

There are several cases to be distinguished in the update function: *a*) The just accessed block is moved to the most recent position in the cache and is assigned an age of 0. *b*) If the age of the accessed block b is less than the considered block b', the age of b' does not change because the cache state only changes in the lower portion of the cache state sequence. *c*) The block b' ages by one position if the accessed block b is older than b'. This includes the situation that b was not cached previously. *d*) The block b' remains uncached.

To capture all feasible cache states, subsets of the domain D_C^{Must} are annotated to every program location. However, as each possible state is represented by a unique value the analysis has exponential complexity with respect to the program size. Thus, the analysis quickly becomes computationally infeasible.

To analyze the cache behavior more efficiently, Ferdinand and Wilhelm [FW99] developed the *Must* and *May* analyses. These analyses operate on an abstract domain where each element represents multiple possible concrete states. By reducing the size of the analysis domain, the analysis efficiency increases.

The *Must* analysis determines the maximal LRU age of every cache block. This abstraction can be used to derive definitive cache hits. The *May* analysis works analogously and determines the minimal LRU age of cache blocks over all potential concrete states. This abstraction is used to determine definite cache misses.

We will now focus on the construction of the *Must* analysis. A set of concrete cache states can be abstracted into an abstract *Must* state by determining the maximal cache age for every cache block:

$$\alpha^{Must} \,:\, 2^{D_C^{Must}} \to D_A^{Must} \tag{5.3a}$$

$$\alpha(D) = \lambda b. \max_{d \in D} \{\, d(b) \,\} \tag{5.3b}$$

The corresponding concretization function determines the set of feasible cache states based on the upper cache block age:

$$\gamma^{Must} : D_A^{Must} \to 2^{D_C^{Must}} \tag{5.4a}$$

$$\gamma^{Must}(d_A) = \{\, d_C \in D_C^{Must} \mid \forall b \in \mathscr{C} : d_C(b) \leq d_A(b) \,\} \tag{5.4b}$$

To perform the analysis in the abstract domain, an update function, which models the execution of memory accesses, and a join function, which merges two abstract states are required. The abstract update function is similar to the concrete update function (5.2). The singular modification is the change from the $<$ operator to $\leq$ in the second case. In the abstract domain, we have to pay particular attention to the case where the upper bound on the cache age is equal for two cache blocks. This was not the case for the concrete domain, as we enforced uniqueness of ages less than $\mathscr{A}$. However, in the abstract domain two distinct cache blocks may have the same upper age bound.

If an access is performed to a particular cache block, it is safe to not modify the maximal age for other blocks of equal age. Either the upper bound is realized by the accessed cache block, in which case, the actual age of the other block is strictly smaller and the upper bound remains valid; Or the upper bound is realized by the other block and the accessed block has strictly smaller cache age, in which case there is no need to adjust the upper age bound.

To clarify, consider the example that there are two feasible concrete cache states (A, B), and (B, A), where A and B are two cache blocks. The upper age bound for both blocks is 1. On an access to block B, the maximal age of A would not be modified. This is reasonable as in either case, the feasible concrete state after the access is (B, A), and 1 is a safe upper bound on the cache age of A.

The abstract update function is given by:

$$\mathscr{U}_A^{Must} : (D_A^{Must} \times \mathscr{C}) \to D_A^{Must} \tag{5.5a}$$

$$\mathscr{U}_A^{Must}(d, b) = \lambda b'. \begin{cases} 0 & \text{if } b = b' \\ d(b') & \text{else if } d(b) \leq d(b') \\ d(b') + 1 & \text{else if } d(b) > d(b') \wedge d(b') < \mathscr{A} \\ \mathscr{A} & \text{otherwise} \end{cases} \tag{5.5b}$$

Joining two abstract states is performed by taking the maximal age for each cache block from either state:

$$\sqcup^{Must} : (D_A^{Must} \times D_A^{Must}) \to D_A^{Must} \tag{5.6a}$$

$$d \sqcup^{Must} d' = \lambda b. \max \{\, d(b),\ d'(b) \,\} \tag{5.6b}$$

This abstraction allows for the safe analysis of LRU cache sets with respect to the maximal block age. The *Must* analysis as defined by Ferdinand and Wilhelm [FW99] is not precise

as it can overestimate the cache block age. Touzeau et al. [TMMR19, Tou19] proposed a precise analysis domain, which requires higher computational overhead.

The definition of the *May* domain is similar to the *Must* domain, with the exception that the minimal age is determined. The formal definition of the *May* analysis is omitted here, as it is not required to understand the contributions of this thesis.

5.1.2 Access Classification

The result of the *Must* analysis can be used to determine whether an access to a cache block will result in a cache hit, whereas the result of the *May* analysis can be used to determine cache misses. By combining the results of both analyses, a *cache-hit-miss-classification* (CHMC) can be made [HP08].

The CHMC differentiates between three different categories of cache accesses: 1. always-hit; 2. always-miss; 3. unknown. Let d_A^{Must} and d_A^{May} denote the result of the *Must* and *May* analysis for a particular program location, respectively. The CHMC is defined as follows:

$$\text{CHMC} : \mathit{Acc} \to \{\, \mathit{Hit},\ \mathit{Miss},\ \mathit{Unknown} \,\}, \tag{5.7a}$$

$$\text{CHMC}(a) = \begin{cases} \mathit{Hit} & \text{if } d_A^{Must}(cb(a)) < \mathscr{A} \\ \mathit{Miss} & \text{if } d_A^{May}(cb(a)) \geq \mathscr{A} \\ \mathit{Unknown} & \text{otherwise,} \end{cases} \tag{5.7b}$$

where $cb(a)$ corresponds to the cache block targeted by the access $a \in \mathit{Acc}$.

The resulting CHMC can be used in the microarchitectural analysis to derive bounds on the WCET for individual basic blocks. For accesses classified as *Hit*, it is safe to assume the maximal latency of a cache hit, while for accesses classified as *Miss*, the latency of a cache miss has to be assumed. For accesses classified as *Unknown*, the cache analysis can not make a definitive conclusion regarding the timing of the access. Consequently, the microarchitectural analysis must consider both cases when determining the WCET.

5.1.3 Handling Contingent Accesses

In Section 5.1.1 we note that the definition of the *May* and *Must* analyses assumes a single-level cache. For multi-level caches, the definition has to be refined in order to account for the cache access classification of the lower cache levels.

In particular, the analysis is based on the assumption that cache accesses are performed in every scenario. For example, in Equation (5.5), this assumption is used to reset the cache block age of the targeted block to 0. However, this is not appropriate for a second-level, or higher-level, cache. If the memory access is served by the first-level cache, the state of the other cache levels is not modified. Thus, whether an access is performed on the second-level cache is contingent on the state of the first-level cache.

It is not sufficient to simply remove those cache accesses from the CFG annotation, which result in a L1 cache hit, i.e., $\text{CHMC}(a) = \mathit{Hit}$, and perform the same analysis again for the L2 cache. The reason for this is that accesses with a CHMC of *Unknown* have different effect on the cache state than accesses resulting in a *Miss*.

Hardy and Puaut [HP08] have incorporated this behavior in the cache analysis by introducing *cache-access-classifications* (CAC). The CAC describes whether an access is performed: 1. *always* (A), 2. *never* (N), 3. or sometimes, which is captured as a CAC of *uncertain* (U).

The CAC of an access $a \in Acc$ for a particular cache level $l > 1$ depends upon the CHMC and CAC of the access for the previous cache level $l-1$. Let $CHMC_{l-1}$, and CAC_{l-1} denote the hit-miss classification and access classification for the previous cache level. Then, the access classification of the current level $l > 1$ is given by:

$$\mathrm{CAC}_l(a) = \begin{cases} A & \text{if } \mathrm{CAC}_{l-1}(a) = A \wedge CHMC_{l-1}(a) = Miss \\ N & \text{if } \mathrm{CAC}_{l-1}(a) = N \vee CHMC_{l-1}(a) = Hit \\ U & \text{otherwise.} \end{cases} \tag{5.8}$$

Note that, in particular accesses having a $CHMC_{l-1}$ of *Unknown* have a CAC_l of U. I.e., whether the access to cache level l is performed is contingent upon the state of cache level $l-1$.

The access classification of the first cache level CAC_1 is not influenced by other caches and can be assumed to be A for all annotated accesses, without loss of generality.

The access classification has to be considered in the cache analysis. If the access a is performed always, i.e., $CAC_l(a) = A$, the cache state can be modeled as described previously. On the contrary, if the CAC is equal to N, the access will not occur in any scenario and the state of the cache is not affected. The update function in this case is the identity function. If it is uncertain whether the access will reach the cache, both scenarios have to be considered in the analysis. This can be realized in the analysis by applying the update function to the original state and joining the result with the original state. Let d_{old}^{Must} denote the abstract state of the cache just before the access is performed. Then, we determine the new cache state after the access, denoted by d_{new}^{Must}, using the following equation:

$$d_{new}^{Must} = d_{old}^{Must} \sqcup^{Must} \mathcal{U}_A^{Must}(d_{old}^{Must}, cb(a)). \tag{5.9}$$

A similar approach can be applied when the target address of an access is not known precisely. This is the case when the value analysis is unable to determine the precise target of a memory access. This can be handled by evaluating all possible scenarios and joining the resulting states [Kel14, p. 71].

The analyses introduced in the previous sections allow us to determine the cache behavior for a particular program under the assumption that no interference occurs. Interference is caused by tasks executing on other cores in a multi-core architecture, or from preemptively scheduling multiple tasks on the same core. We discuss the inter-core interference from tasks running on other cores in a multi-core architecture in the upcoming Section 5.2, and the inter-task interference occuring due to preemptions in Section 5.3.

5.2 Inter-Core Interference in Shared Caches

In a multi-core architecture containing a shared cache, the parallel execution of tasks on different cores influences the timing behavior of all tasks accessing the shared cache.

The adverse effect on the cache behavior from multiple parallel task executions is termed *inter-core cache interference*. Inter-core cache interference occurs in a shared cache when tasks running in parallel compete for cache space. This competition for limited resources causes the eviction of useful data from the cache and thus increases the rate of conflict and capacity misses for the shared cache. Consequently, additional delays are created when the data has to be reloaded from memory.

Consider the following scenario for clarification. We assume a dual-core architecture with a shared cache. The shared cache is fully-associative with 4 ways and uses the LRU replacement policy. Let us assume that the first core issues the following sequence of accesses to distinct cache blocks: $S_1 = (A, B, A, C)$. Furthermore, let the second core issue the following accesses to cache blocks : $S_2 = (X, Y, Z, X)$. The logical order in which these accesses arrive at the shared cache is called an *interleaving*. For example, a possible interleaving would be $I = (A, X, B, Y, Z, A, C, X)$.

Depending on the particular interleaving, accesses can hit or miss in the shared cache. The interleaving is a result of the timing of the individual accesses and the bus arbitration policy. Assuming there is no inter-core interference, the second access to A in S_1 and the final access to X in S_2 will result in a cache hit. However, in the example interleaving I, both of these accesses result in a cache miss due to inter-core interference. Consequently, the timing behavior of the shared cache and thus the timing of the task execution on the first core depends on the actions of the second core, and vice versa.

To evaluate the worst-case timing of a task and verify that it does not violate its deadline, all possible interleavings need to be considered. The number of possible different interleavings for two access sequences S_1, S_2 is:

$$\binom{|S_1| + |S_2|}{|S_1|} = \frac{(|S_1| + |S_2|)!}{|S_1|! \cdot |S_2|!}. \tag{5.10}$$

This large number of potential access interleavings makes it impossible to exhaustively consider all interleavings individually. The complexity of the problem becomes even worse when we consider that a task does not issue a fixed sequence of accesses, but the issued accesses can vary from job to job as the path taken through the CFG may differ.

Consequently, which interleaving manifests at runtime depends on a multitude of factors: 1. the (relative) time at which the two tasks were started, 2. the path taken through the control-flow graph of both tasks, 3. the timing of previous computations and memory accesses, and 4. the bus arbitration policy. Kelter [Kel14] showed that exhaustive consideration of all possible interleavings is computationally infeasible even for moderately sized systems.

A straightforward way to solve this problem in cache analysis is to assume that all potential interference from the interfering tasks occurs for every access to the shared cache. This approach has been used by Hardy et al. [HPP09]. The number of cache blocks that may be

accessed by interfering tasks is named the *cache-block conflict number* (CCN). The CHMC is modified to classify an access only as a definite cache hit in the shared cache if the access would be a cache hit without interference and potential interference is unable to cause the eviction of the targeted cache block. This means that the *Hit* condition from Equation (5.7) is modified to $d_A^{Must}(cb(a)) + \text{CCN} < \mathscr{A}$.

Taking interference magnitude to be equal to the number of cache blocks that may be accessed by all interfering tasks is pessimistic. For example, an access by an interfering task to the cache block X may be mutually exclusive with an access to another cache block Y. However, the CCN value would account for both X and Y to potentially cause interference. This would only be possible if multiple jobs of the task cause interference. Furthermore, the interference is assumed to occur for each analyzed access. However, not all cache blocks accessed by the interfering tasks may cause interference in feasible scenarios. Thus, the interference is overestimated by this approach.

To reduce the pessimism of the CCN approach, Liang et al. [LDM+12] determined which of all co-running tasks may actually execute at the same time by examining the schedule of task executions and potential dependencies between tasks.

Other analysis approaches [Nag16, ZLCJ22] pivoted from a qualitative analysis of cache accesses, i.e., deriving cache hit classifications, to a quantitative analysis. This means that an upper bound for the total delay caused by shared cache interference is determined instead of classifying accesses individually.

We show in this thesis that the existing approaches do not offer satisfactory precision. We present a novel analysis approach, which achieves higher analysis precision by incorporating timing information in the classification process. This analysis is presented in Chapters 6 and 7. Chapter 6 presents the general analysis approach and applies it to non-preemptively scheduled systems. Chapter 7 expands the approach to preemptive scheduling.

5.3 Inter-Task Interference in Caches

Another type of cache interference occurs when tasks are scheduled preemptively on the same core. Preemptive scheduling allows for the rapid response to high-priority events as well as the fair usage of resources by all tasks in the system. However, for a system featuring a cache, a preemption has the drawback of disturbing the cache contents, as the preempting task issues memory requests that modify the state of the cache.

When the preempted task resumes its execution, the cache state can differ from the state it was in when the task was initially interrupted. Consequently, preemptions may evict useful data. The preempted task incurs additional delays to its execution because it has to reload the evicted data back into the cache. This additional delay of the preempted task is called *cache-related preemption delay* (CRPD).

Similarly to the effect of inter-core interference, a preemption modifies the sequence of memory accesses issued by the processor core. Let us assume that there are two tasks that need to be executed, with associated memory access sequences $S_1 = (A, B, A, C)$, and $S_2 = (X, Y, Z, X)$. Suppose the first task is preempted after the access to B. The resulting access sequence to the cache is (A, B, X, Y, Z, X, A, C). Consequently, in a 4-way associative LRU

cache with a single cache set, the second access to A results in a cache miss due to the preemption. The execution of the first task is thus delayed due to the cache miss induced by the preemption.

To quantify the potential preemption effects on the cache state, the number of conflicting cache blocks loaded by the preempting task is determined. These blocks are called *evicting cache blocks*.

Definition 15 (Evicting Cache Block [AMR10])**.** *An **evicting cache block** (ECB) of a task τ is a cache block that is potentially accessed by τ during its execution.*

During a preemption, ECBs can cause the eviction of *useful* data from a cache. To quantify the number of cache blocks that have to be reloaded from the backing store in the worst case, the data stored in the cache is differentiated into useful and not-useful data. Data is useful if it may be used in the future and it is currently stored in the cache.

Definition 16 (Useful Cache Block [AB09])**.** *A cache block is a **useful cache block** (UCB) at a program location, iff it is definitely cached at that location and it may be reused at a reachable location without eviction from the cache.*

For LRU caches, the ability of UCBs to resist eviction by ECBs is called their *resilience* [AMR10]. The resilience corresponds to the maximal number of accesses to conflicting cache blocks that can be performed by a preempting task while still guaranteeing that an access to the useful cache block results in a cache hit.

By determining the UCBs, their resilience, and the ECBs of the preempting task, the CRPD can be determined for a preemption by a singular task. In order to model a preemption by multiple tasks, i.e., nested preemptions, the union of ECBs from all preempting tasks needs to be considered.

A detailed discussion of different approaches to determine the CRPD in single-level caches can be found in [Alt12]. Experimental comparisons on CRPD analysis for direct-mapped caches are presented in [SHR18]. The result of the CRPD analysis can then be used to determine the response time using Equation (4.6), as described in Section 4.3.

In single-level LRU caches, the effects of a preemption can be measured directly by the amount that data in the cache has aged due to the preemption. For multi-level caches there is an additional effect on the cache behavior that creates delayed evictions of useful data from the higher-level caches. These evictions and the penalty to reload the data can occur long after the actual preemption has finished.

This second interference effect is called the *indirect interference* of a preemption. It is caused by the increased intra-task interference in the second-level cache, due to the evictions from the first-level cache.

When useful data is evicted from the first-level cache during a preemption, this data needs to be reloaded after the preemption. Reloading the data carries with it a request to the second-level cache, which would not have happened if the task had not been preempted. This access to the second-level cache can in turn contribute to the eviction of other data contained in the L2 cache. When the data evicted from the L2 cache is needed at a later point in time, there is a further delay to fetch it from memory. This is the indirect interference effect.

Indirect interference was first discussed by Chattopadhyay and Roychoudhury in [CR14] for non-inclusive cache hierarchies. An analysis of indirect interference for inclusive cache hierarchies was presented by Zhang and Koutsoukos [ZK16]. A recent analysis for non-inlusive caches was presented by Rashid et al. in [RNT22].

In this thesis, Chapter 8 deals with CRPD analysis for non-inclusive two-level cache hierarchies. We 1. highlight issues in the state-of-the-art analysis by Rashid et al., which cause unsafe results, 2. introduce formal foundations for the analysis of indirect effects, and 3. present a novel, safe analysis with increased precision.

Timing-aware shared cache analysis for non-preemptive scheduling

6

This chapter presents an analysis approach for the inter-core interference occurring in shared caches of multi-core systems. In the approach, techniques from real-time calculus and static code analysis are combined to construct a qualitative cache analysis. Inter-core interference is quantified using event-arrival curves. This means that the aging of data stored in a shared cache is modelled as a function of time. The time between accesses to data in the shared cache is then determined using a data-flow analysis. Together, these two components enable a qualitative analysis of the shared cache. I.e., individual accesses to the shared cache are classified as either definitive cache hits or potential cache misses. The analysis results allow for a safe and precise estimation of the task WCET.

The core idea, to use timing information in the analysis of cache behavior, was initially presented at the *Design, Automation & Test in Europe Conference & Exhibition (DATE)* in 2023 and published as an extended abstract in [FF23b]. This publication is reproduced in Appendix A.

The analysis was then developed further and presented in a more detailed fashion at the *31st International Conference on Real-Time Network and Systems (RTNS)* conference in 2023 and published in [FF23a]. This publication is reproduced in Appendix B.

The analysis was then extended to support non-preemptive scheduling of multiple tasks and published as an article in the *Springer Real-Time Systems* journal [FF24b]. This publication is reproduced in this chapter.

Springer – Real-Time Systems

Manuscript submitted February 28, 2024.
Revised June 13, 2024.
Accepted September 13, 2024.
Published online September 30, 2024.
Journal Volume 60, Issue 4, December 2024.
DOI: 10.1007/s11241-024-09430-8

Real-Time Systems
https://doi.org/10.1007/s11241-024-09430-8

Timing-aware analysis of shared cache interference for non-preemptive scheduling

Thilo L. Fischer[1] · **Heiko Falk**[1]

Accepted: 13 September 2024

Abstract
In multi-core architectures, the last-level cache (LLC) is often shared between cores. Sharing the LLC leads to inter-core interference, which impacts system performance and predictability. This means that tasks running in parallel on different cores may experience additional LLC misses as they compete for cache space. To compute a task's worst-case execution time (WCET), a safe bound on the inter-core cache interference has to be determined. We propose an interference analysis for set-associative shared least-recently-used caches. The analysis leverages timing information to establish tight bounds on the worst-case interference and classifies individual accesses as either cache hits or potential cache misses. We evaluated the analysis performance for systems containing 2 and 4 cores using shared caches up to 64 KB. The evaluation shows an average WCET reduction of up to 23.3% for dual-core systems and 8.5% for quad-core systems.

Keywords Shared cache · WCET analysis · Multi-core · Event-arrival curve

1 Introduction

Real-time systems are subject to timing requirements, meaning that the tasks in a real-time system must be completed before their associated deadline. Otherwise, the system can fail with catastrophic consequences. In order to verify that the timing requirements are met even in the worst-case, static analyses need to be performed on the system. An important property to analyze is the worst-case execution time (WCET) of each task, which describes the longest duration required to complete an instance of that task.

✉ Thilo L. Fischer
thilo.leon.fischer@tuhh.de

Heiko Falk
heiko.falk@tuhh.de

[1] Institute of Embedded Systems, Hamburg University of Technology, Am Schwarzenberg-Campus 1, 21073 Hamburg, Germany

Published online: 30 September 2024

Modern architectures use caches to bridge the performance gap between the fast processor core and the slower memory. Although caches improve the average case performance of a system, careful analysis is required to determine their worst-case behavior. When analyzing the behavior of a cache, *cache-hit-miss-classifications* (CHMC) are used to describe whether an access will be a hit, a miss, or result in unknown behavior. These classifications can be used to derive a safe upper bound on the WCET. Analyses (Ferdinand and Wilhelm 1999; Touzeau et al. 2019) based on the well established framework of abstract interpretation (Cousot and Cousot 1977) have been developed to analyze the contents of caches. These analyses perform data-flow analyses to determine the maximal and minimal intra-task cache interference for each access to a particular cache block. The result of these data-flow analyses directly induces the CHMC classifications when considering the associativity of the cache.

In modern systems, the trend is shifting from single-core to multi-core systems, where the memory hierarchy typically consists of multiple cache levels, with the last level cache shared between cores. The parallel execution of tasks on different cores influences the state of the shared cache, invalidating the cache hit classifications if the additional cache conflicts due to cache sharing are not considered. For this reason, the effects cache sharing have to be taken into account in the system analysis. However, inter-core interference on shared caches is notoriously difficult to quantify. This unpredictability leads to potential overestimation of the worst-case timing behavior. To avoid this overestimation, it is essential to establish a tight bound on the effects of inter-core cache interference.

In this paper, we propose a novel analysis approach to derive cache hit classifications for individual accesses, considering inter-core interference. The approach is applicable to set-associative shared caches using the *least-recently-used* (LRU) replacement policy. To quantify inter-core interference, we model cache accesses issued by each task as an event stream. These event streams are characterized using the concept of event-arrival curves, which have been applied previously in *network calculus* (Le Boudec and Thiran 2001) and *real-time calculus* (Thiele et al. 2000). By applying the concept of event-arrival curves to shared caches, we can examine the inter-arrival time between multiple cache access events and quantify the impact on the shared cache.

This perspective on inter-core cache interference essentially induces a *time-to-live* (TTL) for information stored in the shared cache. The TTL of a cache block corresponds to the time frame during which interfering tasks cannot issue a sufficient number of conflicting accesses to evict the cache block. Consequently, if a block is accessed before its *time-to-live* has expired, the access will definitely result in a cache hit. We use the term *interference curve* for an event-arrival curve describing shared cache interference in this paper.

For a single interfering task, the interference curve can be derived from the control-flow graph of that task. This was first demonstrated in Fischer and Falk (2023). However, when multiple tasks run on a single interfering core, they can collaborate to evict data from the shared cache. In this paper, we extend the concept of interference curves from a single task to multiple tasks executing on the same core using the *max-plus algebra*. This paper focuses on non-preemptive scheduling, as this

application has not been explored previously. Interference curves have been applied to preemptively scheduled systems in Fischer and Falk (2024). Non-preemptive scheduling is of interest for real-time systems as it avoids context switching costs. In the presence of caches, each context switch creates *cache-related preemption delay* (Altmeyer and Maiza Burguière 2011). In particular for the multi-level cache hierarchy considered in this paper, there exist additional *indirect effects of preemptions* that increase the context switching costs (Chattopadhyay and Roychoudhury 2014). Non-preemptive scheduling avoids these delays.

The key contributions of this paper are:

- Formulation of an ILP model to derive event-arrival curves for inter-core cache interference by a single task.
- Modelling of non-preemptive scheduling using interference curves.
- Design of a data-flow analysis that leverages timing information to classify shared cache accesses as cache hits or potential misses using interference curves.
- Correctness proofs for all components of the presented analysis.
- An evaluation demonstrating the scalablility and significant improvements over previous methods (Hardy et al. 2009; Nagar and Srikant 2016).

The remainder of this paper is structured as follows: in Sect. 2, related research is discussed. Section 3 discusses the system model and explores existing analyses for shared caches. Section 4 describes the general workflow of the presented analysis. We introduce the event-arrival perspective on cache interference in Sect. 5. In Sect. 6 accesses to the shared cache are classified as cache hits or potential misses by applying the interference curves in a data-flow analysis. An evaluation using realistic workloads is presented in Sect. 7. Section 8 concludes the paper.

2 Related work

For over two decades, many different facets of cache behavior and cache-related delays have been analyzed. Ferdinand and Wilhelm (1999) introduced *may* and *must* analyses that determine whether data will potentially or definitely be present in the cache. The concrete state of the system is abstracted using *abstract interpretation* (Cousot and Cousot 1977) to the minimal and maximal age of a cache block. Twenty years later, Touzeau et al. (2019) presented an abstraction based on zero-suppressed binary decision diagrams that allows for a precise analysis of may/must information.

The may and must analyses are *qualitative* analyses. Each individual access to the cache is analyzed and classified as either a hit, a miss, or unknown. In contrast, *quantitative* analyses provide an upper bound on the total delay due to some cache behavior. One such quantitative analysis is the *persistence* analysis, which determines whether a set of accesses can result in at most one cache miss (Ferdinand and Wilhelm 1999; Reineke 2018).

A survey of multi-core analysis techniques was published by Maiza et al. (2019), whereas cache specific analyses were surveyed by Lv et al. (2016).

Cache-related preemption delay (CRPD) occurs when multiple tasks are assigned to a single core and preemptions are allowed. CRPD for single-level caches has been studied extensively (Lee et al. 1998; Altmeyer and Maiza Burguière 2011). Chattopadhyay and Roychoudhury (2014) presented the first CRPD analysis for a two-level cache hierarchy, focusing on non-inclusive caches. Zhang and Koutsoukos (2016) developed an analysis for inclusive cache hierarchies. The more recent work by Rashid et al. (2022) introduced the concept of L1-UCBs and L2-UCBs to compute CRPD in a two level non-inclusive cache hierarchy.

Xiao et al. (2017, 2022) integrated the effects of inter-core interference from shared caches directly into the schedulability analysis for non-preemptive systems. They compute an interference penalty by multiplying the number of accesses to potentially evicted cache blocks by a cache miss penalty.

Yan and Zhang (2008) and Zhang and Yan (2009) analyzed direct-mapped shared instruction caches in multi-core systems. Yan and Zhang (2008) differentiate between accesses which are contained in loops and those which are not in loops. An L2 cache hit is degraded to a cache miss if an interfering task accesses the same cache set. Zhang and Yan (2009) models inter-core interference in an ILP to determine the worst-case execution time.

Hardy et al. (2009) analyzed set-associative shared caches in multi-core systems by counting the number of potentially interfering cache blocks. This value is called the *cache block conflict number* (CCN). An access to a cache block is classified as a hit if the number of conflicting blocks is less than the associativity minus the block age. Liang et al. (2012) extended this approach by incorporating a lifetime analysis. The lifetime analysis determines which tasks are running concurrently, as tasks with a disjoint lifetime do not create any mutual interference on the cache. While this approach yields safe results, it is pessimistic as it assumes that an interfering task can potentially issue all interfering accesses concurrently at any point in time.

Kelter (2014) analysed shared caches by evaluating all possible state combinations of tasks executing on different cores. Due to the high analysis overhead, this approach proved to be feasible only for small systems.

Zhang et al. (2022) aimed to reduce the pessimism of the CCN analysis by excluding infeasible interferences. Memory accesses are grouped based on their location in the control-flow graph. This creates a *happens-before* partial order on all accesses contained in a task. Using this ordering, infeasible combinations of interfering accesses are excluded from the interference estimation. The additional execution time caused by cache misses due to interference is computed using the cumulative execution count of all accesses to potentially evicted cache blocks. This upper bound for the additional execution time is named *worst-case-extra-execution-time* (WCEET). They evaluate the approach for dual-core systems and compare it to the CCN approach. The approach is evaluated for a single instance of a single co-running task. Using the MRTC benchmark suite (Gustafsson et al. 2010), an average WCET reduction of 13% is reported. However, in the median only a 1% improvement is achieved.

A similar approach was used by Dharishini and Murthy (2021) to analyze multi-threaded programs running on multi-core systems. Synchronization points

between threads are used to determine which sections of the program may run in parallel in different threads. This information is then used to reduce the number of potentially occurring conflicts.

Nagar and Srikant (2014, 2016); Nagar (2016) approach the cache interference problem from a different perspective. They developed a shared cache analysis by capturing inter-core interference in an ILP model. The total amount of possible interfering accesses originating from competing tasks is statically determined and used to limit the interference experienced by the analysed task. The worst-case distribution of interfering accesses along the control-flow graph (CFG), which results in the largest increase in execution time, is then determined by solving the ILP. The analysis thus determines the *worst case interference placement* (WCIP). This approach does not depend on the exact interleaving of memory accesses issued by different tasks and yielded more precise WCET estimations than the CCN classification method. In addition to the precise ILP model, an approximate algorithm is presented, which exhibits similar precision but lower analysis overhead compared to the ILP model. In this paper, we pursue an orthogonal approach. Instead of limiting the total number of interfering accesses, we examine how quickly a sequence of multiple accesses may be issued.

The two techniques presented by Zhang et al. (2022) and Nagar (2016) do not produce a hit or miss classification for any single cache access but only bound the overall increase in execution time for the complete program. That is, they are quantitative analyses. The problem of classifying each individual access is more complex than determining the WCEET in the sense that given a classification of individual accesses, a safe WCEET value can be determined, but the inverse is not possible. In this paper, we tackle the harder problem of classifying each individual access as a cache hit or potential miss.

Oehlert et al. (2018); Oehlert (2021) presented a method to quantify memory accesses issued by a task using event-arrival curves. Memory access events are derived from the program code at the level of basic blocks. To compute an upper bound on the number of events arriving in a given time frame, an ILP model using *implicit path enumeration* is developed. The event-arrival curves are used to analyze, and subsequently improve, the bus contention in multi-core systems. Based on this work, we develop an ILP model to quantify inter-core cache interference. Instead of quantifying the total number of memory accesses, we analyze how much time is required for a task to access a given number of interfering cache blocks.

3 System model & background

In this section, we discuss the assumptions made by the analysis on the system architecture and describe existing analysis approaches to establish the current landscape of shared cache analysis.

3.1 System model

The system architecture considered in this paper consists of multiple cores with private L1 caches, which are connected to a shared L2 cache via a shared bus. The shared cache uses the least-recently-used (LRU) replacement policy. The analysis is applicable to set-associative shared caches employing the LRU replacement policy with associativity $\mathcal{A}$. As cache sets operate independently of one another, we may consider them in isolation during the analysis. We concentrate on instruction caches in this paper because the memory layout of the program code is known at compile time. However, the analysis approach is general and can be applied to data, instruction, and unified caches.

The analysis requires that the minimal and maximal blocking time for an access over the shared bus is known and finite, e.g. due to round-robin arbitration.

We denote the set of all cores by $\mathcal{C}$. In the remainder of the paper, we use the convention that $C \in \mathcal{C}$ represents the core of the task currently under analysis, while $C' \in \mathcal{C} \backslash \{C\}$ represents an interfering core. The set of tasks executing on a core C are denoted by T_C. We use the convention that $\tau \in T_C$ refers to the task under analysis and $\varphi \in T_{C'}$ refers to an interfering task. An instance of a task is called a job.

In our formalization, we differentiate between the sets $T_{C'}, C' \in \mathcal{C} \backslash \{C\}$ to support partitioned scheduling. Partitioned scheduling refers to the fact that the mapping of a task to the executing core does not change. However, the analysis is applicable both to partitioned and non-partitioned scheduling of tasks. When applying the analysis to non-partitioned scheduling, all tasks $\varphi \neq \tau$ can cause shared cache interference for τ. This needs to be considered when determining the interference curves, which means that the set of tasks potentially executing on a core C' needs to be updated to $\bigcup_{C \in \mathcal{C}} T_C \setminus \{\tau\}$. We note that, due to its higher predictability, partitioned scheduling will likely lead to more precise analysis results.

We impose no restrictions on the order in which tasks are executed, i.e., there are no execution order dependencies between tasks. Each task τ may be executed repeatedly, with a minimal inter-arrival time of $Period(\tau)$ cycles.

The pair (V_τ, E_τ) represents the control-flow graph (CFG) of τ. The set V_τ contains the nodes in the graph, whereas the edges between nodes are contained in the set E_τ.

The set of accesses performed by a task τ to the shared cache are denoted by Acc_τ, whereas the set of accessed cache blocks is denoted by $\mathcal{B}_\tau$. Each access targets a particular cache block, which is given by the function $cb : Acc_\tau \rightarrow \mathcal{B}_\tau$.

Multiple cache accesses may be associated to each node $v \in V_\tau$. However, it is required that the accesses associated to a node are ordered, i.e., they will be issued in a specific order when executing v. This is necessary, as the conflicts occurring between these accesses has to be known during the analysis. This restriction may prohibit speculative execution and out-of-order execution. These features are problematic for a cache analysis, because it is no longer possible to statically predict which accesses are performed or in which order they happen. This is a problem for WCET analyses in general and not just limited to the presented analysis.

The notation introduced above is summarized in Table 1.

Table 1 General notation

Symbol	Meaning
$\mathcal{C}$	Set of all cores
T_C	Set of tasks assigned to core C
$\tau \in T_C$	The task for which to analyze the shared cache
$\varphi \in T_{C'},\ C \neq C'$	A task causing interference at the shared cache
$\mathcal{A}$	Associativity of the shared cache
Acc_τ	Set of all shared cache accesses from τ
$\mathcal{B}_\tau$	Set of all cache blocks of τ
$cb : Acc_\tau \rightarrow \mathcal{B}_\tau$	Mapping of an access to the targeted cache block
$Period(\tau)$	Period of task τ
(V_τ, E_τ)	Nodes and edges in the control-flow graph of task τ

3.2 Background on shared cache analysis

In this section, we will give an introduction into cache analysis and analysis of inter-core interference for shared caches.

Ferdinand and Wilhelm (1999) introduced the *May* and *Must* analyses for LRU caches. These analyses perform a data-flow analysis on the control-flow graph of the analyzed task τ to determine the minimal and maximal age of every cache block at each program location.

The *Must* analysis computes the maximal number of conflicting cache blocks accessed since the last access to a given cache block. As we focus on a single cache set at a time, the *Must* information at particular program location can be represented by a mapping $Must : \mathcal{B}_\tau \rightarrow \{0, \ldots, \mathcal{A}-1\} \cup \{\infty\}$, where ∞ indicates that the block is definitely not contained in the cache. A mapping $Must(b) = n < \mathcal{A}$ shows that at most n conflicting cache blocks have been accessed by the analyzed task since the last access to b. Consequently, an access to the cache block b would result in a cache hit.

The difference between the maximal age of a cache block and the associativity of the cache is called the *resilience*, which corresponds to the minimal amount of additional interference for which a cache miss may occur.

The *May* analysis is the analog to the *Must* analysis in that it computes the minimal age of cache blocks and can be used to deduce definite cache misses. We denote the minimal age of a cache block by the mapping $May : \mathcal{B} \rightarrow \{0, \ldots, \mathcal{A}-1\} \cup \{\infty\}$. For an in depth description of the *Must* and *May* analyses see Ferdinand and Wilhelm (1999). To differentiate between the information regarding the first and second level cache, we use a superscript, i.e., $May^l, Must^l$ for $l \in \{1, 2\}$.

The *May* and *Must* information of the L1 cache can be used to determine the *cache access classification* (CAC) for the second level cache. The CAC indicates whether the access will reach the shared cache *always* (A), *never* (N), or whether the behavior is *uncertain* (U) (Hardy and Puaut 2008). This is represented by the function: $\text{CAC} : Acc_\tau \rightarrow \{A, N, U\}$. Assuming the access is always performed on the L1 cache, the CAC for the L2 cache is defined as follows:

$$\mathrm{CAC}(a) = \begin{cases} A & \text{if } May^1(cb(a)) = \infty \\ N & \text{else if } Must^1(cb(a)) < \infty \\ U & \text{otherwise} \end{cases}. \tag{1}$$

As noted above, without any additional interference, an access will definitely result in a cache hit, if the *Must* age of the targeted cache block is less than the cache's associativity. For shared caches, the classification process must account for inter-core interference in addition to intra-task interference. An intuitive approach is to consider all potentially interfering cache blocks as actually interfering with the analyzed access. The number of such interfering blocks is called the *cache block conflict number* (CCN). When deriving cache hit classifications, the associativity of the shared cache is effectively reduced by the conflict number. This approach was presented by Hardy et al. (2009) and later refined by Liang et al. (2012) to consider only tasks that may execute in parallel. The pessimism in this approach is apparent as interfering tasks may not access all potentially interfering cache blocks simultaneously, repeatedly, and at any point in time.

Formally, the interference caused by a task $\varphi \in T_{C'}$ is defined as:

$$CCN_\varphi = |\mathcal{B}_\varphi| = |\{cb(a) \mid a \in Acc_\varphi\}|. \tag{2}$$

An access $a \in Acc_\tau$ issued by task τ running on core C to the shared cache is classified as a cache hit if and only if:

$$Must^2(cb(a)) + \sum_{C' \neq C} \sum_{\varphi \in T_{C'}} CCN_\varphi < \mathcal{A}. \tag{3}$$

Instead of classifying accesses to the shared cache individually, Nagar and Srikant (2014, 2016); Nagar (2016) analyzed shared cache interference by computing an upper bound on the additional delay. The actual WCET of a task in a multi-core environment is calculated as the sum of the WCET without interference and the extra execution time due to inter-core interference.

Next, we will provide an overview of the approach, which works by solving the so-called *worst-case interference placement* (WCIP) problem. It is important to note that the approach is only formulated for a single instance of interfering tasks in the existing literature (Nagar and Srikant 2014, 2016; Nagar 2016). We will also introduce an extension of the approach that allows for the analysis of multiple interfering jobs from the same task after explaining the basic approach.

The goal of the approach is to find the worst-case distribution of interfering accesses in the control-flow graph of the analyzed task, resulting in the largest increase in execution time. The number of interfering accesses is determined beforehand from the control-flow graph of the interfering tasks. We call this value the *interference budget*. For a single instance of an interfering task, this interference budget can be determined by solving an ILP that maximizes the number of accesses to the shared cache. This method uses the *implicit path enumeration technique* (IPET) as presented in Li and Malik (1997).

Nagar and Srikant present an ILP that provides a precise and safe solution to the problem and an approximate algorithm, which also provides a safe result. The approximate algorithm is comparable in precision to the ILP solution but requires much lower computational effort (Nagar 2016). In the following, we will focus on the approximate algorithm to solve the WCIP problem.

The algorithm approximates the problem by assuming that all accesses to the shared cache of the analyzed task lie on a singular path. These accesses are then ordered by the cache age of the targeted cache block. Accesses with a high age in the cache are easily evicted and are placed at the front of the queue. The algorithm then greedily removes accesses from the front of the queue. For each removed access, the interference budget is reduced according to the resilience of the targeted cache block, and the interference penalty is increased by one cache miss. The algorithm stops when the budget is depleted or all accesses have been converted to cache misses.

A single interfering access can contribute to the eviction of multiple cache blocks. For this reason, the interference budget is multiplied by the *overlapping factor*. The overlapping factor is the maximum number of accesses that can be affected by a single interfering access. This is the second approximation performed by the algorithm. The overlapping factor can be determined by a data-flow analysis, which keeps track of the maximal number of overlapping cache hit paths. A cache hit path of an access is a path along which the access will experience a cache hit Nagar and Srikant (2014).

The pseudocode for the approximate WCIP algorithm is shown in Algorithm 1. In the algorithm, the total interference penalty I is computed by computing a penalty for each cache set s. The value B_s denotes the interference budget, i.e., the number of interfering accesses, and $CCN(s)$ is the sum of conflicting cache blocks from all interfering tasks. The value $numhits_s^k$ corresponds to the number of cache accesses resulting in a cache hit, where the targeted cache block has a resilience of k. This means that for $numhits_s^k$ accesses, the access may result in a cache miss after at least k interfering accesses. Consequently, the maximal age of the targeted cache block is $\mathcal{A} - k$. The value o_s is the overlapping factor as described above.

Algorithm 1 Compute the approximate WCIP penalty (Nagar and Srikant 2016).

Input: For each cache set s:
interference budget B_s, the cumulative CCN for cache set s: $CCN(s)$,
the number of shared cache accesses $numhits_s^k$ with resilience k,
the overlapping factor o_s.
Output: Worst-case cache interference penalty I

```
1:  I ← 0
2:  for all cache sets s do
3:      B_s ← o_s · B_s
4:      for k ← 1, ..., A do
5:          if CCN(s) ≥ k then
6:              if B_s ≤ k · numhits_s^k then
7:                  I ← I + (⌈B_s / k⌉ · L2_miss)
8:                  B_s ← 0
9:              else
10:                 I ← I + (numhits_s^k · L2_miss)
11:                 B_s ← B_s − (numhits_s^k · k)
12:             end if
13:         end if
14:     end for
15: end for
```

The evaluation in Nagar (2016) reports WCET improvements for 9 out of 27 tasks from the MRTC benchmark suite (Gustafsson et al. 2010). In the evaluation, interference is generated by a single instance of the task *nsichneu*.

Note that this algorithm assumes that the number of interfering accesses can be determined a priori. This value is then used to compute the final WCET estimate of a task. However, when tasks are executed repeatedly, a single interfering task φ may be executed multiple times during the execution of the analyzed task τ. Thus, the interference budget actually depends on the maximal execution time.

Consider the following example for clarification. Let $\text{WCET}_0(\tau)$ be the time required to execute τ without any inter-core interference and let the interfering task φ be activated periodically with a minimum inter-arrival time of $Period(\varphi)$ cycles. φ may be executed up to $\lceil \frac{\text{WCET}_0(\tau)}{Period(\varphi)} \rceil + 1$ times in a time frame of $\text{WCET}_0(\tau)$ cycles.

To get the total interference budget from φ, the execution count of φ has to be multiplied by the number of interfering accesses per execution of φ. The result can then be used to compute the interference penalty I_0 using the WCIP approach.

Let $\text{WCET}_1(\tau) = \text{WCET}_0(\tau) + I_0$. Due to the increased execution time, the interfering task φ may be activated more often. I.e., it is possible that the following holds:

$$\lceil \frac{\text{WCET}_0(\tau)}{Period(\varphi)} \rceil < \lceil \frac{\text{WCET}_1(\tau)}{Period(\varphi)} \rceil. \tag{4}$$

Consequently, the interference budget increases due to the higher execution time, and it is necessary to repeat the computation of the penalty value of the WCIP approach using an increased interference budget. For this reason, we make the following observation:

Observation 1 When using the WCIP approach to compute the WCET of a task, the WCIP problem has to be solved iteratively, as the number of interfering accesses is not known a priori. The iteration can be stopped when the resulting WCET value has converged.

Without this iterative application, the WCIP approach is only applicable to systems where each task is executed at most once. To our knowledge, this dependence between repeated task executions and the WCIP approach has not been discussed previously. We compare the approach presented in this paper to the iterative WCIP approach in Sect. 7.

We will now discuss another approach to analyze conflicts in shared caches. Zhang et al. (2022) present a different quantitative analysis approach. In their approach, infeasible cache conflicts are eliminated from the interference computation by considering the order in which conflicts have to occur.

This is done by transforming the control-flow graph of the analyzed task and the interfering task into an *unordered region* (UR) control-flow graph. An unordered region is defined to be a cache access that is not contained in a loop, or sets of cache accesses contained in an out-most loop. The unordered regions are the nodes of the UR-CFG. The key property of the UR-CFG is that it is loop free. Thus, when the control of the program has advanced to another UR, it may never return to a previous UR. All possible paths through the UR-CFG are enumerated and are named *constant execution order path*s (CEOP). Due to its reliance on the constant execution order paths, we refer to this method as the CEOP analysis approach.

It is now possible to detect infeasible conflicts in the shared cache as the conflict pairs between unordered regions can be mutually exclusive. Intuitively, when an access at the start of τ experiences interference in the shared cache from an access near the end of φ, an access performed later by τ will never conflict with an access performed by φ at the beginning. Consider two tasks τ and φ consisting of two unordered regions (UR_1^τ, UR_2^τ) and $(UR_1^\varphi, UR_2^\varphi)$, respectively. When an access contained in UR_1^τ experiences interference from an access contained in UR_2^φ, the accesses contained in UR_2^τ will not conflict with accesses contained in UR_1^φ. Note that this obviously only holds for a single instance of the interfering task φ.

The approach then computes the maximal number of shared cache accesses that are subject to inter-core interference. The interference penalty is determined by multiplying the number of affected accesses with the L2 cache miss penalty. To model multiple instances of an interfering task φ, the penalty is multiplied by the number of jobs from φ that may execute in parallel to the analyzed task τ.

The evaluation performed in Zhang et al. (2022) compares the analysis precision of the CEOP analysis to the CCN approach for a single interfering task instance. The evaluation is performed using the MRTC benchmark suite (Gustafsson et al. 2010). In total 169 different task pairings are evaluated. For 63 (37.3%) task pairings no WCET improvement is achieved. For a total of 92 (54.4%) task pairs the improvement is $\leq 1\%$. On average, the WCET is reduced by 13%.

The number of interfering instances is assumed to be known in Zhang et al. (2022). This means that, like noted for the WCIP approach in Observation 1, an iterative computation is necessary to compute the final WCET value. Again, to the best of our knowledge, the iterative nature of the approach for repeated task executions has not been discussed before.

We note that while the interference computed by the CEOP approach, as described above, scales linearly with the number of conflicting jobs, the WCIP approach scales sub-linearly. I.e., in the WCIP method, the penalty occurring due to two instances of an interfering task is less or equal to two times the penalty of a single instance. This follows directly from the construction of the WCIP problem. The number of interferences required to trigger an additional cache miss increases monotonically until all shared cache accesses are considered cache misses. We capture this fact in the following observation:

Observation 2 The WCIP approach scales better than the CEOP approach, when the number of interfering cores / tasks / jobs increases.

The respective evaluations of the WCIP approach and the CEOP approach assume a single interfering job. The improvement over the baseline CCN analysis will be smaller when considering repeated task activations. As the median WCET improvement for the MRTC benchmark suite is 0% and 1%, for WCIP and CEOP respectively, it is expected that these quantitative approaches perform similar to the CCN analysis for the periodic task model considered in this paper. We compare the precision of the WCIP approach to the presented analysis, which solves the harder problem of classifying individual accesses, in Sect. 7.

4 Analysis overview

We will now give an overview of the analysis presented in this paper. The aim of the analysis is to classify each access to the shared cache as either a cache hit or a potential cache miss. This means that the presented analysis is a qualitative analysis, unlike the WCIP and CEOP approaches, which are quantitative analyses and only provide an upper bound on the total delay due to cache sharing.

The key idea of the proposed analysis is to analyze the timing of accesses to the shared cache. At the start of the analysis, we determine how quickly an interfering task can issue conflicting accesses to the shared cache. Then, we determine for each access from the analyzed task, how long ago the requested data was inserted into the shared cache. Using this information, it is possible to determine whether interfering

tasks running on other cores could have evicted the accessed data prior to the analyzed access. The analysis approach is divided into the following five steps:

1. Initial best-case and worst-case execution time analysis for each task.
2. Cache interference analysis from individual tasks using event-arrival curves.
3. Bounding interference for non-preemptively scheduled task sets.
4. Data-flow analysis to determine cache hit classifications.
5. Final WCET analysis using the new hit classifications.

We will now give a detailed description of the five analysis steps.

Step 1**:** To perform the shared cache analysis, we require both worst-case and best-case timing information on a basic block level. Thus, as the first step, an isolated timing analysis is conducted for each task. To compute WCET estimates, accesses to the shared cache are considered to be cache misses. The best-case execution time (BCET) is computed using a classical age-based abstract domain as presented by Ferdinand and Wilhelm (1999), without considering the impact of inter-core interference. The BCET analysis assumes that tasks are not sharing code across cores. Otherwise, a task running on another core could effectively prefetch data into the shared cache, while the analyzed task is performing some other action, such as performing computations using data stored in the private L1 cache. This would invalidate the BCET estimate. In order to allow for sharing code across cores, the BCET analysis could assume that every access to shared code in the shared cache results in a cache hit.

Step 2**:** Following the initial timing analysis, the event-arrival curves of cache access events originating from interfering tasks $\varphi \in T_{C'}$ are derived. More precisely, it is determined how much time has to pass for φ to access a particular number of interfering cache blocks by solving an ILP model. The ILP model contains a parameter to specify the required level of interference. As the replacement algorithm for each cache set operates independently, the ILP model is solved repeatedly for all cache sets and interference levels from 1 to the cache associativity. The solutions of the ILP for the different parameters give rise to the interference curve of φ. During this step, the BCET information is used to arrive at a safe upper bound on the interfering traffic on the shared cache. This step is discussed in detail in Sect. 5.2.

Step 3**:** The previous step yields interference information for all interfering tasks $\varphi \in T_{C'}$. However, when multiple tasks are assigned to execute on a core C', these tasks can collaborate to evict data from $\tau \in T_C$. In the third step, we extend the concept of interference curves from individual tasks to a set of tasks running on a single core C'.

We analyze the impact of non-preemptively scheduling tasks on an interfering core C' by evaluating potential sequences of task executions on C' and performing a convolution of the associated event-arrival curves in the max-plus algebra. This allows us to compute a safe upper bound on the total interference created by an interfering core executing multiple different tasks in a non-preemptive fashion. This step is explained in Sect. 5.3. The total inter-core interference experienced by a task $\tau \in T_C$ is then given as the sum of the interference caused by other cores $C' \in \mathcal{C} \backslash \{C\}$.

Step 4**:** Based on the information from the previous analysis steps, a backward *data-flow analysis* (DFA) is performed on the control-flow graph of the analyzed task (Aho et al. 2007, p. 597ff). The DFA investigates all accesses which potentially result in a cache hit. An access is considered as a *potential-hit*, if it results in a cache hit without any inter-core interference. For each *potential-hit* there exist corresponding cache accesses which initially caused the relevant cache block to be loaded into the shared cache. The DFA determines the maximal duration between these pairs of cache accesses, which load the block into the shared cache and subsequently access it again. Additionally, it also keeps track of any intra-task interference. By evaluating the interference curves of interfering cores for the maximal load-access path duration, the inter-core interference can be safely bounded. This means that potential-hits can be classified as either definite cache hits or potential cache misses. The foundations for hit classifications based on timing information are presented in Sect. 6.1 and the data-flow analysis is presented in Sect. 6.2.

Step 5**:** We use the access classifications computed in the previous step in a WCET analysis to determine the final WCET for each task. The results achieved in this WCET analysis are evaluated in Sect. 7.

5 Interference curves for shared cache interference

This section deals with the first key component of the timing-aware cache interference analysis presented in this paper. In Sect. 5.1, we introduce the concept of interference curves and motivate their use in the analysis of shared cache interference. The following Sect. 5.2 corresponds to the second step of the analysis as described in the previous section. In it, we show how interference curves can be derived from the control-flow graph of a task using best-case execution time information. Finally, in Sect. 5.3, the third step of the analysis is discussed. The interference curves of multiple tasks are combined to compute the total interference resulting from a non-preemptively scheduled set of tasks.

5.1 Definition and motivation of interference curves

We formally introduce interference curves and motivate why it is beneficial to analyze shared cache interference from this perspective in this section.

Traditionally, shared cache interference has been quantified by counting the number of cache blocks that an interfering task may load into the shared cache. The

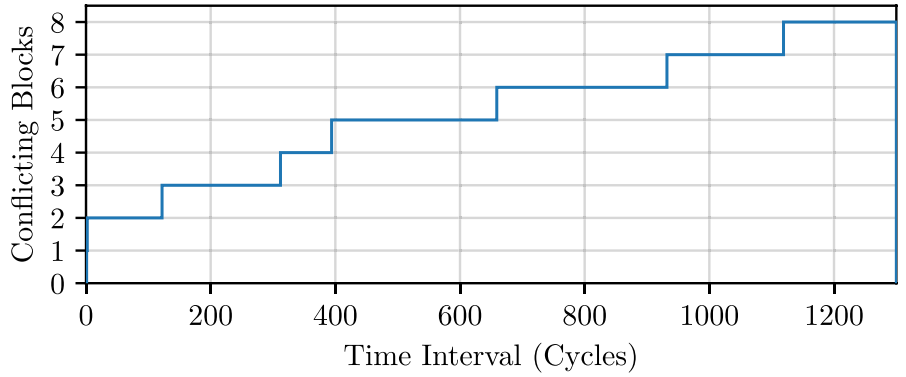


Fig. 1 Example of an event-arrival curve expressing shared cache interference. Derived from the `aiifft01` benchmark of the EEMBC AutoBench suite

pessimism of this approach becomes apparent when the time that passes between accesses to the conflicting blocks is considered.

Under the LRU replacement policy, cache blocks are ordered using the least-recently-used property. Consequently, the age of a cache block $b \in \mathcal{B}_\tau$ corresponds to the number of different cache blocks, which were stored in the cache since the last access to b. It follows that multiple accesses to the same conflicting cache block do not cause further aging of b in the cache, as the set of conflicting blocks does not change.

Using this observation, we define the notion of an *aging-event for a cache-block* in the context of shared cache interference as an access by an interfering task, which causes the age of an analyzed cache block to increase. Consequently, multiple events correspond to multiple accesses targeting a set of distinct cache blocks.

Definition 1 (Interference curve) An interference curve η_φ is a function, which maps a time frame of Δt cycles to the maximal number of cache blocks accessed by $\varphi \in T_C$: $\eta_\varphi : \mathbb{N}_0 \to \mathbb{N}_0$, where $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$.

A mapping $\eta_\varphi(\Delta t) = n$ means that there exists a scenario in which φ will access n different cache blocks during a time frame of Δt cycles. The statement $\eta_\varphi(\Delta t) = n$ is thus equivalent to the fact that there exists an initial hardware state and path in the CFG of φ that causes accesses to n distinct cache blocks under the constraint that the best-case duration of the path is less or equal to Δt cycles. Note that we refer to the best-case execution time here, as the fastest execution of an interfering task causes the worst-case interference for the analyzed task.

Figure 1 shows an example for such an interference curve. The x-axis shows a time interval, and the y-axis represents how many different cache blocks may be accessed. In this example, the interference grows quickly in the beginning. Two interfering accesses may happen almost instantaneously. After a time frame of 122 cycles, the task may have issued accesses to 3 different cache blocks. However, the time interval required to observe a larger amount of interference is significantly

higher. To observe interference from 8 conflicting blocks takes 1119 cycles. This example curve has been derived from the `aiifft01` benchmark from the EEMBC benchmark suite (Poovey et al. 2009), with a shared cache size of 4 KB using the approach presented in the upcoming subsection.

This clearly demonstrates the pessimism in the currently available analysis techniques (Liang et al. 2012; Nagar 2016; Zhang et al. 2022). All these techniques assume that every conflicting block may be accessed instantaneously by an interfering task. However, we can see here that it takes a substantial amount of time for the interfering task to generate the traffic on the shared cache. Accounting for this delayed onset of interference after storing data in the shared cache is the key idea behind the timing-aware shared cache analysis presented in this paper.

5.2 Deriving interference curves of a task

This section shows how an interference curve, as seen in Fig. 1, can be derived from a task's control-flow graph.

To derive an event arrival curve from the CFG of a task φ, we have to associate cache accesses to the nodes in the CFG (V_φ, E_φ). For data caches, this relation arises from the memory-accessing instructions contained in the program and their respective target addresses. For instruction caches, cache accesses originate from fetching operations in the processor pipeline, so we have to consider the pipeline behavior.

Let $\rightsquigarrow \subset V_\varphi \times \mathcal{B}_\varphi$ be a relation that contains the pair (v, b) iff executing the node $v \in V_\varphi$ may cause an access to the block $b \in \mathcal{B}_\varphi$. A path π is a sequence of nodes $\pi = (v_1, \ldots, v_p)$ with $(v_i, v_{i+1}) \in E_\varphi,\ i \in \{1, \ldots, p-1\}$. The execution of a path π in the CFG of φ causes another task τ to experience n interfering cache accesses:

$$n = \left| \bigcup_{v \in \pi} \left\{ b \in \mathcal{B}_\varphi \mid v \rightsquigarrow b \right\} \right|. \tag{5}$$

Using this relation, we derive the interference curve of a task φ. Note that η_φ is a step function. Furthermore, we are only interested in the arrival of the first $\mathcal{A}$ events, as after $\mathcal{A}$ events all previously cached blocks are evicted from an LRU cache. Thus, we can capture a task's cache access behavior by deriving the minimal time required to access $1,\ 2,\ \ldots,\ \mathcal{A}$ distinct cache blocks. These values correspond to the location of the steps in Fig. 1, where the interference jumps upwards.

To compute the minimal time required for a particular number of events, we utilize an ILP model based on the *implicit path enumeration technique* (Li and Malik 1997). We base the model on the variant presented by Oehlert et al. (2018), which evaluates bus contention in a multi-core system. As the basic construction of the ILP model is out of the scope of this paper, we focus only on the additional constraints required to model cache interference. In the following, we present the variables and constraints introduced into the model to capture inter-core cache interference. The notation is summarized in Table 2.

To decide whether a particular cache block will contribute to the interference, we introduce a binary decision variable for each cache block. This decision

variable indicates whether the cache block is accessed on the considered path. The variables are denoted by t_b for $b \in \mathcal{B}_\varphi$. The indicator variables t_b are constrained by the constraints given in Eq. (6) and Eq. (7), where q_v is the variable containing the execution count of v.

$$0 \leq t_b \leq 1 \tag{6}$$

$$t_b \leq \sum_{v:\ v \nrightarrow b} q_v \tag{7}$$

Thus, if any node v is executed which may access the block b, the indicator variable t_b may take the value 1, otherwise it is set to 0. The number of variables t_b depends on the number of cache blocks potentially accessed by φ. For an instruction cache, this scales linearly with the code size of the analyzed task φ and is inversely proportional to the cache line size.

To determine the time frame during which n cache blocks may be accessed, only paths containing accesses to at least n cache blocks are considered:

$$n \leq \sum_{b \in \mathcal{B}_\varphi} t_b \,. \tag{8}$$

In summary, the cache interference modeling requires an additional binary variable for each cache block (see Eq. 6), a corresponding constraint (see Eq. 7), and a constraint enforcing the required level of interference (see Eq. 8). The size of the ILP model is thus similar to the size of the underlying IPET formulation (Oehlert et al. 2018), with the additional constraints growing linearly in the code size of the analyzed task.

In the model, n is a parameter which may be set to values in $\{1, \ldots, \mathcal{A}\}$ to determine the minimal cycle count required for different levels of interference.

Consequently, the objective of the ILP is to minimize the number of cycles required to cause accesses to n distinct cache blocks, as shown in Eq. (9). The path duration is computed as the sum of the execution time contributions z_v of each node $v \in V_\varphi$:

$$\text{minimize} \sum_{v \in V_\varphi} z_v \,. \tag{9}$$

Table 2 ILP variables

Symbol	Meaning
$q_v \in \mathbb{N}$	Execution count of a node $v \in V_\varphi$
$z_v \in \mathbb{N}$	Execution time contribution from $v \in V_\varphi$
$t_b \in \{0,1\}$	Indicates whether a conflicting block $b \in \mathcal{B}$ is accessed
$s_v \in \{0,1\}$	Indicates whether $v \in V_\varphi$ is the starting node of the path
$e_v \in \{0,1\}$	Indicates whether $v \in V_\varphi$ is the final node of the path

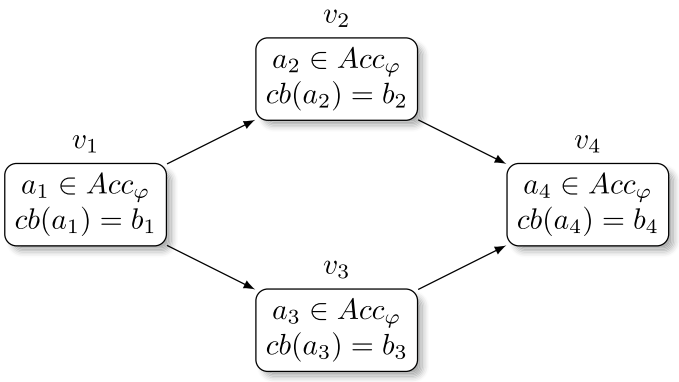


Fig. 2 Example CFG containing four nodes $v_1, v_2, v_3, v_4 \in V_\varphi$. Nodes contain accesses $a_1, a_2, a_3, a_4 \in Acc_\varphi$, and target cache blocks $b_1, b_2, b_3, b_4 \in \mathcal{B}_\varphi$, respectively. The accesses a_2 and a_3 are mutually exclusive. The maximal interference caused by this CFG is 3

The execution time contribution z_v depends on the best-case execution time and the execution count q_v of the node v. As we do not make assumptions about the distribution of events inside the nodes, the time contributions of the first and last nodes of the (partial) path are reduced to a single cycle.

To clarify this, consider a simple linear CFG with three nodes, v_1, v_2, v_3, as an example. Assume that each node in the graph contains an access to a single cache block and has a BCET of 100 cycles. When we want to answer the question of how quickly contents in the shared cache may age three times, only the complete path (v_1, v_2, v_3) is a viable candidate. The other partial paths in the CFG generate two or fewer events. The BCET of the complete path is 300 cycles. However, the access to the shared cache in node v_3 might not happen at the very end of processing v_3, but near the beginning of v_3. Thus, in a real execution, three events may happen already after, for example, 210 cycles. To generate a safe lower bound, we assume that accesses associated to a node will happen at the very beginning (end) of the final (first) node in the considered path. Hence, the first and last node are considered to contribute only a single cycle to the execution time. A safe lower bound on the time to observe three events for this example would thus be 102 cycles. This approximation is needed to ensure the safety of the result but also introduces some pessimism into the analysis. To combat this pessimism, the analysis can be performed at a finer scale by virtually splitting large basic blocks in the CFG into multiple smaller nodes.

Note that by determining interference on the basis of paths in the CFG, we implicitly eliminate mutually exclusive accesses from the interference computation. For example, consider the control-flow graph shown in Fig. 2. The nodes $v_1, v_2, v_3, v_4 \in V_\varphi$ each contain an access targeting a different cache block. The maximal interference caused by this CFG is 3 blocks: either the blocks $\{b_1, b_2, b_4\}$ or $\{b_1, b_3, b_4\}$ are accessed. The blocks b_2 and b_3 are never accessed together on the same path. Such mutually exclusive access behavior is therefore safely excluded from the event-arrival curves, leading to a tight estimation of the possible interference. In contrast, the standard CCN interference estimation technique simply counts

the number of potentially accessed blocks and would thus conclude a maximal interference of 4.

To construct the complete interference curve for a task $\varphi \in T_{C'}$, the presented ILP is solved for all parameter values $1 \leq n \leq \mathcal{A}$. Let $\Delta t_1, \ldots, \Delta t_{\mathcal{A}}$ denote the solution of the ILP for a task φ for interference levels $n = 1, \ldots, \mathcal{A}$, respectively. If the task never causes n or more interfering accesses during any execution, the ILP will be infeasible for that parameter. In that case we set the value $\Delta t_n = \infty$. Using these results, the interference curve for a task φ is defined as follows:

$$\eta_\varphi(\Delta t) = \begin{cases} 1 & \text{if } \Delta t_1 \leq \Delta t < \Delta t_2, \\ 2 & \text{if } \Delta t_2 \leq \Delta t < \Delta t_3, \\ & \vdots \\ \mathcal{A} & \text{if } \Delta t_{\mathcal{A}} \leq \Delta t, \\ 0 & \text{otherwise.} \end{cases} \tag{10}$$

As noted before, the interference on different cache sets is analyzed independently of each other. Interference curves are thus derived seperately for each cache set. Assuming that a single task is assigned to each core, we can now give a safe upper bound on the total interference using the computed curves.

Theorem 1 *The interference $\gamma_\tau(\Delta t)$ at the shared cache over a time frame of Δt cycles, experienced by the task $\tau \in T_C$ from other cores $C' \neq C$, given that all cores C' only process a single task is bounded by the addition of the curves η_φ from co-running tasks $\varphi \in T_{C'}, \forall C' \in \mathcal{C} \backslash \{C\}$:*

$$\gamma_\tau(\Delta t) = \sum_{\varphi:\ T_{C'} = \{\varphi\},\ C' \neq C} \eta_\varphi(\Delta t). \tag{11}$$

Proof We prove the theorem by contradiction. Assume that there exists a task that is subject to n' interfering accesses at the shared cache over a time frame of Δt cycles with $n' > \gamma_\tau(\Delta t)$. Due to the least-recently-used replacement property, interfering accesses issued by two different cores C'_1 and C'_2 can not collaborate to create larger interference than the sum of both contributions. It follows that there must be a core $C' \neq C$ with $T_{C'} = \{\varphi\}$, that issued more than $n_\varphi = \eta_\varphi(\Delta t)$ interfering accesses. However, the solution computed for the interference parameter n_φ by the ILP gives the smallest duration of any path from φ causing at least n_φ interfering accesses. We conclude that no path causing more than n_φ interference with duration $\leq \Delta t$ exists in φ's CFG and have proven the theorem by contradiction. □

Using the cumulative interference function γ_τ, the time-to-live of a cache block $b \in \mathcal{B}_\tau$ can be expressed as a function of its *Must* age.

Definition 2 (Time-to-live) The time-to-live (TTL) of a cache block $b \in \mathcal{B}_\tau$ in the shared cache is given by the function $\text{TTL}_\tau : \{0, \ldots, \mathcal{A} - 1\} \to \mathbb{N}_0 \cup \{\infty\}$:

$$\mathrm{TTL}_{\tau}(age) = \sup_{0 \leq \Delta t} \{\Delta t \mid \gamma_{\tau}(\Delta t) < \mathcal{A} - age\} \tag{12}$$

The value *age* refers to the maximal number of conflicting blocks that have been accessed by τ since the last access to b, without considering the inter-core interference, i.e., the *Must* age of b.

Note that the TTL of a block may be $+\infty$ cycles in case the interfering tasks never create sufficient interference to trigger the eviction.

5.3 Analyzing non-preemptive scheduling

The approach described in the previous section computes an interference curve η_{φ} based on the control-flow graph of a singular task $\varphi \in T_{C'}$. The total interference γ_{τ} experienced by the analyzed task $\tau \in T_C$ at the shared cache is then computed as the sum of the interference curves of all tasks $\varphi \in T_{C'}$ running on interfering cores C', as formulated in Eq. (11). This method is only applicable to systems where a single task causes interference per core. When multiple different tasks are executed on the same core, these tasks can collaborate to evict data from the shared cache.

The interference curve of one particular task may be vastly different from the interference curve of another task. To safely classify cache accesses to a shared cache in a such a system, we must determine the worst-case interference caused by the set of tasks $T_{C'}$ running on each interfering core $C' \neq C$.

In this section, we compute the interference curve resulting from multiple tasks executing on a single core under non-preemptive scheduling. This constitutes the third step of the presented analysis.

Under non-preemptive scheduling, once a task has started to execute, it will not be interrupted until it has finished its computations. This contrasts preemptive scheduling, where the scheduler may decide to interrupt the currently running task in favor of a different task. An application of interference curves to preemptive scheduling is presented in Fischer and Falk (2024).

We do not assume any precedence constraints between different tasks. I.e., the order in which tasks execute on an interfering core C' is not restricted. Consequently, a task running on core C can experience interference from all other tasks $\varphi \in \bigcup_{C' \neq C} T_{C'}$. Furthermore, we do not require a minimal inter-arrival time between two instances of the same task. This means that the interference bounds derived in this section are general and apply to all non-preemptive schedules.

This is another differentiating factor to the existing WCIP and CEOP analysis approaches by Nagar (2016) and Zhang et al. (2022). The WCIP and CEOP approaches inherently operate on the assumption that the number of interfering jobs is known. Thus, the final WCET value has to be computed in an iterative fashion using the periods of the interfering tasks (see Observation 1). The approach presented in this section makes no such assumption and the resulting interference curve is valid for arbitrary task activations patterns.

The rest of this section is structured as follows: Sect. 5.3.1 provides a brief introduction to the calculation of event-arrival curves in the max-plus algebra. We formally describe the interference computation for non-preemptive scheduling in Sect. 5.3.2 and construct an algorithm to compute the resulting interference curve in Sect. 5.3.3.

5.3.1 Operations in max-plus algebra

When multiple tasks execute on the same core, they all contribute to the interference at the shared cache. To compute the total interference curve for a set of tasks, the interference curves of the individual tasks running on the core have to be combined. This calculation is performed using the max-plus algebra.

The max-plus algebra, and its counterpart the min-plus algebra, have been used in the area of *network calculus* (Le Boudec and Thiran 2001) to analyze the traffic in a network, and in *real-time calculus* to determine the schedulability of a system (Thiele et al. 2000).

In the max-plus algebra, the operators of addition and multiplication from the standard algebra are replaced by the maximum and addition operators, respectively. A crucial operator for the calculation of interference curves is the max-plus convolution. The max-plus convolution of two interference curves η_{φ_1} and η_{φ_2} from tasks $\varphi_1, \varphi_2 \in T_{C'}$ is represented using the $\circledast$ operator:

$$(\eta_{\varphi_1} \circledast \eta_{\varphi_2})(\Delta t) = \max_{0 \leq \delta \leq \Delta t} \{\eta_{\varphi_1}(\delta) + \eta_{\varphi_2}(\Delta t - \delta)\}. \tag{13}$$

The max-plus convolution determines the maximal interference caused by two tasks over a total duration of Δt cycles. This calculation considers all possible distributions of the Δt cycles between the two tasks, ensuring that the worst-case interference is identified by taking the maximum interference value across all time distributions over η_{φ_1} and η_{φ_2}.

An interference curve $\eta_{\varphi}(\Delta t)$ addresses the question of how many cache blocks may cause interference after Δt cycles. However, it can be useful to view cache interference from an alternative perspective: how much time must elapse to potentially experience n or more interference events? This perspective can be taken using the pseudo-inverse of an event-arrival curve. The pseudo-inverse of an event-arrival curve is defined as:

$$\eta^{-1}(n) = \inf\{\Delta t \geq 0 \mid \eta(\Delta t) \geq n\}. \tag{14}$$

Thus, the pseudo-inverse $\eta^{-1}(n)$ gives the minimal duration over which n interfering accesses may occur and satisfies the following properties (Le Boudec and Thiran 2001):

$$\eta(\Delta t) \geq n \implies \eta^{-1}(n) \leq \Delta t, \tag{15}$$

$$\eta^{-1}(n) < \Delta t \implies \eta(\Delta t) \geq n. \tag{16}$$

Furthermore, in max-plus algebra, $-\infty$ serves as the absorbing element, meaning that $\forall n \in \mathbb{N}_0 : -\infty + n = -\infty$. We use the value $-\infty$ to signal infeasible situations. Such situations arise when computing the interference created by a full execution of a task φ, but the allotted time frame is insufficient to fully execute φ according to its best-case execution time.

To eliminate infeasible situations and thus increase the precision of the final result, we introduce a window function W_t. The window function $W_t : \mathbb{N} \to \{0, -\infty\}$ maps to either 0 or $-\infty$, depending on the parameter t and the input Δt:

$$W_t(\Delta t) = \begin{cases} 0 & \text{if } \Delta t \geq t, \\ -\infty & \text{else.} \end{cases} \tag{17}$$

For example, consider the calculation: $((\eta_{\varphi_1} \circledast \eta_{\varphi_2}) + W_{100})(\Delta t)$. A window W_{100} with parameter 100 is added to the convolution of η_{φ_1} and η_{φ_2}. The resulting curve thus yields $-\infty$ for $\Delta t < 100$. By applying a window to a curve, it can be marked as infeasible for short durations. In the context of interference curves, applying W_t refines the interference computation by enforcing a minimal duration for the scenario associated to that curve.

We will use these concepts in the next section to compute a safe upper bound on the interference stemming from a core executing a set of tasks using non-preemptive scheduling.

5.3.2 Modelling non-preemptive scheduling

To apply timing-aware interference analysis to non-preemptive scheduling, we compute an interference curve that captures the worst-case interference created by multiple tasks assigned to a single interfering core. This encompasses, for example, scenarios where a task $\varphi_1 \in T_{C'}$ ends its execution and another task $\varphi_2 \in T_{C'}$ starts execution immediately afterward. Consequently, between two subsequent accesses to a cache block from task $\tau \in T_C$, multiple tasks can contribute interference for the cache block and trigger its eviction. The goal of this section is to capture all possible interference scenarios and determine the worst-case interference caused by any of these scenarios. To this end, we define the different scenarios that may occur on an interfering core C'.

Definition 3 (Execution Scenario) An execution scenario for an interfering core C' is a sequence of tasks $(\varphi_1, \ldots, \varphi_j) \in (T_{C'})^j$. Let $\mathcal{S}_{C'} = (T_{C'})^{\mathbb{N}} = \bigcup_{j \in \mathbb{N}} (T_{C'})^j$ be the set containing all execution scenarios.

The tasks in an execution scenario $(\varphi_1, \ldots, \varphi_j) \in \mathcal{S}_{C'}$ are executed sequentially in ascending order by the interfering core C'. We are interested in the behavior of C' during a cache block reuse windows, which is the time frame from the point of storing information in the shared cache up to the point where another access to that content performed. This means an execution scenario does not represent a complete execution of the system from startup to shutdown, but models the behavior of the

interfering core between accesses to blocks stored in the shared cache. Thus, for an execution scenario $(\varphi_1, \ldots, \varphi_j) \in \mathcal{S}_{C'}$, a cache block is stored in the shared cache by the analyzed task τ while the interfering core executes task φ_1. The access to be classified as either a cache hit or potential miss occurs while the interfering core executes φ_j.

To issue cache hit classifications that hold even in the worst case, we must compute the worst-case interference curve over all execution scenarios $\mathcal{S}_{C'}$. Let F be a function that maps an execution scenario of an interfering core C' to the associated interference curve:

$$F_{C'} : \mathcal{S}_{C'} \to (\mathbb{N}_0 \to (\mathbb{N}_0 \cup \{-\infty\})). \tag{18}$$

Note that the image of $F_{C'}$ contains functions mapping to $-\infty$, as a particular execution scenario may be infeasible for short time durations. A scenario containing a complete execution of a task $\varphi \in T_{C'}$ may not occur in a time frame that is shorter than the BCET of φ. By mapping the interference value for $\Delta t < \text{BCET}(\varphi)$ to $-\infty$, the scenario will not contribute to the maximal interference for $\Delta t < \text{BCET}(\varphi)$.

The worst-case interference created by multiple tasks on a non-preemptive scheduled core C' is given by:

$$\mu_{C'}(\Delta t) = \max_{s \in \mathcal{S}_{C'}} \{F_{C'}(s)(\Delta t)\}. \tag{19}$$

In the remainder of this section, we develop the precise definition of $F_{C'}$ to complete the formal model of shared cache interference under non-preemptive scheduling.

The contribution of a task φ to the total interference depends on the analyzed time frame Δt during which interference can occur. Additionally, it depends on the

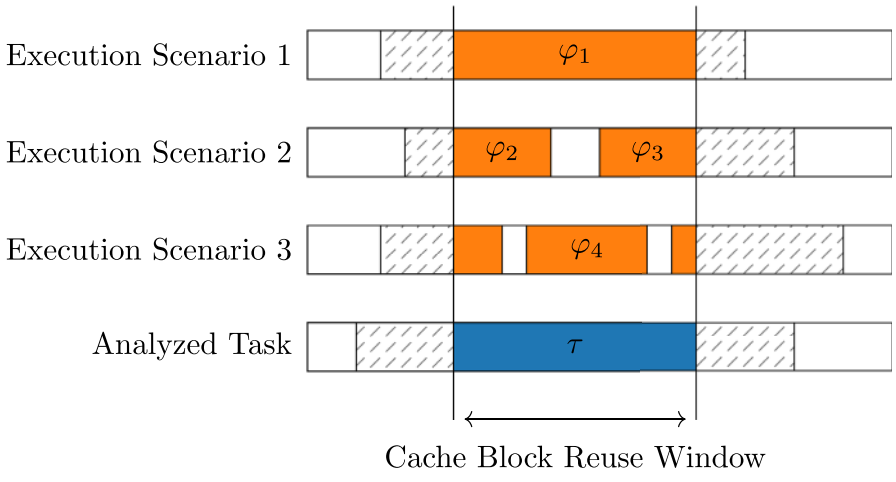


Fig. 3 Three different execution scenarios for an interfering core C'. The cache block reuse window of the analyzed task, running on core C, is marked in blue. The orange sections of the co-running tasks contribute to the shared cache interference

fact whether the co-running task φ starts and/or ends its execution within the analyzed reuse window.

Multiple different execution scenarios interfering with the reuse of a cache block are shown in Fig. 3. The time frame between the initial access to the cache block and its reuse by the analyzed task τ is shown in blue and labeled as the *Cache Block Reuse Window*. Three different execution scenarios are shown for the interfering core. The orange sections in the different scenarios indicate the source of the inter-core cache interference from co-running tasks on the core C'. The co-running tasks creating interference in the three scenarios are labeled as φ_1, φ_2, φ_3, and φ_4, respectively.

We differentiate between four different categories of interfering tasks, named as follows:

1. `single-task`: A task already executing at the start of the reuse window that does not finish executing during the reuse window (φ_1 from Fig. 3).
2. `in-task`: A task already executing at the start of the reuse window that finishes executing during the reuse window (φ_2 from Fig. 3).
3. `out-task`: A task that starts during the reuse windows and does not finish during the reuse window (φ_3 from Fig. 3).
4. `body-task`: A task that starts and finishes during the reuse window (φ_4 from Fig. 3).

We will use these categories to quantify how quickly a task may create interference. Differentiating between these different categories is useful as we may place additional constraints on the paths from which the interference curves are derived, allowing us to create a tighter estimate on the total interference. The interference curve of a task φ in one of the four categories is denoted using a superscript, i.e., η_φ^{single}, η_φ^{in}, η_φ^{out}, and η_φ^{body}.

The curve η_φ^{single} for `single-task` corresponds to the unconstrained curve as described in Sect. 5.2. An `in-task` already executes at the start of the reuse window and will finish executing during the reuse window. Therefore, the curve for an `in-task` of φ is computed considering only paths ending in a sink of φ's CFG. This is beneficial because by the end of a task's execution, the private L1 cache has warmed up, reducing the likelihood of L1 cache misses, which cause shared cache interference, compared to the task's initial execution phase. Similarly, an `out-task` begins executing during the reuse window. Thus, the η_φ^{out} curve of φ is computed considering only paths starting at the initial node of the CFG of φ. This consideration is useful as set-associative caches are divided into multiple cache sets. For example, to interfere with the third cache set, a task may first need to execute code mapped to the first and second cache set, Hence, the accesses to the third set are delayed and the interference behavior relative to the unconstrained curve η_φ^{single} improves, leading to better cache hit classification performance.

Formally, we observe that by construction $\eta_\varphi^{in}(\Delta t) \leq \eta_\varphi^{single}(\Delta t)$, and $\eta_\varphi^{out}(\Delta t) \leq \eta_\varphi^{single}(\Delta t)$ for all $\Delta t \in \mathbb{N}$. When determining the curve of a *body* task, it is guaranteed that the task will fully execute from start to

finish. It is thus infeasible for the curve $\eta_{\varphi}^{body}(\Delta t)$ to contribute to interference for time frames $\Delta t < \text{BCET}(\varphi)$.

Incorporating these constraints imposed by the four categories improves accuracy of the interference curves. Consequently, the interference bound can be tightened by differentiating between the stages a task is in while contributing to the shared cache interference.

Next, we integrate these constraints into the curve derivation ILP presented in Sect. 5. As before, we adopt the notation of Oehlert et al. (2018) to formulate the ILP constraints: The binary ILP decision variable $s_v \in \{0, 1\}$ indicates whether a node $v \in V_{\varphi}$ in the CFG is used as the starting point in the ILP solution. Similarly, the binary ILP variable $e_v \in \{0, 1\}$ signifies whether the considered path ends at v.

Let $v^0 \in V_{\varphi}$ be the initial node in the CFG of φ, whereas $V_{\varphi}^{end} \subset V_{\varphi}$ contains all sinks of the CFG. v^0 corresponds to the initial node of the entry function, e.g. the `int main()` function, and nodes $v^{end} \in V_{\varphi}^{end}$ correspond to `returns` from the entry function. Adding the constraints shown in Eq. (20a) and Eq. (20b) into the ILP formulation from Sect. 5 allows us to compute interference curves for `out-tasks` and `in-tasks`, respectively.

$$s_{v^0} = 1 \tag{20a}$$

$$\sum_{v^{end} \in V_{\tau}^{end}} e_{v^{end}} = 1 \tag{20b}$$

It is not necessary to perform a dedicated curve derivation for the `body-task` curves as they can be derived from the unconstrained curves, which are used to compute `single-task` interference. As a `body-task` φ needs to execute for at least $\text{BCET}(\varphi)$ cycles, η_{φ}^{body} is induced by η_{φ}^{single} using an additional window:

$$\eta_{\varphi}^{body}(\Delta t) = \eta_{\varphi}^{single}(\Delta t) + W_{\text{BCET}(\varphi)}(\Delta t). \tag{21}$$

We apply a windowing function $W_{\text{BCET}(\varphi)}$, which ensures that a scenario containing a `body-task` of φ impacts the final interference computation only for time durations $\Delta t \geq \text{BCET}(\varphi)$, because such a scenario is infeasible for shorter durations.

Using the interference curves of each task in the four categories, we can calculate the total interference created by a core C'. The scenarios in $\mathcal{S}_{C'}$ can be partitioned into two types: scenarios that consist only of a `single-task`, i.e., $s = (\varphi) \in \mathcal{S}_{C'}$, and scenarios that contain an `in-task`, `out-task` and zero or more `body-tasks`, i.e., $s = (\varphi_1, \ldots, \varphi_j) \in \mathcal{S}_{C'}$. We thus differentiate the definition of $F_{C'}$ based on the size of the execution scenario:

$$F_{C'}(\varphi_1, ..., \varphi_j) = \begin{cases} \eta_{\varphi_1}^{\text{single}} & \text{if } j = 1, \\ \eta_{\varphi_1}^{in} \circledast \left(\circledast_{1<i<j} \eta_{\varphi_i}^{body} \right) \circledast \eta_{\varphi_j}^{out} & \text{otherwise.} \end{cases} \tag{22}$$

When a scenario $s \in \mathcal{S}_{C'}$ contains only a single task φ, i.e., $s = (\varphi)$, its interference curve is equal to the interference curve of that task. For scenarios that contain more than a single task, we compute the worst-case interference as the max-plus convolution of the curves from the `in-task`, `body-tasks`, and the `out-task`. The resulting curve requires at least $\sum_{1<i<j} \text{BCET}(\varphi_i)$ cycles to be feasible. For shorter durations, $F_{C'}(\varphi_1, \ldots, \varphi_j)$ yields $-\infty$.

Note, in an actual execution of the system a scenario may occur where no `in-task` or `out-task` exists. This might happen if the interfering core remains idle for some time during the cache block reuse window. However, as we are interested in the worst-case interference, it is safe to disregard such situations in this computation, as they are subsumed in the considered scenarios. I.e., an idle core will not create worse interference conditions than an active core: formally $\forall \varphi \in T_{C'} : 0 \leq \eta^{in}_{\varphi} \wedge 0 \leq \eta^{out}_{\varphi}$, by construction of the interference curves.

Using the function $F_{C'}$, defined in Eq. (22), we can determine the maximal interference caused by a specific scenario $s \in \mathcal{S}_{C'}$. Moreover, by taking the maximal interference over all possible scenarios, we obtain a safe upper bound on the total interference caused by a non-preemptively scheduled core. This upper bound is given by $\mu_{C'}$, as defined in Eq. (19).

5.3.3 Algorithmic computation of the worst-case interference

The upper bound on the interference $\mu_{C'}(\Delta t)$ depends on all possible execution scenarios. As the tasks in the analyzed system may be activated repeatedly, there are infinitely many different execution scenarios. This makes it intractable to compute interference curves for all scenarios in $\mathcal{S}_{C'}$ and determine the maximum interference from the results. However, we are not interested in the interference generated by any specific sequence of tasks but rather the maximal interference for any sequence of tasks. In this section, we present an efficient algorithm that computes the function $\mu_{C'}$ by directly determining the worst-case interference for any Δt. The key idea of the presented algorithm is to intelligently

rearrange the required computations, leveraging the properties of the max-plus convolution, particularly its associativity and commutativity (Le Boudec and Thiran 2001).

Algorithm 2 Compute the worst-case interference curve μ for a core c'

Input: Task set $T_{C'}$ with interference curves $\eta_\varphi^{single}, \eta_\varphi^{in}, \eta_\varphi^{out}, \eta_\varphi^{body}$ for all $\varphi \in T_{C'}$.
Output: Worst-case interference curve μ for up to $\mathcal{A}$ events.

1: $\mu \leftarrow (\Delta t \mapsto 0)$
2: **for all** $\varphi \in T_{C'}$ **do**
3: $\quad \mu(\Delta t) \leftarrow \max\{\mu(\Delta t),\ \eta_\varphi^{single}(\Delta t)\}$
4: **end for**
5: **for all** $\varphi_1 \in T_{C'}$ **do**
6: $\quad$ **for all** $\varphi_2 \in T_{C'}$ **do**
7: $\quad\quad \mu(\Delta t) \leftarrow \max\{\mu(\Delta t),\ (\eta_{\varphi_1}^{in} \circledast \eta_{\varphi_2}^{out})(\Delta t)\}$
8: $\quad$ **end for**
9: **end for**
10: **repeat**
11: $\quad done \leftarrow$ **True**
12: $\quad$ **for all** $\varphi \in T_{C'}$ **do**
13: $\quad\quad \mu'(\Delta t) \leftarrow (\mu \circledast \eta_\varphi^{body})(\Delta t)$
14: $\quad\quad$ **if** $\exists \delta \in \mathbb{N}, 1 \leq \delta \leq \mu^{-1}(\mathcal{A}) : \mu'(\delta) > \mu(\delta)$ **then**
15: $\quad\quad\quad \mu(\Delta t) \leftarrow \max\{\mu(\Delta t), \mu'(\Delta t)\}$
16: $\quad\quad\quad done \leftarrow$ **False**
17: $\quad\quad$ **end if**
18: $\quad$ **end for**
19: **until** $done =$ **True**

The pseudocode to compute the worst-case interference is shown in Algorithm 2. Line 1, initializes the result to the constant zero function. Lines 2–4 compute the maximal interference stemming from `single-task` scenarios by taking the maximum interference over all curves η_φ^{single} for all $\varphi \in T_{C'}$. Then, in the following lines 5–9, scenarios that contain exactly two tasks are considered. The `in-task` and `out-task` curves are combined using a max-plus convolution. This calculation gives the maximal interference caused by any two tasks executing in sequence.

Additionally, this step also prepares the computation of the interference from scenarios containing more than two tasks. This is the case, as the max-plus convolution is commutative and associative, i.e., $\eta_{\varphi_1}^{in} \circledast \eta_{\varphi_2}^{body} \circledast \eta_{\varphi_3}^{out} = (\eta_{\varphi_1}^{in} \circledast \eta_{\varphi_3}^{out}) \circledast \eta_{\varphi_2}^{body}$. Thus, it is possible to perform the convolutions required by Eq. (22) in any order. We use this property to first compute the contribution by the `in` and `out` tasks before iteratively adding additional `body` tasks to the intermediate result.

The outer loop spanning lines 10–19 iterates until there are no more changes observed on the result. This is realized using the *done* flag. If there was a change made to the curve in the current iteration, the flag is set to **False** (line 16). Otherwise, the algorithm terminates.

The inner loop, from lines 12–18, checks whether adding a `body` task of any φ to the intermediate result μ causes the interference to increase. The modified curve

μ' is computed in line 13 by performing a max plus convolution of μ with the `body` curve of the currently considered task φ.

In line 14, we determine the minimal number of cycles required to generate $\mathcal{A}$ events using the current result μ. The value is computed using the pseudo-inverse of μ: $\mu^{-1}(\mathcal{A})$. This bound is determined as after $\mathcal{A}$ accesses to distinct cache blocks, all data will be evicted from an $\mathcal{A}$-way LRU cache. Note that $\mu^{-1}(\mathcal{A})$ is the infimum (see Eq. 14), so that it may be equal to $+\infty$, if the curve μ never creates $\mathcal{A}$ events.

Using this upper time limit $\mu^{-1}(\mathcal{A})$, it is checked whether the curve μ', containing an additional `body` task of φ, creates higher interference than μ for time frames $1 \leq \Delta t \leq \mu^{-1}(\mathcal{A})$. In an implementation, this check can be performed in $\mathcal{A}$ comparisons using the pseudo-inverse of μ and μ'.

If the interference increased, the current result is updated to the maximum of μ and μ' (line 15). The maximum is taken as there may be values Δt for which $\mu(\Delta t) > \mu'(\Delta t)$. In particular, this is the case for $\Delta t < \text{BCET}(\varphi)$. Finally, line 16 updates the *done* flag to **False** to ensure that another iteration of the outer loop is performed. As noted above, this process terminates if adding another `body` task of any $\varphi \in T_{C'}$ does not create a worse interference scenario.

Theorem 2 *The loop in lines* 10 – 19 *in Algorithm* 2 *terminates after at most* $\mathcal{A} - 1$ *iterations*.

Proof The loop terminates if there is no further task $\varphi \in T_{C'}$, such that convolving the intermediate result μ with η_{φ}^{body} creates a higher interference curve for $\Delta t \leq \mu^{-1}(\mathcal{A})$. We will prove the theorem by induction. Let the value μ after i iterations be denoted by μ_{i+2}, and the maximal interference curve by $\mu_{C'}$.

Base case**:** Before reaching the loop, the algorithm computes an intermediate value, which we denote by μ_2.

This curve is maximal for up to two events. In other words, the pseudo-inverse μ_2^{-1} is minimal for $n \leq 2$. I.e., $\mu_2^{-1}(n) = \mu_{C'}^{-1}(n)$ for $n \leq 2$. This is the case as the code in lines 2 – 4 has determined the minimal time required for a single event, whereas the code in lines 5 – 9 has already performed the convolution of two tasks. Convolving additional *body* tasks to the result will not cause 2 events to occur more quickly than $\mu_2^{-1}(2)$ cycles.

Induction step**:** Given a curve μ_N such that $\mu_N^{-1}(n) = \mu_{C'}^{-1}(n)$ for $n \leq N < \mathcal{A}$. Another iteration of the loop in lines 10 – 19 will produce μ_{N+1}, such that:

$$\forall n \leq (N+1) : \mu_{N+1}^{-1}(n) = \mu_{C'}^{-1}(n). \tag{23}$$

Given Eq. (23) holds, the theorem follows directly, as the algorithm only continues to iterate as long as there is a change in the interference for $\Delta t \leq \mu^{-1}(\mathcal{A})$. As the loop starts with the given value μ_2 and needs an additional iteration to notice no change has happened, the loop will stop after $(\mathcal{A} - 1)$ iterations.

We will now prove by contradiction that Eq. (23) holds. Assume that there exists an M, such that M is the smallest $M > N + 1$ with

$$\mu_M^{-1}(N+1) < \mu_{N+1}^{-1}(N+1) \tag{24}$$

Let $\Delta t = \mu_M^{-1}(N+1)$, i.e., Δt is the minimal duration to observe at least $N+1$ events, according to μ_M. The curve μ_M creates higher interference than μ_{N+1} over Δt cycles. This means that $\mu_M(\Delta t) > \mu_{N+1}(\Delta t)$.

By definition of μ_M, there is a task φ such that we can construct μ_M as the convolution of μ_{M-1} and a body task of φ. In particular, by definition of the convolution, there is a $\delta \leq \Delta t$ such that Eq. (25) holds:

$$\mu_M(\Delta t) = \mu_{M-1}(\delta) + \eta_\varphi^{body}(\Delta t - \delta). \tag{25}$$

There are two different cases based on the value of $0 < \delta \leq \Delta t$. If $\delta = \Delta t$, we have arrived at the contradiction due to the choice of M:

$$\mu_M(\Delta t) = \mu_{M-1}(\Delta t) + \eta_\varphi^{body}(0) \leq \mu_{N+1}(\Delta t). \tag{26}$$

Equation (26) holds because $\mu_{M-1}(\Delta t) \leq \mu_{N+1}(\Delta t)$ by choice of M and $\eta_\varphi^{body}(0) \leq 0$ for any task φ. Equation (24) and Eq. (26) contradict. We thus assume that δ is strictly smaller than Δt.

We note that an iteration only increases the interference. The computed curve never decreases after another iteration, due to the use of the max operator in line 15.

We know that $\mu_{M-1}^{-1}(n) = \mu_{N+1}^{-1}(n)$ for $n \leq N$ because these values are already optimal due to $\mu_N^{-1}(n) = \mu_{C'}^{-1}(n)$ for $n \leq N$ and because $N < (M-1)$. Furthermore, it is true due to the choice of M that $\mu_{M-1}^{-1}(N+1) \geq \mu_{N+1}^{-1}(N+1) > \Delta t$. From this it follows that $\mu_{M-1}(\delta) = \mu_{N+1}(\delta)$ as $\delta < \Delta t$:

$$\begin{aligned} \mu_M(\Delta t) &= \mu_{M-1}(\delta) + \eta_\varphi^{body}(\Delta t - \delta) \\ &= \mu_{N+1}(\delta) + \eta_\varphi^{body}(\Delta t - \delta). \end{aligned} \tag{27}$$

We know that $\mu_N(\delta) = \mu_{N+1}(\delta)$ because $\delta < \Delta t < \mu_{N+1}^{-1}(N+1)$ and $\mu_N^{-1}(n) = \mu_{N+1}^{-1}(n)$ for $n \leq N$:

$$\begin{aligned} \mu_M(\Delta t) &= \mu_{M-1}(\delta) + \eta_\varphi^{body}(\Delta t - \delta) \\ &= \mu_{N+1}(\delta) + \eta_\varphi^{body}(\Delta t - \delta) \\ &= \mu_N(\delta) + \eta_\varphi^{body}(\Delta t - \delta). \end{aligned} \tag{28}$$

By definition of μ_{N+1}, we have arrived at the contradiction:

$$\begin{aligned} \mu_M(\Delta t) &= \mu_{M-1}(\delta) + \eta_\varphi^{body}(\Delta t - \delta) \\ &= \mu_{N+1}(\delta) + \eta_\varphi^{body}(\Delta t - \delta) \\ &= \mu_N(\delta) + \eta_\varphi^{body}(\Delta t - \delta) \\ &\leq \mu_{N+1}(\Delta t). \end{aligned} \tag{29}$$

We have consequently arrived at a contradiction by assuming that there exists a curve μ_M, computed by the iterative convolution of `body` tasks, that yields greater

interference than μ_{N+1} for time frames $< \mu_{N+1}^{-1}(N+1)$. As the loop only continues iterating as long as there is an increase in the curve for interference levels $\leq \mathcal{A}$, we have shown that the loop will not change the result of μ after $\mathcal{A}-2$ iterations. This is true as the initial value of μ when entering the loop is already maximal for 2 events. A final iteration is needed by the algorithm as written in Algorithm 2 to detect that no change has occured, resulting in the final maximal loopbound of $\mathcal{A}-1$ iterations. □

Note in particular that it also follows from the proof of Theorem 2 that the value μ computed by Algorithm 2 is equal to the worst-case interference curve $\mu_{C'}$ for up to $\mathcal{A}$ events:

$$\forall \Delta t \leq \mu_{C'}^{-1}(\mathcal{A}) : \mu(\Delta t) = \mu_{C'}(\Delta t). \tag{30}$$

Corollary 1 *The complexity of Algorithm 2 is* $\mathcal{O}(|T_{C'}|^2\mathcal{A}^2 + |T_{C'}|\mathcal{A}^3)$.

Proof The interference curves and the intermediate result can be stored as their pseudo-inverse, which can be represented as an array of $\mathcal{A}$ numbers. Consequently, the loop in lines 2–4 requires $|T_{C'}|\mathcal{A}$ operations. The nested loops in lines 5–9 perform $|T_{C'}|^2$ convolutions, each of which requires $\mathcal{A}^2$ operations. The maximum of the convolution and the intermediate result can be determined in $\mathcal{A}$ operations. Therefore, lines 5–9 are performed in $\mathcal{O}(|T_{C'}|^2\mathcal{A}^2)$ operations. From Theorem 2, we know that the loop spanning lines 10–19 is executed at most $\mathcal{A}-2$ times. In each iteration, a convolution is performed for each task in $T_{C'}$. Updating the intermediate result μ requires $\mathcal{A}$ operations.

For this reason, the algorithm has a time complexity of $\mathcal{O}(|T_{C'}|\mathcal{A} + |T_{C'}|^2\mathcal{A}^2 + |T_{C'}|\mathcal{A}^3)$, which is equivalent to $\mathcal{O}(|T_{C'}|^2\mathcal{A}^2 + |T_{C'}|\mathcal{A}^3)$. □

For a given hardware architecture, the associativity of the shared cache is a constant, relatively small number. Therefore, the complexity of the algorithm collapses to $\mathcal{O}(|T_{C'}|^2)$ for large task sets.

We can safely bound the inter-core interference at a shared cache for non-preemptive scheduling using Algorithm 2. The total interference from multiple interfering cores is bounded by the sum of the interference of each individual core:

$$\gamma_\tau(\Delta t) = \sum_{C' : \tau \in C \wedge C' \neq C} \mu_{C'}(\Delta t). \tag{31}$$

6 Timing-aware cache hit classification

In Sect. 6.1, we construct a cache hit classification criterion for individual cache accesses using the interference curves introduced in the previous section. We use this classification approach to construct a data-flow analysis in Sect. 6.2, which determines whether a cache access will hit or potentially miss in the LLC. Hence, this section corresponds to the fourth step of the presented analysis approach. The output of the DFA are the hit classifications that are used in the final WCET analysis, which is the final step of the timing-aware shared cache analysis.

6.1 Cache hit condition

Both intra-task and inter-core interference contribute to the eviction of cache blocks. We account for these interference sources as follows: We determine the intra-task interference on the path between subsequent accesses to a particular block $b \in \mathcal{B}_\tau$ as the set of accessed cache blocks $b' \neq b$, which are mapped to the same cache set. Inter-core interference is quantified using the interference curves described in the previous section. The key observation is that, in an LRU cache, an access to a cache block moves the block to the youngest position in the cache. Therefore, if a subsequent access to that same block happens quickly enough, inter-core interference will not be able to evict the block from the shared cache. For this reason, we determine the path duration between cache accesses to the same cache block.

To analyze intra-task interference, we abstract a path in the CFG to a sequence of accesses $(a_i)_{i=1}^m$. The access sequence of a path $\pi = (v_1, \dots, v_n)$ is given by concatenating the access sequences associated to each individual node $v \in \pi$.

Definition 4 Let $\pi = (v_1, \dots, v_p)$ be a path with associated access sequence $(a_i)_{i=1}^m$. The intra-task interference on π between two accesses a_1 and a_m to the same cache block $cb(a_1) = cb(a_m)$ is given by

$$int((a_i)_{i=1}^m) = \left\{ cb(a_s) \;\middle|\; \begin{array}{l} 1 \leq s \leq m : \\ cb(a_s) \neq cb(a_m) \wedge \\ \mathrm{CAC}(a_s) \neq N \end{array} \right\}. \tag{32}$$

Without loss of generality, we assume that the analyzed access pair occurs as the first and last accesses in the sequence associated to π, i.e. a_1 is the access to move the target block into the youngest cache position and a_m is the access to be classified. Accesses occurring prior to the initial access to the targeted cache block or after the classified access do not contribute to the intra-task interference.

The set $int((a_i)_{i=1}^m)$ consists of all cache blocks which may be accessed in the sequence $(a_i)_{i=1}^m$ that conflict with the target cache block $cb(a_m)$. Given the associativity $\mathcal{A}$ of the shared cache, we can define the eviction distance along a path.

Definition 5 The eviction distance ξ of a cache block $cb(a_m)$ along a path with access sequence $(a_i)_{i=1}^m$ is the minimal number of additional interfering cache accesses that could lead to a cache miss for a_m:

$$\xi((a_i)_{i=1}^m) = \mathcal{A} - \left|int((a_i)_{i=1}^m)\right|. \tag{33}$$

As mentioned in Sect. 4, we want to analyze paths that may lead to a cache hit on the shared cache. These paths are characterized by a positive eviction distance. To denote such paths, we introduce the notion of a *potential-hit path*.

Definition 6 A path π with associated cache access sequence $(a_i)_{i=1}^m$ is called a potential-hit path for the access a_m iff:

$$\mathrm{CAC}(a_m) \neq N, \tag{34a}$$

$$cb(a_1) = cb(a_m) \wedge \mathrm{CAC}(a_1) = A, \tag{34b}$$

$$0 < \xi((a_i)_{i=1}^m). \tag{34c}$$

Equation (34a) contains the requirements for the access a_m to potentially result in a hit on a particular path: (a) the access a_m may reach the shared cache, (b) the targeted cache block is loaded into the cache by a_1, (c) intra-task interference does not cause the eviction of $cb(a_m)$.

The only missing piece to classify the access a_m as a cache hit is to check whether the inter-core interference is less than the eviction distance on the analyzed path π. As seen in Eq. (11) and Eq. (31), using interference curves, the inter-core interference may be quantified as a function of time. By evaluating γ_τ for the WCET of the considered path, a safe bound on the inter-core interference can be given. We call the WCET of the path π the temporal reuse distance of the cache block $cb(a_m)$.

We will now construct a cache hit classification that depends on the temporal reuse distance of the cache block and the interference function derived from the event-arrival curves discussed in Sect. 5.

Theorem 3 *An access $a \in Acc$ always results in a cache hit if it may only be reached by traversing potential-hit paths π with access sequence $(a_i)_{i=1}^m$, $a_m = a$ satisfying*:

$$\gamma_\tau(\mathrm{WCET}(\pi)) < \xi((a_i)_{i=1}^m). \tag{35}$$

Proof Consider the scenario that the access a could result in a cache miss. This may happen for three different reasons: the targeted cache block was (a) not loaded into the shared cache previously, (b) evicted due to intra-task interference, or (c) evicted due to inter-core interference.

However, all executions containing a must load the targeted cache block into the cache and this block will not be evicted due to intra-task interference, as π is a potential-hit path. Furthermore, $\gamma_\tau(\mathrm{WCET}(\pi))$ is a safe upper bound on the number of aging events due to interfering accesses from competing tasks. As Eq. (35) requires that the eviction distance is strictly greater than the interference, the cache

block $cb(a)$ will not be evicted due to inter-core interference. Thus, the access will always result in a cache hit. □

The condition given in Eq. (35) can be written equivalently using the time-to-live function:

$$\mathrm{WCET}(\pi) \leq \mathrm{TTL}_{\tau}\left(\left|int((a_i)_{i=1}^{m})\right|\right). \tag{36}$$

At this point, we are able to quantify cache interference using event-arrival curves and have formulated a sufficient condition to classify an access as a hit. In the next section, we describe how we can efficiently use these two components to derive a hit / potential-miss classification for every access.

6.2 Cache access path analysis

In this section, we utilize Theorem 3 to determine whether specific accesses to the shared cache definitively result in a cache hit. To this end, we perform a data-flow analysis (DFA) on the CFG of the analyzed task $\tau \in T_C$. In the DFA, we examine accesses that would result in a cache hit, provided that no inter-core interference occurs. Checking the condition given in Eq. (35) for a particular access $a \in Acc_{\tau}$ requires knowledge of the WCET of potential hit-paths leading to the access and their respective eviction distance. Consequently, we may abstract a path from a concrete sequence of nodes to a safe upper bound on its execution time and a bound on the intra-task interference, measured as a set of conflicting blocks potentially accessed on the path.

Path information for access classification is represented in an abstract domain using a semi-lattice $D = (\mathbb{N} \times 2^{\mathcal{B}_{\tau}}) \cup \{\bot, \top\}$. Elements $(\Delta t, B) \in \mathbb{N} \times 2^{\mathcal{B}_{\tau}}$ represent a maximal path duration of Δt cycles and intra-task interference $B \subseteq \mathcal{B}_{\tau}$. The value $\top$ corresponds to a potential cache miss, while $\bot$ indicates a finished load-access path. I.e., another access will load $cb(a)$ into the shared cache en route to the access a, resulting in a cache hit. The tuples are ordered $(\Delta t_1, B_1) \leq (\Delta t_2, B_2) \iff \Delta t_1 \leq \Delta t_2 \wedge B_1 \subseteq B_2$, while $\top$ is the greatest element and $\bot$ is the least element. Joining of two elements is performed by the operator $\sqcup$:

$$(\Delta t_1, B_1) \sqcup (\Delta t_2, B_2) = (\max(\Delta t_1, \Delta t_2), B_1 \cup B_2), \tag{36a}$$

$$d \sqcup \top = \top,\ d \sqcup \bot = d,\ d \in D. \tag{36b}$$

To compute the data-flow information for all nodes in the control-flow graph, we conduct a data-flow analysis in the backward direction. We specify the data-flow information as the mappings $in[v] : Acc_{\tau} \to D$ and $out[v] : Acc_{\tau} \to D$ for $v \in V_{\tau}$. Although the analysis is conducted in the backward direction, we use the word *in* (*out*) to denote the data-flow information at the beginning (end) of a node in the regular sense.

Every node $v \in V_{\tau}$ possesses an associated sequence of cache accesses, which is denoted using the superscript v, $(a_i^v)_{i=1}^m$. The impact of a single cache access a_i^v issued during the execution of v on the analyzed access a is determined by the function f_i^v:

$$f_i^v(a,(\Delta t,B)) = \begin{cases} (a,\top) & \text{if } \gamma_\tau(\Delta t) \geq \mathcal{A} - |B'| \\ (a,\bot) & \text{else if } \begin{matrix}\mathrm{CAC}(a_i^v) = A \wedge \\ cb(a) = cb(a_i^v)\end{matrix} \\ (a,(l,B')) & \text{otherwise} \end{cases} \tag{36c}$$

$$f_i^v(a,\top) = (a,\top),\ f_i^v(a,\bot) = (a,\bot), \tag{36d}$$

where

$$B' := B \cup \left\{ cb(a_i^v) \;\middle|\; cb(a) \neq cb(a_i^v) \wedge \mathrm{CAC}(a_i^v) \neq N \right\}. \tag{37}$$

The set B', defined in Eq. (37), contains all blocks that may be accessed by the analyzed task τ on paths leading to the analyzed access a and conflict with the target block $cb(a)$.

The first case in Eq. (36c) covers the situation in which the interference on the shared cache is too high to guarantee a cache hit. This decision is made based on the bounded path duration Δt and the cumulative interference curve γ_τ. If the interference is at least as high as the resilience $\mathcal{A} - |B'|$, where $|B'|$ is an upper bound on the intra-task interference, the block may be evicted by inter-core interference $\gamma_\tau(\Delta t)$. Thus, the value is updated to $\top$, showing that this access is a potential miss.

In the second case, the path duration is acceptable and the access a_i^v actually causes the target block $cb(a)$ to be loaded into the cache. The value is updated to $\bot$ to show that the load-access path is terminated at this point and a will result in a definite hit from this location. In the final case, the data-flow information is propagated further with the updated set of conflicting cache blocks B'. The functions f_i^v can be composed to operate on sequences of accesses $f_{\alpha,\ldots,\beta}^v = f_\alpha^v \circ \ldots \circ f_\beta^v$.

Data-flow information for accesses contained in v is initially generated by $G[v]$:

$$G[v] = \left\{ f_{1,\ldots,(t-1)}^v(a_t^v,(\mathrm{WCET}(v),\emptyset)) \;\middle|\; 1 \leq t \leq m : a_t^v \text{ is potential-hit} \right\}. \tag{38}$$

For every access $a_t^v \in (a_i^v)_{i=1}^m$ associated to the node v, potentially resulting in a cache hit, $G[v]$ inserts a tuple representing an abstract path. The initial path duration is approximated by the worst-case execution time of the node v: $\mathrm{WCET}(v)$. The initial path information, consisting of the duration $\mathrm{WCET}(v)$ and the empty set for intra-task interference, is propagated backwards inside the node v using the function $f_{1,\ldots,(t-1)}^v$. Note that only accesses from v occurring prior to a_t^v, i.e., $a_i^v, 1 \leq i < t$ can create a conflict for a_t^v at this stage.

Data-flow information arriving at v from successor nodes is contained in $out[v]$. This information is propagated backwards along v by $P[v]$:

$$P[v] = \left\{ f_{1,\ldots,m}^v(a,(\Delta t + \mathrm{WCET}(v),B)) \;\middle|\; (a,(\Delta t,B)) \in out[v] \right\}. \tag{39}$$

The propagation function $f_{1,\ldots,m}^v$ is applied to the accesses contained in $out[v]$ which are not mapped to $\top$ or $\bot$. We do not need to process elements $(a,\top) \in out[v]$ and $(a,\bot) \in out[v]$, as the analysis has already concluded that the access a will result

in either a potential cache miss or definite cache hit, respectively. The path duration Δt is increased to $\Delta t + \text{WCET}(v)$ to reflect the fact that it may increase by up to $\text{WCET}(v)$ cycles by extending the path over v.

Increasing the WCET of the path by $\text{WCET}(v)$ can overapproximate the actual path duration and reduce the analysis precision. This overestimation occurs because the required time of accesses and computations that are not part of the path are included in $\text{WCET}(v)$. Consider the following example for clarification: Assume a basic block issues the access sequence b_1, b_2, b_1, b_3 to the shared cache. We want to analyze the second access to the cache block b_1. The potential-hit path for that access thus consists of the first three accesses b_1, b_2, b_1. Using Eq. (38), the duration of the path is safely estimated to be the WCET of the complete basic block $\text{WCET}(v)$. We can refine this calculation by considering the bus stalling delay in isolation. Instead of including the bus stalling penalties directly in the execution time of a basic block, we may account for potential bus contention separately. To determine the maximal duration of a potential-hit path, we add the maximal stall time for every access to the shared cache that occurs on the path. This approach leads to more precise path durations, as the bus stalling penalty is considered only for accesses that actually lie on the analyzed path. This optimization can be implemented by computing the value $\text{WCET}(v)$ without potential bus stalling delays and adjusting Eq. (36c) to increment the duration l for each access to any cache set. Note that this refinement requires a compositional hardware architecture.

The incoming data-flow information for node v is the combination of $G[v]$ and $P[v]$ (Eq. 40), whereas the outgoing data-flow information consists of the merged information from all successors $w \in V_\tau$ with $(v, w) \in E_\tau$ (Eq. 41).

$$in[v] = G[v] \sqcup P[v] \tag{40}$$

$$out[v] = \bigsqcup_{(v,w)\in E_\tau} in[w] \tag{41}$$

Here, the join operation $\sqcup$ is extended to data-flow mappings by pair-wise joining of elements associated to the same access event.

Each node is initialized with the empty set: $in[v] = out[v] = \emptyset$. The values of $in[v]$ and $out[v]$ are computed iteratively until they converge. Then, an access a can be safely classified as a cache hit if no node v exists with $(a, \top) \in in[v]$.

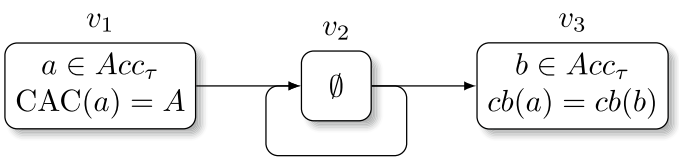


Fig. 4 Example CFG containing three nodes v_1, v_2, v_3. The nodes v_1 and v_3 each contain an access to the same cache block. The node v_2 is part of a loop and may be executed multiple times

6.2.1 Ensuring termination

In its current formulation, the semi-lattice D used in the DFA has infinite height, as the path duration is not limited. Therefore, D does not satisfy the ascending chain condition. In other words, it is not guaranteed that the data-flow information converges after a finite number of iterations. We modify the analysis to guarantee termination in the following way.

We know that all feasible paths contained in the analyzed CFG are of finite duration as we assume that all tasks terminate and thus have a finite WCET. In practice, however, non-termination of the analysis can occur when information is propagated along a path that is infeasible in reality.

Consider the example CFG in Fig. 4. The graph contains three nodes v_1, v_2, v_3. The nodes v_1 and v_3 access the same cache block and the access in v_1 is performed in all circumstances. The access in v_3 may result in a cache hit depending on the duration of the path from v_1 to v_3. However, the CFG contains a loop over node v_2, which is part of the path from v_1 to v_3.

Assume that the interfering cores will never create a sufficient amount of interference to evict the block accessed in v_3. The data-flow analysis will not terminate, as the loop duration is potentially unbounded. Note that this is the case even when an upper loop bound exists, as this information is not available to the data-flow analysis. The DFA is inherently unaware of how many loop iterations have been performed already.

To correct this behavior, it is possible to limit the maximal duration value in the abstract domain. Instead of allowing the path duration to take an arbitrary duration $\Delta t \in \mathbb{N}$, only a limited interval $[1,\ L_\tau] \subset \mathbb{N}$ can be permitted. A natural choice for L_τ is the lowest duration to experience the maximal interference over one full execution of τ, i.e., $\gamma(\text{WCET}(\tau))$. L_τ is thus given by:

$$L_\tau = \gamma_\tau^{-1}(\gamma_\tau(\text{WCET}(\tau))). \tag{42}$$

In case a path duration exceeds this value while being propagated along a node, the duration is instead capped at the upper limit L_τ. This limit on the path duration is safe, as it never underestimates the potential inter-task interference due to the choice of L_τ. Using this upper limit on the path duration, only a finite number of updates may be applied to an abstracted path until the value necessarily stabilizes.

Termination is now ensured, but in practice, the number of iterations required for termination may still be prohibitively large. To solve this problem, we widen an abstracted path $(\Delta t, B)$ to $\top$ after the number of performed propagations over nodes exceeds a certain threshold value. This procedure is safe as we do not introduce faulty cache hit classifications here. However, some precision is sacrificed. We evaluate the impact of different threshold values in Sect. 7.3.

6.2.2 Relative precision

In this section, we discuss the relative precision of the presented analysis compared to the baseline analysis, which was first presented in Hardy et al. (2009).

We abbreviate the timing-aware classification presented in this paper as the TAC method.

It is clear that the quantification of interference over time using event-arrival curves can yield more precise results than assuming that every potentially conflicting block causes interference at all points in time. However, the TAC classification approach presented in this paper is not strictly more precise than the CCN method. This is caused by the inherent loss of precision in the path abstraction.

Consider the control-flow graph in Fig. 4 as an example. Assume that the shared cache is 4-way associative and that interfering tasks may store up to 3 conflicting blocks in the cache. The CFG of the analyzed task contains three nodes v_1, v_2, v_3. The accesses $a \in Acc_\tau$ and $b \in Acc_\tau$ are contained in the nodes v_1 and v_3, respectively. In this scenario, the access a in v_1 causes the block $cb(a)$ to be loaded into the shared cache, while the node v_2 does not contain any accesses to the shared cache. When performing the access b with $cb(b) = cb(a)$, no further blocks were loaded into the cache by the analyzed task. Thus, the block age of $cb(b)$ based on intra-task interference is 0.

An eviction of $cb(b)$ prior to the access b would require interfering tasks to store 4 conflicting blocks in the shared cache. As, in this example, the interfering task only accesses up to 3 conflicting blocks, the CCN approach is able to classify the access as a definite cache hit.

The DFA utilized in the TAC approach has to process the loop which allows the node v_2 to be executed multiple times between v_1 and v_3. The duration of the path from v_1 to v_3 thus depends on the number of iterations performed by the loop v_2. Consequently, the path information associated to the access b originating from v_3 can be propagated many times along the loop v_2. This is the case as the duration Δt in the abstract path information $(b, (\Delta t, \emptyset))$ does not converge until the limit L_τ is reached. As we have discussed in the previous section, the DFA is configured to have a propagation limit for path information, which ensures the quick convergence of the analysis. Depending on the value of the propagation threshold, abstract path information may be widened to $(b, \top)$. The resulting classification for b by the TAC approach could thus be a potential cache miss.

We conclude that the TAC technique is not strictly more precise compared to the classification provided by the standard CCN analysis. However, it is possible to combine the two techniques to create a more precise analysis. To consider a cache access as a hit, it is sufficient that either the CCN analysis or the TAC analysis classifies the access as a cache hit. To combine the two classifications approaches, first, the CCN classification is determined. If the result is not a conclusive cache-hit, TAC analysis can be utilized to decide whether the access is a cache hit or potential miss.

It is important to note that the cache hit classifications of the combined approach are not merely the union of the hit classifications from the timing-aware and baseline techniques when considered in isolation. In fact, a beneficial interaction emerges when both approaches are combined: A cache access that is not classified as a cache hit by either the timing-aware or baseline classifications alone can be classified as a cache hit when both approaches are combined. This situation can occur when the timing-aware classification fails to classify an access as a cache hit because the potential hit-path is too long. The potential hit path duration can be reduced in the

combined analysis if a cache access on the potential hit path is classified as a cache hit by the baseline classification but not the timing-aware classification. In the combined analysis, this reduction of the potential-hit path duration can lead to additional hit classifications that are not possible when the approaches are considered in isolation.

We compare the precision of the combined TAC+CCN approach with the pure TAC analysis in the evaluation in Sect. 7.

6.2.3 Iterative application

The DFA of the presented classification approach contains an implicit dependence between the classification of different accesses. To determine a safe upper bound for the duration of a path, the WCET of the contained nodes is accumulated. Initially, these WCET values are derived under the assumption that no access to the shared cache results in a cache hit. However, after classifying some accesses as definitive cache hits, these WCET values may be reduced. As a consequence, the WCET estimate of paths in the CFG also reduces. The DFA can thus be performed a second time using the tighter WCET values. Accesses previously classified as potential misses may now be classified as cache hits due to the shorter duration of the potential-hit paths. Hence, it is possible to apply the DFA iteratively to gain more and more precise cache hit information in each iteration. The iterative DFA analysis is performed as long as there are changes in the access classifications. This process will terminate, because the there is a limited number of cache accesses that can be classified as cache hits and a hit classification made in a previous iteration will never be invalidated, as the potential-hit path duration decreases monotonically.

Note that this iterative process has no impact on the interference curves. The interference curves are derived under the best case assumption, that no interference will occur at the shared cache. Thus, the interference curves remain the same for all iterations. Only the upper bound on the path duration is tightened.

7 Evaluation

In order to evaluate the performance and effectiveness of the timing-aware classification approach, we implemented it in a WCET analyzer (Falk and Lokuciejewski 2010). This tool chain supports the code-generation and WCET analysis for the ARMv4T instruction set. The target architecture consists of multiple cores with a 3-stage in-order pipeline. Each core is equipped with a private L1 cache, which is connected to a shared L2 cache using a round-robin arbitrated bus. Our evaluations cover systems with 2 and 4 cores, where each core executes up to 4 different tasks. The systems use partitioned scheduling, which means that the mapping of tasks to their core is fixed. For all configurations, we generated 20 random task sets. The tasks are taken from the EEMBC AutoBench 1.1 benchmark suite (Poovey et al. 2009) to create a realistic workload. Tasks are released repeatedly, with the minimal inter-arrival time being denoted by the value $Period(\cdot)$. We generated task periods

Table 3 Worst-case access timing including the bus access delay using round-robin arbitration

Cores	L2 hit	L2 miss	Hit-miss ratio
2	50	80	0.63
4	130	160	0.81

using the UUnifast approach (Bini and Buttazzo 2005), with a target utilization of 0.7. The target utilization refers to the best-case scenario, without any inter-core interference.

References, which are located in a loop often exhibit different behavior on the first loop iteration compared to subsequent loop iterations. For this reason, we employ virtual inlining and unrolling, limited at a depth of 3 contexts. This means that the analyzer differentiates between the first, second, and all following iterations of a loop, as well as different function call contexts, thereby improving the precision of the WCET estimate.

As discussed in Sect. 6.2.1, the DFA to classify cache accesses possesses a propagation threshold parameter. This means that the path information is not propagated beyond a certain number of nodes, in order to ensure quick termination of the analysis In this evaluation, the propagation limit of the analysis was set to 30 nodes, meaning that hit paths containing up to 30 nodes can be detected by the DFA to result in a cache hit. The impact of this parameter on the precision and runtime is evaluated in Sect. 7.3.

We configured the private L1 caches with a size of 256 bytes, using direct-mapping and a cache block size of 16 bytes. For the shared cache, we evaluated cache sizes ranging from 4 to 64 KB, with 8-way associativity, and cache block size of 64 bytes. Both cache levels use the LRU replacement policy. Data accesses were not cached and the timing for each access was set to take 3 cycles. The instruction access timings were set to 1 cycle for an L1 hit, 10 cycles for an L2 hit and 40 cycles in case of an L2 miss. These timing values do not include potential bus access delays.

An instruction fetch may be stalled for up to $(|\mathcal{C}| - 1) \cdot 40$ cycles at the shared bus due to accesses from other cores. The worst-case access timing consists of the sum of the bus access delay and the L2 access time. Table 3 shows the different timing values including the worst-case bus stall time. As the number of cores increases, the relative difference between a cache hit and cache miss diminishes, because the worst-case access delay is primarily determined by the delay to access the shared bus. Thus, improvements in the cache hit classifications may have a smaller impact on the WCET of a task than expected, since the WCET is also affected by other factors such as the bus access delay.

To evaluate the performance, we utilize two different metrics. The first metric is the percentage of accesses contained in the CFG that could be classified as a cache hit. The second metric is the relative WCET achieved by the analyses compared to the WCET using the baseline CCN classifications as the reference value.

We use the hit ratio in addition to the relative WCET, as the WCET value does not capture improved classifications outside the critical path. Additionally, changes in WCET might be small, even though a significant number of accesses were newly classified as cache hits as the WCET is also influenced by other factors such as the bus access delay.

We compare the performance of four different analysis approaches:

1. Baseline classification (CCN)
2. Worst-case interference placement (WCIP)
3. Timing-aware classification (TAC)
4. Timing-aware classification combined with baseline classification (TAC+CCN)

The baseline analysis counts the number of conflicting cache blocks from tasks running on interfering cores for each cache set. The analysis then assumes that all potential conflicts occur for every access to the shared cache. This approach was used in Hardy et al. (2009) and Liang et al. (2012). An access to a cache block is only classified as a cache hit if the maximal block age is less than the cache associativity minus the number of potential conflicts.

The second approach we evaluate is the Worst-Case Interference Placement analysis, as presented by (Nagar and Srikant 2014, 2016; Nagar 2016). An upper bound on the number of interfering cache accesses is determined and distributed to create the maximal possible cache miss penalty. In the evaluation, we use the approximate WCIP algorithm, as it is reported to have similar performance to the precise solution, with lower analysis overhead. The technique is a quantitative analysis as it only computes a WCET value without classifying individual accesses. Thus, we are only able to compare the resulting WCET values but not the number of issued hit classifications. We selected the WCIP approach over the CEOP approach (Zhang et al. 2022) as the comparison because it promises to scale better for multiple interfering tasks (see Observation 2).

The third approach is the timing-aware classification based on interference curves, as it is presented in this paper. We determine the interference curves as described in Sect. 5 and classify individual cache accesses using the DFA presented in Sect. 6.2.

As we have discussed in Sect. 6.2.2, the precision of the timing-aware classification and baseline classification approaches are incomparable, as both techniques can perform better than the other one under certain circumstances. We also evaluate the performance of the combination of both classification approaches. This means that an access will be considered to be a cache hit if either the baseline classification or the timing-aware classification has deemed the access to be a cache hit.

The evaluation is structured into multiple parts. In Sect. 7.1, we evaluate the cache hit classification performance and WCET reduction, and in Sect. 7.2, we

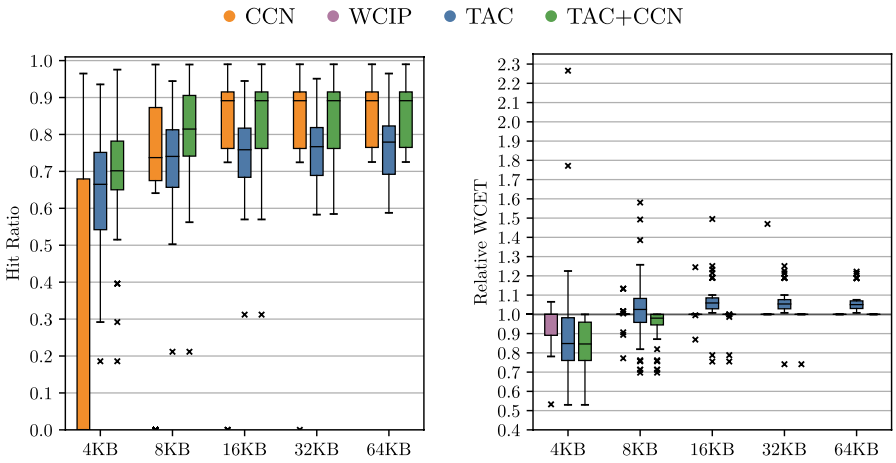


Fig. 5 Hit Ratio and relative WCET for dual-core systems with 1 task per core. WCET results are relative to the baseline CCN approach

evaluate the analysis overhead incurred by the timing-aware classification when determining the WCET estimate. The impact of the propagation threshold value, which was discussed in Sect. 6.2.1, is evaluated in Sect. 7.3.

7.1 WCET reduction

In this subsection, we evaluate the performance of the presented analysis and compare it to other techniques. We use box plots to visualize the results. The line in the middle of each box represents the median value, whereas the lower and upper bounds of a box show the 25th and 75th percentile. The whiskers are at most 1.5 times as large as the central box. Data points outside this range are considered outliers and are marked by a small cross. The metrics are evaluated for each task and grouped for each system configuration.

The results of the baseline CCN approach are colored orange. The WCIP approach is colored purple. The timing-aware classification results, as a standalone analysis and combined with the baseline approach, are colored in blue and green, respectively.

Note that the WCIP approach is a quantitative approach, which does not issue hit classifications. Therefore, the method is absent in the hit ratio evaluation. For the WCET comparison, we normalize the result of the CCN approach at 1.0. A relative WCET smaller than 1.0 corresponds to a reduced WCET, while a larger value means the approach performed worse than the CCN approach.

The results for dual-core systems and quad-core systems are presented in Sect. 7.1.1 and Sect. 7.1.2, respectively.

7.1.1 Results for dual-core systems

We analyzed dual-core systems with 1, 2, and 4 tasks per core. For each task set size, we generated 20 task sets to be analyzed.

The performance results for single task systems are visualized in Fig. 5. The left-hand figure shows the hit ratio, i.e., the percentage of cache accesses in the control-flow graphs, which were successfully classified as a cache hit. The WCIP approach is a quantitative analysis, therefore it does not appear in the comparison of hit ratios. The right-hand figure shows the relative WCET compared to the CCN approach.

For the smallest cache size of 4 KB, the baseline analysis achieves a median hit ratio of 0% (26.6% avg.). Using the TAC approach, as presented in this paper, the median hit ratio increases to 66.5% (63.2% avg.). When the two approaches are combined, the hit ratio further increases to 70.2% (69.2% avg.). Consequently, we see that by leveraging timing information the hit classification precision can be drastically increased.

When increasing the shared cache size to 8 KB, the CCN approach is able to classify significantly more accesses as cache hits. The median hit ratio rises to 73.7%. However, the average is only 64.4% as there are some systems with 0% hit classifications. The TAC achieves a similar median hit ratio of 74.1%, while the average lies at 72.2%.

While both, the CCN and TAC approaches have a comparable median hit ratio, when combining both approaches, the median hit ratio increases to 81.5% (80.3% avg.). This exemplifies the positive effects that occurs when the TAC classification can make use of the CCN classifications to classify additional accesses as cache hits.

For the higher cache sizes from 16 KB to 64 KB, the CCN and TAC+CCN approaches converge to a median hit ratio of 89.1%. The pure TAC approach stabilizes at a hit ratio between 75.0% and 77.6%.

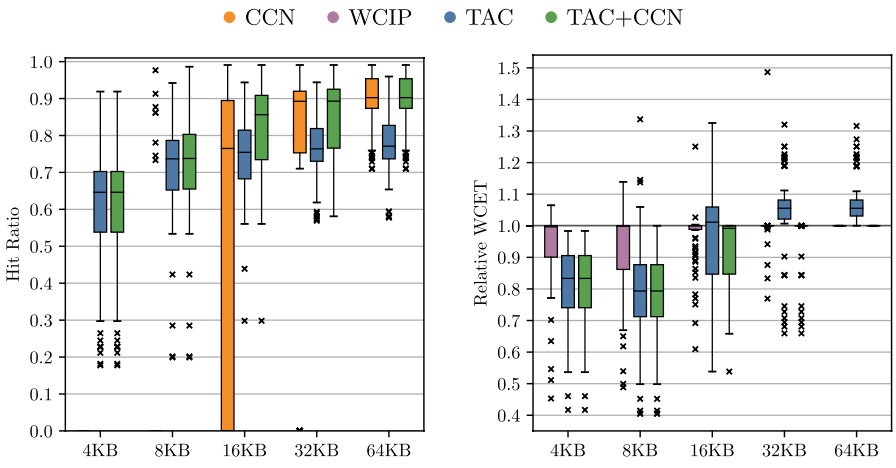


Fig. 6 Hit Ratio and relative WCET for dual-core systems with 2 tasks per core. WCET results are relative to the baseline CCN approach

The impact on the WCET estimate relative to the CCN approach is shown on the right-hand side of Fig. 5.

The biggest changes occur for the smallest 4 KB cache, as this is the configuration, where the CCN approach had a median hit ratio of 0%. In the median, the WCIP approach has no effect on the WCET compared to the CCN approach. On average, we observed 5.1% reduction in the WCET. This is congruent with the reports from Nagar and Srikant (2016), which also reports 0% median WCET reduction. More precisely WCET improvements were reported for 9 out of 27 benchmarks (Nagar and Srikant 2016). In some cases, the WCET computed using WCIP, was larger than the reference value computed using the CCN approach.

The approach presented in this paper performs better than WCIP. We measured a median WCET improvement of 15.2% and 15.4%, for the TAC and TAC+CCN methods, respectively. The average improvement for TAC is 8.9%. The smaller average improvement results from the fact that there are benchmarks in which TAC performs worse than CCN. However, when combining the TAC and CCN approaches, the average reduction stays high at 16.5%. Note, that the combined TAC+CCN approach will never perform worse than the CCN approach.

Using WCIP for the larger 8 KB cache configuration, the average and median WCET reduction is 0.3% and 0.0%. Using only the classifications from the TAC method, the WCET increases by 2.5% in the median. However, for the TAC+CCN approach, the WCET is reduced by a median of 2.0% and 6.3% on average.

For larger cache size the WCIP and TAC+CCN methods perform equally well as the CCN approach, while only using TAC classifications increases the WCET by 5.1% to 5.9% in the median.

We conclude that it is beneficial to use a combined approach, which falls back on the results of the CCN analysis, in case the more complex analyses fail to provide better results. This is the case for both the WCIP, and TAC approach, which can be outperformed by the standard CCN analysis.

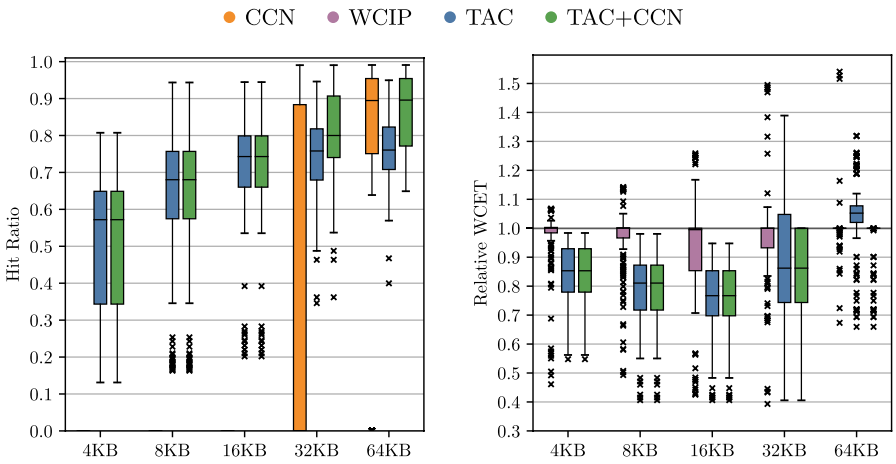


Fig. 7 Hit Ratio and relative WCET for dual-core systems with 4 tasks per core. WCET results are relative to the baseline CCN approach

We also analyzed dual-core systems with two tasks assigned to each core. Again, we evaluated 20 task sets for this configuration. The hit ratio and relative WCET reduction are shown in Fig. 6.

The results show a similar picture compared to the single task systems: For small cache configurations up to 8 KB, the standard analysis is unable to issue any hit classifications, due to the total code size of interfering tasks.

The WCIP approach is able to achieve reductions in the average WCET of 6.2% and 7.4% for the 4 KB and 8 KB L2, respectively. However, the median improvement is 0.3% and 0.1%, respectively.

The TAC and TAC+CCN methods yield significantly higher hit ratios and WCET reductions for the 4 KB and 8 KB cache sizes. The median hit ratios of 64.6% and ca. 73.8% cause a median WCET reduction of 18.4% for 4 KB and 20.5% to 21.6% for 8 KB caches, respectively. For 16 KB caches the TAC+CCN approach reduce the average WCET by 8.7%.

For the 32 KB and 64 KB cache size, the TAC approach, without the CCN classifications as fallback, performs slightly worse than the pure CCN approach, resulting in a median WCET increase of 5.6% and 5.5%. The CCN analysis gives high cache hit ratios, which prevents improvements in the WCET estimate. For these cache sizes, the cache is able to accommodate most of the code at the same time, resulting in few inter-core conflict misses. In the median, the WCET is not changed for when using the combined TAC+CCN approach. For the 32 KB shared cache, the WCET is reduced by 2.9% on average.

We further increased the task set size and analyzed dual-core systems containing four tasks per core. The hit ratio and relative WCET reduction are shown in Fig. 7. A pattern similar to the 1 task-per-core and 2 tasks-per-core systems can be observed for these systems. The presented TAC analysis is able to issue hit classifications even for small caches and thus generates a reduction in the relative WCET. For larger shared caches, the baseline analysis performance rapidly increases once a

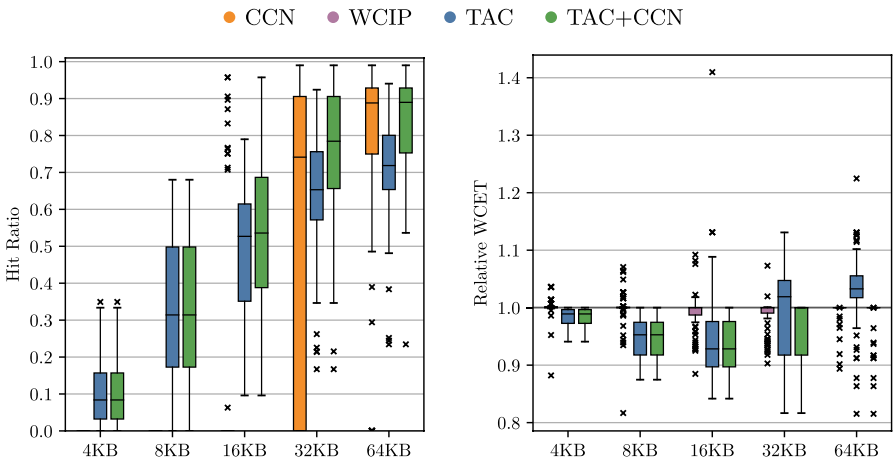


Fig. 8 Hit Ratio and relative WCET for quad-core systems with 1 tasks per core. WCET results are relative to the baseline CCN approach

threshold cache size has been reached. For even larger caches, the inter-core interference becomes less significant as most code is able to fit in the shared cache simultaneously. On average, the code size for each core was 27.7 KB.

The average WCET reductions using the WCIP approach were 3.5%, 3.7%, 7.4%, 3.6%, and −0.2% for the 4 KB to 64 KB cache sizes. In contrast, the qualitative TAC+CCN method achieved significantly higher performance compared to the CCN and WCIP methods. The average WCET reductions were 15.0%, 20.3%, 23.3%, 14.7%, and 3.0%. The performance of TAC was similar to TAC+CCN for all cache sizes below 64 KB, as the CCN approach had a median hit ratio of 0% in these configurations. Consequently, falling back to the CCN classification provided no significant precision increase.

7.1.2 Results for quad-core systems

To analyze the scalability of the presented approach to higher core counts, we also analyzed quad-core systems containing 1, 2, and 4 tasks per core, with shared caches between 4 to 64 KB.

The evaluation results for 20 systems containing a single task per core are shown in Fig. 8. Compared to the dual-core systems with one task per core, as shown in Fig. 5, the number of cache hit classifications are considerably lower. The reason behind this is that by doubling the number of cores from 2 to 4, the number of interfering cores triples from 1 to 3. Thus, a quad-core system has inherently higher inter-core interference, which makes it harder to issue definite hit classifications.

For all cache size configurations, the performance of the WCIP approach was close to the CCN approach. The average WCET reduction was between −0.1% and 1.3%. It can be seen that in the small configurations, where the CCN approach classifies all accesses as cache misses, the WCIP approach may compute a larger WCET value than the CCN method, which classifies all L2 accesses as cache misses. This

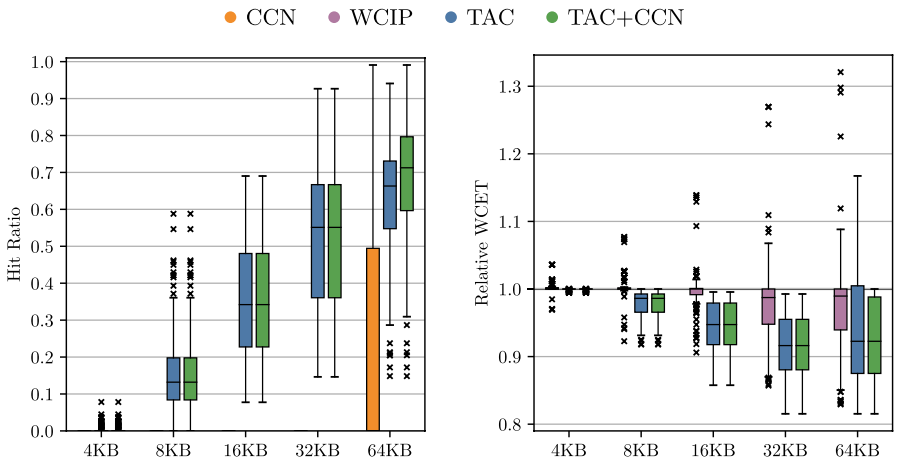


Fig. 9 Hit Ratio and relative WCET for quad-core systems with 2 tasks per core. WCET results are relative to the baseline CCN approach

overestimation is a result of the approximations made in the WCIP approach. The WCIP approach assumes that all potential cache hits lie on the worst-case path in the CFG and computes the penalty accordingly. When the potential interference is so large that all potential hits are classified as cache misses, the resulting penalty can lead to a WCET estimate which exceeds the WCET assuming all L2 cache accesses will result in a cache miss.

For the 4 KB cache the TAC analysis is classifying a smaller fraction of L2 cache accesses as hits compared to the dual-core system. The median hit ratio is 8.4%, leading to a 1.1% median WCET reduction. The performance of the analysis increases for larger cache sizes. The WCET is reduced further for 8 KB and 16 KB caches, where the TAC+CCN analysis has a median hit ratio of 31.4% and 53.3%, respectively. In contrast, the CCN analysis has a median hit ratio of 0% for up to 16 KB. These hit classifications result in a median WCET reduction of 4.7% (5.2% avg.) and 7.2% (6.7% avg.) for 8 KB and 16 KB shared caches.

For the 32 KB shared cache configuration, the CCN analysis is able to issue hit classifications for 74.1% of accesses in the median, compared to 78.5% for the TAC+CCN analysis. This gap in the hit ratio leads to an average WCET reduction of 3.7%. For the largest evaluated cache size of 64 KB, the CCN and TAC+CCN approaches lead to a similar hit classification ratio, as the impact of inter-core interference on the worst-case performance diminishes. However, there are outliers for which better hit classifications where made, leading to an average WCET reduction of 1.2%.

On average, the three interfering cores had a total code size of 19.9 KB. This is reflected in the performance of the CCN approach. For the 16 KB cache configuration, the median hit ratio is 0%. However, as soon as the shared cache size reaches 32 KB, and exceeds the code size of the interfering tasks, the hit ratio jumps to 74.1%. These results clearly demonstrate the pessimism of the CCN approach, in particular for small cache sizes.

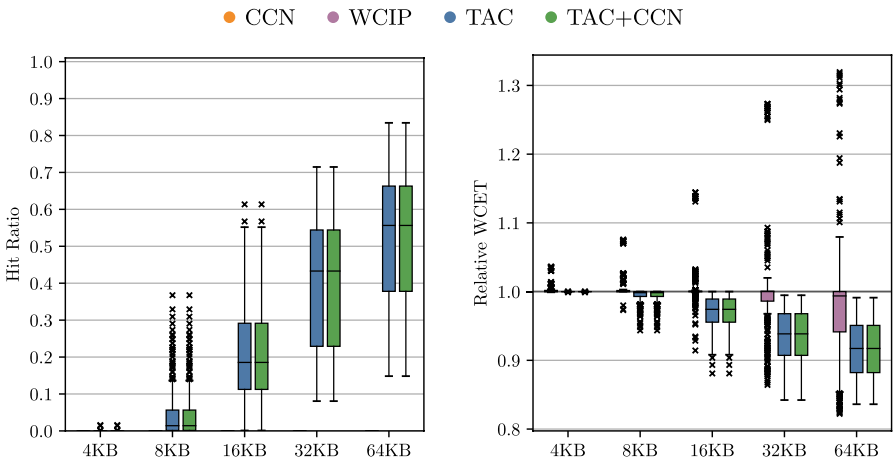


Fig. 10 Hit Ratio and relative WCET for quad-core systems with 4 tasks per core. WCET results are relative to the baseline CCN approach

We also analyzed quad-core systems with two tasks assigned to each core. This means that six different tasks can contribute to the eviction of data from the shared cache. The evaluation results for this configuration are shown in Fig. 9. The large potential of inter-core interference hinders the CCN approach to issue many hit classifications. The median hit ratio is 0% for all cache sizes.

The TAC+CCN analysis achieves a median hit ratio of 0.0%, 13.2%, 34.2%, 55.1%, and 71.3%, for the cache sizes 4 KB – 64 KB respectively. The resulting average WCET reduction is 0.1%, 2.2%, 5.4%, 8.4%, and 7.6%. The TAC and TAC+CCN methods perform identically for 4 KB – 32 KB, as the CCN does not issue hit classifications for these cache sizes. In the 64 KB configuration, relying solely on TAC hit classifications yields a WCET reduction of 5.9% over CCN, compared to 7.6% when both qualitative analyses are combined.

Using the WCIP algorithm, the WCET is reduced on average by −0.2%, −0.3%, 0.2%, 2.2%, and 2.9%. The median relative WCET remains unchanged for 4 KB to 16 KB, whereas a 1.3% and 1.0% decrease are observed for the 32 KB and 64 KB configuration.

The results for quad-core systems with 4 tasks per core are shown in Fig. 10. This means that in total 12 different tasks can contribute to the eviction of data stored in the shared cache. Furthermore, these 12 tasks can be executed repeatedly, which further increases the potential inter-core interference. As the CCN approach does not issue hit classifications, i.e., assumes all shared cache access to result in cache misses, the TAC and TAC+CCN approaches perform identically.

In contrast, the timing aware classification presented in this paper achieves an average hit ratio of 0%, 4.3%, 21.2%, 39.6%, and 51.2%. The WCET median WCET is reduced by 0.0% (0.0% avg.), 0.2% (0.6% avg.), 2.6% (3.0% avg.), 6.1% (6.3% avg.), and 8.3% (8.5% avg.), for 4 KB – 64 KB L2 caches respectively.

The quantitative WCIP method has no impact on the median WCET for cache sizes up to 32 KB. For the 64 KB cache, the relative WCET is decreased by 0.6% in the median and 1.9% on average.

7.2 Runtime evaluation

In this subsection, we take a look at the analysis overhead of the proposed analysis and compare it to the runtime of the CCN and WCIP approach. The evaluations were conducted on an Intel Xeon Server containing 48 cores running at 3.2GHz. All analyses were configured to use only a single processor core.

The runtime of the CCN method consists of the cache block conflict number summation for every cache set and a WCET analysis. It is the approach with the lowest analysis overhead.

To compute the WCET using the WCIP approach, the following steps are performed: (a) calculate WCET without inter-core interference, (b) determine number of conflicting accesses per task execution for each cache set, (c) determine number of cache hits with resilience $k = 1, \ldots, \mathcal{A}$, (d) iteratively solve the interference placement problem until the WCET converges.

In Nagar and Srikant (2016), the number of cache hits and the interference budget is calculated by determining loop bounds and the maximal execution count of each function. By determining these values, it is possible to bound the execution count of each access to the shared cache. The total access count to the shared cache and the number of hits with particular resilience is then given as the sum of all execution counts for the corresponding accesses.

We determine the interference budget by solving an IPET ILP that maximizes the number of L2 cache accesses for a single execution of the task. We apply the same ILP based approach to determine the number of cache hits with different levels of resilience. This means, that for each task and each cache set we solve $\mathcal{A}+1$ ILPs. The approach of using ILPs, which we employ in this paper, is mentioned as a potentially more precise alternative to the execution count based estimate in Nagar and Srikant (2016).

Using the period of the interfering tasks, we determine how many times each interfering task may be released during a single instance of the analysed task. The approximate WCIP algorithm computes the interference penalty based on these values. Algorithm 1 is performed multiple times until the WCET estimate converges as described in Sect. 3.2.

The runtime of the presented analysis consists of: (a) the BCET/WCET analysis used to derive the event-arrival curves and required to perform the DFA, (b) the derivation of the interference curves, (c) the computation of interference due to non-preemptive scheduling, (d) the DFA to classify accesses, (e) the final WCET analysis.

To derive the interference curves, up to $\mathcal{A}$ ILPs are solved for each task and cache set. Each of these ILPs provides a solution that is used to define the interference curves, as given in Eq. (10). Up to $2\mathcal{A}$ additional ILPs have to be solved to determine the interference curves for the `in-task` and `out-task` categories. It is not always necessary to solve all ILPs, because for example, a task that only accesses three distinct cache blocks in a cache set will never generate four or more events.

Table 4 Average analysis time in minutes

Cores	Cache size	1 Task			2 Tasks			4 Tasks		
		CCN	WCIP	TAC	CCN	WCIP	TAC	CCN	WCIP	TAC
2	4 KB	0.2	0.8	2.1	0.7	2.6	5.7	1.3	5.3	11.7
	8 KB	0.3	1.5	3.1	1.0	5.3	8.2	1.8	12.1	16.9
	16 KB	0.4	3.1	3.2	1.3	10.9	9.1	2.5	23.6	19.4
	32 KB	0.3	5.7	3.4	1.1	20.3	9.9	2.1	44.8	20.6
	64 KB	0.3	10.4	3.2	0.8	34.7	9.7	1.9	80.5	23.7
4	4 KB	0.6	2.3	5.2	1.4	6.0	12.3	2.9	12.7	25.5
	8 KB	0.8	5.0	7.3	2.0	12.5	17.5	4.2	29.7	36.7
	16 KB	1.1	9.8	8.0	2.7	25.9	20.2	5.5	61.1	42.0
	32 KB	1.0	18.8	8.2	2.5	48.4	21.6	4.9	124.5	47.8
	64 KB	0.7	35.8	8.3	1.6	88.2	23.1	3.5	220.0	57.7

Hence, the ILPs determining the time required for more than three events will be infeasible and do not need to be solved.

The interference curves are then combined according to Algorithm 2. The resulting total interference curves are used in the following data-flow analysis to classify the individual cache accesses.

Table 4 shows the average analysis runtimes of the three analysis approaches.

Using the presented analysis, the time required on average to analyse a dual-core system with a single task per core ranges from 2.1 to 3.4 minutes depending on the cache size. The TAC analysis has an overhead between 8.9× and 12.2× compared to the baseline analysis. For single task quad-core systems, the analysis took between 5.2 and 8.3 minutes on average. The overhead over the CCN analysis ranges from 7.3× to 11.4×. Setting the number of tasks per core to 2, the runtime of the TAC analysis increases to 5.7–9.9 minutes for dual-core systems and 12.3–23.1. For a task set size of 4 tasks per core, the timing-aware analysis requires between 11.7 and 23.7 minutes for two cores and from 25.5 to 57.7 minutes for four cores on average. Over all configurations, the overhead of the presented TAC analysis ranges from 7.3× to 16.6×.

From the construction of the analysis, a linear scaling of the analysis time is expected when increasing the system size. This can trend be observed in Table 4. When doubling the number of tasks for dual-core systems, the analysis runtime increases by a factor of 2.8× from 1 to 2 tasks and 2.2× from 2 to 4 tasks. For quad-core systems, the runtime scales by a factor of 2.6× and 2.2× respectively. When keeping the number of tasks constant and doubling the core count from 2 to 4 cores, the runtime scales by a factor of 2.5×, 2.2×, and 2.3×, for 1, 2, and 4 tasks per core respectively.

For the CAC and TAC analysis, the runtime for large caches can decrease compared to the medium-sized caches. For example, for dual-core systems the analysis of 64 KB caches is faster than for 32 KB caches for 1 and 2 tasks per core. With 4 tasks per core, this trend is also visible in the CCN analysis but not the TAC analysis. As this trend occurs also in the CCN analysis, we conclude that it is not inherent to the analyses but rather a property of the underlying WCET analysis tool.

For the presented analysis, the main contributor to the runtime are the ILPs formulated in Sect. 5 to derive the event-arrival curves. As the ILPs are independent of each other, this step can be parallelized to reduce the time requirement. To explore this aspect, we performed the analysis of the largest systems, four cores each executing four tasks, while allowing up to four ILPs to be solved in parallel. The required time for the analysis dropped to 12.6, 18.4, 22.9, 24.7, and 28.4 minutes, for the different cache sizes respectively. These values indicate a reduction of the required runtime by a factor of 2×. Thus, we conclude that the scalability of the presented analysis can be improved by parallelizing the required computations.

The data-flow analysis presented in Sect. 6.2 was insignificant compared to deriving the interference curves, with an average runtime of less than two seconds per system. Combining the interference curves to account for non-preemptive scheduling of tasks, i.e., executing Algorithm 2, did not create significant overhead, taking less than a second to compute.

We observe, that the runtime of our implementation of the WCIP approach is significantly higher compared to the presented TAC analysis. This is a consequence of the way the interference budget and number of cache hits are determined in our implementation of the approach. We solve multiple ILP models to determine these values. As mentioned above, Nagar and Srikant (2016) used an efficient alternative, which operates directly on the control-flow graph of each task. Thus, a more efficient implementation of the WCIP approach is possible. However, as seen in the previous section, the WCIP approach performs similar to the CCN approach, and is significantly outperformed by the presented TAC analysis.

7.3 Propagation limit sensitivity

To classify shared cache access as cache hits, the duration of potential-hit paths is analysed in the presented TAC approach. The maximal duration of these paths is determined using the data-flow analysis developed in Sect. 6.2.

In order to guarantee termination of the data-flow analysis, an upper bound L_τ is imposed on the abstract path duration. This bound corresponds the duration required to experience the maximal inter-core interference. Bounding the path duration ensures that the analysis terminates. However, it may require many iterations for a path to converge on this upper limit.

To ensure quick convergence, we introduced a propagation limit, which widens the information for an access to $\top$, when the propagation limit is exceeded. This means that after propagating path information over a certain number of nodes in the CFG without finding the start of the potential-hit path, i.e. the access that loads the target block into the shared cache, the DFA gives up on this path and classifies the access as a potential miss.

We now explore whether this cut off threshold is necessary and how it affects the analysis precision. The results from the previous sections were determined using a propagation limit of 30 nodes. This means, that potential-hit paths containing up to 30 nodes may be recognized by the DFA as resulting in a cache hit. To explore the sensitivity of the analysis to this parameter, we performed the analysis of single-task systems with different threshold values.

First the propagation limit is decreased to 5 nodes. As only short hit paths may be recognized with this setting, a decrease in the number of successful cache hit classifications is expected. We observed that a lower path propagation threshold had no significant impact on the analysis precision. The average hit ratio results were within 0.2% of the results obtained with the higher threshold of 30 nodes. To check whether a higher propagation limit is useful, we increased the propagation limit to 150 nodes. As for the lower propagation limit, the hit ratio results were within 0.2

percentage points. Thus, a higher propagation limit yields only minor improvements. These results suggest that accesses to the shared instruction cache exhibited highly localized behavior.

The average runtime of the data-flow analysis was 0.17 seconds with the propagation limit set to 30. Decreasing the limit to 5 reduced the runtime to 0.035 seconds, whereas increasing the limit to 150 caused the analysis to require 1 s on average. When removing the propagation limit altogether, 32 of the 200 analyzed systems did not terminate after a cutoff of 4 h per core. We conclude that the propagation limit is necessary, but the analysis precision is not very sensitive to the choice of the parameter value.

8 Conclusion

In this paper, we have presented a novel analysis perspective for shared caches in multicore systems. In the analysis, the inter-core interference between tasks running on different cores is expressed using event-arrival curves. These curves quantify how much time must pass until an interfering task accesses multiple conflicting cache blocks. A core executing multiple tasks using non-preemptive scheduling can be analyzed in this framework by considering different task execution scenarios. The total interference of the scenarios is then computed by convolving the individual task curves in the max-plus algebra. This perspective enables us to view inter-task cache interference as a function of time.

To classify accesses to the shared cache as cache hits, we designed a data-flow analysis, which determines the temporal reuse distance of cache blocks stored in the shared cache. These two components, the event-arrival curves and the temporal reuse distance, allow us to derive safe cache hit classifications for individual accesses to the shared cache.

We evaluated the performance of the analysis using realistic workloads from the EEMBC AutoBench 1.1 benchmark suite. We evaluated systems containing up to 4 cores with 1, 2, and 4 tasks per core. The shared cache size ranged from 4 KB to 64 KB. Utilizing the classifications derived using the presented approach yielded better results compared to the baseline classifications.

The baseline analysis collects all potentially accessed cache blocks and assumes that these blocks may be accessed at any time, which caused the analysis to not produce any hit classifications for many systems. Compared to this baseline analysis, the presented analysis performed particularly well for systems with a small cache size relative to the total program size.

Additionally, we compared the WCET estimates with the quantitative WCIP approach presented by Nagar and Srikant (2014, 2016); Nagar (2016). We observed that the presented approach gave more precise results for all system configurations.

In the future, it would also be interesting to develop an analysis that determines precise temporal reuse values, e.g., using an IPET ILP to determine the longest duration for each potential-hit path. Such an analysis would likely not scale but it could provide valuable insights into the optimal classification of accesses. In particular, it

is of interest whether the hit ratio of large systems is limited by the precision of the analysis, or whether it is inherent to the system itself and no better classifications are possible.

We imposed no restrictions on the set of feasible scenarios $\mathcal{S}_{C'}$. This means that the results are general as all sequences of task executions on C' are considered. In the future, the properties of the chosen scheduling approach could be used to disregard infeasible scenarios. The interference curves could be tightened in this way, improving classification performance.

To reduce the required runtime of the analysis, it is conceivable to compute safe approximations of the interference curves instead of precise curves. For example, instead of computing the exact interference curves, as defined in Eq. (10), a subset of the step locations could be determined, while safely approximating the interference curve at the unknown values. This trade off between runtime and precision could be explored in future work.

As presented in this paper, the interference generated from multiple cores is computed in isolation. Considering dependencies or potential interleavings of tasks on multiple cores could increase the analysis precision. This is a topic for future work.

Another direction for future research are memory layout optimizations. Interference curves could be used to guide the placement of data objects and code in the memory layout to reduce interference and increase worst-case performance.

Funding Open Access funding enabled and organized by Projekt DEAL.

Data Availability The evaluation results used to generate the figures are available upon request from the corresponding author.

Declarations

Conflict of interest The authors have no Conflict of interest to declare that are relevant to the content of this article.

References

Aho AV, Lam MS, Sethi R et al (2007) Compilers: principles, techniques and tools, 2nd edn. Pearson Education, London

Altmeyer S, Maiza Burguière C (2011) Cache-related preemption delay via useful cache blocks: survey and redefinition. J Syst Archit 57(7):707–719. https://doi.org/10.1016/j.sysarc.2010.08.006

Fischer TL, Falk H (2023) Analysis of shared cache interference in multi-core systems using event-arrival curves. Proc Real-Time Netw Syst 23:23–33. https://doi.org/10.1145/3575757.3593643

Fischer TL, Falk H (2024) Shared cache analysis under preemptive scheduling. In: Proceedings of Design, Automation & Test in Europe Conference (DATE), pp 1–6. https://doi.org/10.23919/DATE58400.2024.10546581

Bini E, Buttazzo GC (2005) Measuring the performance of schedulability tests. Real-Time Syst 30:129–154. https://doi.org/10.1007/s11241-005-0507-9

Chattopadhyay S, Roychoudhury A (2014) Cache-Related preemption delay analysis for multilevel noninclusive caches. ACM Trans Embed Comput Syst (TECS) 13(5s):1–29. https://doi.org/10.1145/2632156

Cousot P, Cousot R (1977) Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL), pp. 238–252, https://doi.org/10.1145/512950.512973

Dharishini PPP, Murthy PVR (2021) Precise shared instruction cache analysis to estimate WCET of multi-threaded programs. In: Proc. of INDICON, pp 1–7, https://doi.org/10.1109/INDICON52576.2021.9691620

Falk H, Lokuciejewski P (2010) A compiler framework for the reduction of worst-case execution times. Real-Time Syst 46(2):251–300. https://doi.org/10.1007/s11241-010-9101-x

Ferdinand C, Wilhelm R (1999) Efficient and precise cache behavior prediction for real-time systems. Real-Time Syst 17(2):131–181. https://doi.org/10.1023/A:1008186323068

Gustafsson J, Betts A, Ermedahl A, et al (2010) The mälardalen WCET benchmarks—past, present and future. In: 10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010), pp 136–146, https://doi.org/10.4230/OASIcs.WCET.2010.136

Hardy D, Puaut I (2008) Wcet analysis of multi-level non-inclusive set-associative instruction caches. In: Proceedings of Real-Time Systems Symposium, pp 456–466, https://doi.org/10.1109/RTSS.2008.10

Hardy D, Piquet T, Puaut I (2009) Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In: Proceedings of 30th IEEE Real-Time Systems Symposium, pp 68–77, https://doi.org/10.1109/RTSS.2009.34

Kelter T (2014) Wcet analysis and optimization for multi-core real-time systems. Technische Universität Dortmund, Dortmund

Le Boudec JY, Thiran P (2001) Network calculus: a theory of deterministic queuing systems for the internet, 1st edn. Springer, Berlin

Lee CG, Hahn H, Seo YM et al (1998) Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. IEEE Trans Comput 47(6):700–713. https://doi.org/10.1109/12.689649

Li YTS, Malik S (1997) Performance analysis of embedded software using implicit path enumeration. IEEE Trans Comput-Aided Design Integr Circuits Syst 16(12):1477–1487. https://doi.org/10.1109/43.664229

Liang Y, Ding H, Mitra T et al (2012) Timing analysis of concurrent programs running on shared cache multi-cores. Real-Time Syst 48(6):638–680. https://doi.org/10.1007/s11241-012-9160-2

Lv M, Guan N, Reineke J et al (2016) A survey on static cache analysis for real-time systems. Leibniz Trans Embed Syst. https://doi.org/10.4230/LITES-v003-i001-a005

Maiza C, Rihani H, Rivas JM et al (2019) A survey of timing verification techniques for multi-core real-time systems. ACM Comput Surv (CSUR) 52(3):1–38. https://doi.org/10.1145/3323212

Nagar K (2016) Precise analysis of private and shared caches for tight WCET estimates. Indian Institute of Science Bangalore, Bengaluru

Nagar K, Srikant YN (2014) Precise Shared Cache Analysis using Optimal Interference Placement. In: IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), pp 125–134, https://doi.org/10.1109/RTAS.2014.6925996

Nagar K, Srikant YN (2016) Fast and precise worst-case interference placement for shared cache analysis. ACM Trans Embed Comput Syst. https://doi.org/10.1145/2854151

Oehlert D (2021) Worst case execution time oriented code optimization of hard real-time multicore systems. Technische Universität Hamburg, Hamburg

Oehlert D, Saidi S, Falk H (2018) Compiler-based extraction of event arrival functions for real-time systems analysis. In: 30th Euromicro Conference on Real-Time Systems (ECRTS 2018), pp 4:1–4:22, https://doi.org/10.4230/LIPIcs.ECRTS.2018.4

Poovey JA, Conte TM, Levy M et al (2009) A benchmark characterization of the EEMBC benchmark suite. IEEE Micro 29(5):18–29. https://doi.org/10.1109/MM.2009.74
Rashid SA, Nelissen G, Tovar E (2022) Tightening the crpd bound for multilevel non-inclusive caches. J Syst Archit. https://doi.org/10.1016/j.sysarc.2021.102340
Reineke J (2018) The Semantic Foundations and a Landscape of Cache-Persistence Analyses. Leibniz Trans Embed Syst. https://doi.org/10.4230/LITES-v005-i001-a003
Thiele L, Chakraborty S, Naedele M (2000) Real-time calculus for scheduling hard real-time systems. In: 2000 IEEE International Symposium on Circuits and Systems (ISCAS), pp 101–104 vol.4, https://doi.org/10.1109/ISCAS.2000.858698
Touzeau V, Maïza C, Monniaux D, et al (2019) Fast and exact analysis for LRU caches. Proceedings of the ACM on Programming Languages (POPL). https://doi.org/10.1145/3290367
Xiao J, Altmeyer S, Pimentel A (2017) Schedulability analysis of non-preemptive real-time scheduling for multicore processors with shared caches. In: 2017 IEEE Real-Time Systems Symposium (RTSS), pp 199–208, https://doi.org/10.1109/RTSS.2017.00026
Xiao J, Shen Y, Pimentel AD (2022) Cache interference-aware task partitioning for non-preemptive real-time multi-core systems. ACM Trans Embed Comput Syst (TECS) 21(3):1–28. https://doi.org/10.1145/3487581
Yan J, Zhang W (2008) WCET analysis for multi-core processors with shared L2 instruction caches. In: 2008 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp 80–89, https://doi.org/10.1109/RTAS.2008.6
Zhang W, Yan J (2009) Accurately estimating worst-case execution time for multi-core processors with shared direct-mapped instruction caches. In: 2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pp 455–463, https://doi.org/10.1109/RTCSA.2009.55
Zhang Z, Koutsoukos X (2016) Cache-related preemption delay analysis for multi-level inclusive caches. In: Proceedings of the 13th International Conference on Embedded Software (EMSOFT), https://doi.org/10.1145/2968478.2968481
Zhang W, Lv M, Chang W, et al (2022) Precise and scalable shared cache contention analysis for WCET estimation. In: Proceedings of the 59th ACM/IEEE Design Automation Conference, pp 1267–1272, https://doi.org/10.1145/3489517.3530613

Thilo L. Fischer is currently a doctoral candidate at the Institute of Embedded Systems, Hamburg University of Technology. He received his master's degree with distinction in Computer Science from the Hamburg University of Technology in 2021. His research interests include compiler-based analyses and optimizations, with a particular focus on multi-level caches for real-time systems.

Heiko Falk received the PhD degree in computer science from the University of Dortmund, Germany, in 2004. From 2004 until 2011, he worked as assistant professor in the embedded systems group at the Technical University of Dortmund. Between 2011 and 2015, he worked as full professor for embedded systems and real-time systems at Ulm University (Germany). Since 2015, he is full professor at Hamburg University of Technology (TUHH) and heads the Institute of Embedded Systems. His PhD focused on high-level source code optimizations. Typical embedded multimedia applications only use a small fraction of their execution time to compute audio or video data. Most of the execution time is used to evaluate complex control flow. Motivated by this observation, he developed novel techniques for control flow optimization at the source code level. Another focus of his work is on code generation and optimization for performance and predictability of safety-critical real-time systems. The WCC compiler initially established by him and developed by the research teams led by him is the currently only known compiler which is able to systematically reduce the worst-case execution time (WCET) of programs by tightly integrating static timing analyses into the code generation and optimization stage. He works on novel concepts to handle parallelism during real-time analysis and optimization. Mutual interferences between tasks running preemptively on the same CPU, or between tasks running on different cores and using shared resources are the key challenges of his current work. Since embedded systems regularly have to meet various additional design constraints besides real-timeliness, his current research also puts an emphasis on multi-criterial optimizations. He and his team develop novel compiler optimizations that are able to systematically trade, e.g., performance versus energy efficiency versus code size. He regularly publishes his research results at prestigious conferences and journals edited, e.g., by Springer, ACM or IEEE. He regularly serves as program committee member and reviewer for the international research community, and he was conference chair and local host of the 2023 edition of the ACM/IEEE Embedded Systems Week (ESWEEK).

Shared Cache Analysis under Preemptive Scheduling

7

The previous chapter introduced a novel analysis for shared cache interference in multi-core systems. The analysis can be applied to systems with a single task per core and non-preemptively scheduled multi-tasking systems. Preemptive scheduling, however, needs additional consideration as interfering tasks may collaborate in new ways to create higher inter-core interference. There are two major challenges that need to be addressed: 1. interfering tasks may be preempted, leading to new possible interference scenarios, and 2. after preemption, tasks may need to rebuild the L1 cache, causing increased interference in the shared cache. Furthermore, the impact on the CRPD experienced by the analyzed task has to be considered. The following publication formalizes these considerations and expands the analysis capabilities to preemptively scheduled task sets.

Chronologically, the following work was published prior to the previous chapter. For this reason, it refers to the timing-aware cache analysis in its initial version as reproduced in Appendix B, and its application to non-preemptive scheduling is not mentioned.

2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)

Submitted September 11, 2023.
Accepted November 12, 2023.
Presented in Valencia, Spain, March 25 – 27, 2024.
DOI: 10.23919/DATE58400.2024.10546581

Shared Cache Analysis under Preemptive Scheduling

Thilo L. Fischer
Hamburg University of Technology
Hamburg, Germany
thilo.leon.fischer@tuhh.de

Heiko Falk
Hamburg University of Technology
Hamburg, Germany
heiko.falk@tuhh.de

***Abstract*—When sharing a cache between multiple cores, the inter-core interference has to be considered in the worst-case execution time (WCET) analysis. Current interference models are overly pessimistic or not applicable to preemptively scheduled systems. We propose a novel technique to model interference in a preemptive system to classify accesses as cache hits or potential misses. We account for inter-core interference by considering the potential execution scenarios on the interfering core and find the worst-case interference pattern. The resulting access classifications are then used to compute the cache-related preemption delay. Our evaluation shows that the proposed analysis significantly increases the cache hit classifications, reduces WCET on average by up to 11.7%, and reduces worst-case response times on average by up to 15.4% compared to the existing classification technique.**

I. Introduction

In hard real-time systems, the timing properties of tasks are verified to ensure that no deadline will be violated. To this end, the worst-case execution time (WCET) of each task is analyzed. As the memory access latency is a major contributor to the execution time, the behavior of caches needs to be analyzed. The cache behavior can be captured by classifying accesses using cache-hit-miss-classifications (CHMC) [1]. The CHMC distinguishes between accesses that always hit, always miss, or produce uncertain behavior. Based on these access classifications, the WCET can safely be estimated.

In multi-core systems, the last-level cache is often shared between cores, which causes inter-core interference to emerge. The inter-core interference complicates the access classification as interfering cores can evict data from the shared cache. The access classification needs to account for this interference to arrive at a safe WCET estimate. The standard approach to account for inter-core interference counts the number of conflicting cache blocks from tasks running on other cores [2]. All potential conflicts are assumed to occur on each access to the shared cache — this is obviously pessimistic.

Multiple techniques to increase the analysis precision have been proposed [3]–[7]. However, all of these approaches have significant limitations. The exhaustive analysis in [3] computes precise classifications but is computationally infeasible for complex systems. [4] and [5] only compute a timing penalty bound from shared cache interference instead of classifying individual accesses. Additionally, [4] assumes that the total number of conflicting accesses is known; the impact of (non-) preemptive scheduling on the analysis is not discussed. And [5] is limited to non-preemptive scheduling as the access ordering does not account for preemption effects which increase the interference. Finally, [6] and [7] are only applicable to systems with a single task per core.

In this paper, we present a classification method for accesses to a shared cache in a preemptively scheduled multi-core system. Instead of considering all potentially conflicting cache blocks in an access classification, we determine how quickly a task creates interference at the shared cache. Timing properties were first included in the access classification by [6] and [7]; however, without taking the impact of scheduling decisions into account.

The challenge posed by a preemptively scheduled system for access classification is twofold: 1) multiple jobs with different interference profiles can interfere with a single access, and 2) a preempted job may create additional interference when resuming execution.

To solve the first challenge, we construct a regular language, which captures all possible interleavings of jobs on the interfering core. We then compute a bound on the interference of a core by combining the interference caused by the execution of individual jobs. We solve the second challenge by differentiating between previously preempted and unpreempted job executions. For previously preempted jobs, we consider the additional L1 misses due to preemption in the L2 interference computation.

The key contributions of this paper are:

- We develop a model of shared cache interference in preemptive multi-tasking systems based on the inter-arrival time of conflicting cache accesses.
- We incorporate the results of the access classification in the cache-related preemption delay (CRPD) calculation to determine the worst-case response time (WCRT).
- Our evaluation shows significant performance improvements compared to the baseline analysis [2], [8].

In Sec. II, we highlight related work. Sec. III gives an overview of the analysis. A formal language to model scheduling decisions is presented in Sec. IV. Interference curves are computed in Sec. V, while Sec. VI focuses on CRPD. Sec. VII presents the evaluation results. Sec. VIII concludes the paper.

II. Related Work

Hardy et al. [2] analyze shared cache interference by counting the number of conflicting cache blocks. All interfering blocks are considered to cause interference on each access to the shared cache. Liang et al. [9] determine which tasks may run in parallel to reduce the conflict number. Dharishini and Murthy [10] use synchronization points to bound the inter-thread interference in a multi-threaded program. Nagar [4]

presents an analysis for shared caches by determining the worst-case placement of interfering accesses in the control-flow graph. The total delay is bounded using the number of interfering accesses. Zhang et al. [5] use a *happens-before* ordering for conflicting cache accesses. This eliminates infeasible conflicts from the interference computation. The analysis computes an upper bound on the delay caused by cache interference. Xiao et al. [11] analyze the inter-core interference in a non-preemptive system. Inter-core interference is measured as the number of conflicting blocks accessed by interfering tasks. The delay due to interference is computed using the number of accesses targeting potentially evicted blocks.

In contrast to [2] and [9], the approaches [4], [5], [11] are *quantitative* analyses. Instead of classifying each access, only an upper bound on the total delay is given.

Kelter [3] explicitly considers all possible interleavings of accesses to the shared cache. This approach suffers from state space explosion and is not applicable to complex systems. Fischer and Falk [6], [7] use event-arrival curves to analyze how quickly an interfering task may evict data from the shared cache. The reuse time of cache blocks is determined and used to classify accesses as hits or potential misses. However, the analysis is limited to systems with a single task per core.

In contrast to previous research, the approach proposed in this paper is applicable to systems where multiple tasks are scheduled preemptively. Furthermore, the analysis produces a hit or potential miss classification for each access to the shared cache, which is used to determine a bound on the CRPD.

III. System Architecture and Overview

In this section, we describe the system architecture and give an overview of the proposed analysis.

The system architecture consists of multiple cores with private L1 caches. The cores are connected via a round-robin bus to the shared L2 cache. The L2 cache is set-associative with $\mathcal{A}_{L2}$ ways and uses the least-recently-used (LRU) replacement policy. The cache hierarchy is non-inclusive.

We consider a periodic task model; the period of a task τ is denoted by P_τ. Each core is assigned a fixed set of tasks T and uses fixed-priority preemptive scheduling. For tasks $\tau, \varphi \in T$ we write $\tau < \varphi$ iff φ has higher priority than τ. We assume that the scheduling overhead is negligible and does not affect the behavior of the shared cache.

The analysis presented in this paper uses the concept of event-arrival curves to model cache interference [6], [7], [12]. An interference curve expresses how much interference is caused by an interfering core over a specific time frame. Formally, it is a monotonic function $\eta : \mathbb{N}_0 \to \mathbb{N}_0$, which maps a time frame, measured in cycles, to the maximal number of distinct cache blocks that are accessed at the shared cache. This concept allows us to bound the interference based on the time between two accesses to the same cache block.

We will give a brief overview of the analysis described in [7] for systems with a single task per core. It consists of two steps: 1) deriving interference curves and 2) analyzing the cache block reuse time.

The interference curve of a task is derived from its control-flow graph (CFG). To determine the interference over a time frame of Δt cycles, paths in the CFG with a duration less or equal to Δt cycles are considered. The maximal number of accesses to distinct cache blocks is determined using an ILP based on implicit path enumeration. In the second step, potential hits, i.e., accesses to cache blocks that were previously loaded into the cache and have not been evicted by intra-task interference, are analyzed. A backward data-flow analysis determines the maximal duration from the initial access to the potential hit. This value is termed the reuse time. Evaluating the interference curve at the reuse time gives a bound on the interference. Thus, it is possible to classify the access as either a cache hit or potential miss. The approach [7] is restricted to systems with a single task per core as the interference curves are derived from the CFG of individual tasks.

In this paper, we apply the concept of interference curves to shared caches in preemptively scheduled systems. The key idea is to construct a formal language that models the execution of interfering jobs. The language describes all possible interleavings of job executions; each word corresponds to a particular sequence of job executions. Thus, each word in the language has a corresponding interference curve. We can compute the interference curve associated with a word by convolving the individual job curves in the max-plus algebra. We account for preemption effects in this step. Accesses classified as L1 cache hits may access the shared cache after a preemption. This additional interference is included in the interference curve.

The worst-case interference a cache block may experience can be determined by taking the maximum interference over all curves induced by the language. As the language contains infinitely many words, we perform a branch and bound search to determine the worst-case interference curve. We are thus able to make qualitative statements on the behavior of the cache by classifying the cache accesses using the worst-case interference curve. This allows us to safely estimate the WCET accounting for inter-core cache interference.

In a preemptively scheduled system, it is insufficient to only consider the WCET; the delay caused by preemptions has to be considered also. This includes the execution time of the preempting task but also additional context switching costs. In particular, the state of caches is modified during a preemption. Thus, additional cache misses may occur after a preemption.

The available methods [8] [13] for CRPD analysis of non-inclusive two-level caches rely on the standard classification approach [9]. Accesses considered a cache hit by the timing analysis contribute to the CRPD [14]. As the classification approach presented in this paper increases the precision of the hit classification, the additional hit classifications must be accounted for in the CRPD computation. We provide a bound on the additional CRPD in Section VI.

IV. Modelling Interference under Preemptive Scheduling

In this section, we determine how much interference can be created by a core over a particular duration. To compute the worst-case interference curve caused by a set of tasks executing

on a core, we consider all possible execution scenarios, which describe the order in which jobs are processed and how they preempt each other.

There are two types of scheduling events: 1) starting a job, and 2) finishing a job. We call the time frame between two scheduling events a *segment*. In a segment, the core processes the active job. We differentiate the state of the active job into three different categories. They are:

1) **Start:** A job which starts executing in this segment.
2) **Run:** A job which neither starts nor ends in the segment and has been preempted previously.
3) **End**: A job which ends in the segment and may have been preempted.

To represent a job of task τ in the state **Start**, **Run**, or **End**, we utilize the symbols s_τ, r_τ, e_τ respectively. The set Σ contains all possible active jobs (1).

$$\Sigma = \bigcup_{\tau \in \mathrm{T}} \{s_\tau, r_\tau, e_\tau\} \tag{1}$$

The function $C : \Sigma \to (\mathbb{N}_0 \to \mathbb{N}_0)$ maps a segment to its event curve. As noted above, the event curve for a single segment is derived from the CFG of the corresponding task. To compute the value $C(\sigma_\tau)(\Delta t), \sigma_\tau \in \Sigma$, paths of τ that require at most Δt cycles are considered. From these paths, the maximal number of cache blocks stored in the shared cache is determined. When determining the curve for a job that may have been preempted, the additional L2 interference due to the L1 misses caused by the preemption have to be considered. This can be done by considering L1 hits in the unpreempted execution to create L2 interference in the preempted execution.

We will now focus on how the execution of multiple jobs is interleaved due to fixed-priority preemptive scheduling.

Definition 1. *An execution scenario is a sequence of letters from the set* Σ.

As an example, consider the following word over Σ for $T = \{1, 2, 3\}$: $s_1 s_2 e_2 r_1 s_3 e_3 e_1$. The interpretation of the word is as follows: First task 1 starts (s_1); it is preempted by task 2 (s_2); after task 2 finishes (e_2), task 1 resumes execution (r_1); task 1 is preempted again, this time by task 3 (s_3); after task 3 ends (e_3), task 1 finishes (e_1).

Furthermore, we define a *valid scenario* as a scenario that is conforms to the notion of fixed-priority scheduling.

Definition 2. *An execution scenario is valid if it satisfies the following conditions:*

1) A job may only be preempted by a job of higher priority.
2) After a job has finished, control is passed to the unfinished job with the highest priority. If there is no unfinished job, any task may spawn a new instance.

Based on these conditions we can model scheduling decisions in a formal language.

Theorem 1. *The set of valid execution scenarios forms a regular language* L.

We formulate a regular grammar (V, Σ, P, I) for L to prove Theorem 1 by construction. V denotes the set of non-terminal variables, Σ is the alphabet, P are the production rules, and I is the initial non-terminal. The state of the core is abstracted to the active job and the preempted jobs (2).

$$\mathcal{TS} = \{(\tau_{i_1}, \ldots, \tau_{i_n}) \mid j < k \implies \tau_{i_j} < \tau_{i_k}\} \tag{2}$$

$\mathcal{TS}$ contains all sequences of tasks ordered by ascending priority. The interpretation of a sequence $(\tau, \varphi) \in \mathcal{TS}$ is that a job of τ was running and got preempted by a job of φ. The last task in the sequence indicates the currently active job, in this example a job of task φ. The set of non-terminal variables V contains the initial non-terminal I and is further induced by the set $\mathcal{TS}$ and the category of the last scheduling event (3).

$$V = \{I\} \cup \bigcup_{ts \in \mathcal{TS}} \{S_{ts}, R_{ts}, E_{ts}\} \tag{3}$$

The non-terminals S_{ts} signify that the last scheduling event was the start of a new job. Similarly, the non-terminals R_{ts} show that the system currently processes a **Run** job, while E_{ts} shows that a job was just finished. Using these non-terminals, we formulate regular production rules P in (4).

$$\forall(\ldots, \tau) \in \mathcal{TS}: \; I \to s_\tau S_{(\ldots,\tau)} \mid r_\tau R_{(\ldots,\tau)} \mid e_\tau E_{(\ldots)} \tag{4a}$$
$$\forall \varphi > \tau: \; R_{(\ldots,\tau)} \to s_\varphi S_{(\ldots,\tau,\varphi)} \tag{4b}$$
$$\forall \varphi > \tau: \; S_{(\ldots,\tau)} \to s_\varphi S_{(\ldots,\tau,\varphi)} \tag{4c}$$
$$\forall(\ldots, \tau) \in \mathcal{TS}: \; S_{(\ldots,\tau)} \to e_\tau E_{(\ldots)} \tag{4d}$$
$$\forall(\ldots, \tau) \in \mathcal{TS}: \; E_{(\ldots,\tau)} \to r_\tau R_{(\ldots,\tau)} \mid e_\tau E_{(\ldots)} \tag{4e}$$
$$\forall \tau \in T: \; E_{()} \to s_\tau S_{(\tau)} \tag{4f}$$
$$\forall ts \in \mathcal{TS}: \; S_{ts} \to \varepsilon, \; R_{ts} \to \varepsilon, \; E_{ts} \to \varepsilon \tag{4g}$$

The initial non-terminal I is transformed into an active job τ in one of the three states (4a). Note that the rule does not restrict the initial state of the preempted jobs. All possible scenarios in $(\ldots, \tau) \in \mathcal{TS}$ are considered because we are interested in the scheduling decisions that occur between two usages of a cache block, which happen at an arbitrary point in time. The rule (4b) covers the scenario that a job τ in the category of **Run** is preempted. The start symbol s_φ for the new job φ is appended and the stack of running jobs is extended to $(\ldots, \tau, \varphi)$. By definition, a non-terminal $R_{(\ldots,\tau)}$ can only lead to a preemption and not to the ending of the current job. After starting a new job, there are two possibilities for the next event. Either, the newly started job gets preempted (4c), or it will finish its execution without preemption (4d). In the latter case, φ is removed from the active task stack. The rules (4e) and (4f) are concerned with the scheduling events that happen after a job has finished executing. Rule (4e) covers the situation where a preempted task τ resumes execution after a preemption. It may either run until it is preempted again or finish without being preempted again. (4f) covers the case that the last active job has finished executing. The only action that is possible in this situation is to start a new job. The final rule (4g) allows all non-terminals to be converted to the empty word ε to end the derivation.

Consider a task set $T = \{1, 2, 3\}$, listed in ascending priority. The scenario of a preemption of 1 by 3 immediately followed by starting 2, i.e., $r_1 s_3 e_3 s_2$, is not directly included in L.

However, the behavior is equivalently covered by $r_1 s_3 e_3 r_1 s_2$, as the second r_1 segment may be considered to last for 0 cycles.

Thus, we have created a regular grammar (V, Σ, P, I), which generates the scheduling language L and have proven Theorem 1 by construction. The words in L describe all execution scenarios which may occur on an interfering core. This language is used in the next section to examine the emergent interference behavior at the shared cache. Each scenario $l \in L$ has a different interference pattern on the shared cache. As we want to make hit classifications that hold even under the highest level of interference, we have to consider the worst-case interference generated from all scenarios.

V. Computing Interference Curves

In this section, we will compute the interference curve of a single execution scenario and find the scenario causing the worst-case interference. Let F represent a mapping from a scenario to the induced curve (5).

$$F : L \to (\mathbb{N}_0 \to \mathbb{N}_0^-), \text{ where } \mathbb{N}_0^- = \mathbb{N}_0 \cup \{-\infty\} \tag{5}$$

To properly define the function F, we first need to construct the basic components. Event-arrival curves are computed using the *max-plus* algebra [15]. In the max-plus algebra, $-\infty$ is the absorbing element, i.e., $\forall n \in \mathbb{N}_0 : -\infty + n = -\infty$. We use the value $-\infty$ to signalize infeasible situations. We write the max-plus convolution of η_1 and η_2 as the $\circledast$ operator (6).

$$(\eta_1 \circledast \eta_2)(\Delta t) = \max_{0 \le \delta \le \Delta t} \{\eta_1(\delta) + \eta_2(\Delta t - \delta)\} \tag{6}$$

The function W_t either maps to 0 or $-\infty$ depending on the parameter and applies a *window* to an interference curve (7).

$$W_t(\Delta t) = \begin{cases} 0 & \text{if } \Delta t \ge t \\ -\infty & \text{else} \end{cases} \tag{7}$$

Consider $((\eta_1 \circledast \eta_2) + W_{100})(\Delta t)$ as an example. The window W_{100} is added to the convolution of η_1 and η_2. The resulting curve thus yields $-\infty$ for $\Delta t < 100$. By applying a window to a curve, it can be marked as infeasible for short durations. Thus, applying W_t refines the interference computation for a scenario by enforcing a minimal time between scheduling events.

Each scenario has a minimal duration to be feasible. For shorter durations, it does not contribute to the interference calculation. Let $minDur(\sigma_1 \dots \sigma_n)$ be the minimal duration of a scenario. We give two lower bounds for the minimal duration.

For the first bound, we introduce the helper function $\omega : \Sigma^2 \to \mathbb{N}_0$. It gives a lower bound for the time that needs to pass between scheduling events (8).

$$\omega(a, b) = \begin{cases} BCET(\tau) & \text{if } a = s_\tau, b = e_\tau \\ 0 & \text{else} \end{cases} \tag{8}$$

Eq. (8) states that the minimal time between starting a job and finishing that job is given by the best-case execution time (BCET). Thus, the minimal duration of a scenario is bounded by the time required for all complete job executions (9).

$$minDur(\sigma_1 \dots \sigma_n) \ge \sum_{1 \le i < n} \omega(\sigma_i, \sigma_{i+1}) \tag{9}$$

We create a second bound on the minimal duration by considering the number of jobs that have started for each task (10). Let $\#\tau$ denote the number of starts of τ in the considered scenario $\sigma_1 \dots \sigma_n$. The value $P_\tau \cdot (\#\tau - 2)$ is a lower bound on the time required to activate the task $\#\tau$ times.

$$minDur(\sigma_1 \dots \sigma_n) \ge \max_{\tau \in T} \{P_\tau \cdot (\#\tau - 2) \mid \#\tau > 2\} \tag{10}$$

Using these building blocks, we define F in (11). The function $G_{(\sigma_1 \dots \sigma_n)}$ recursively convolves the individual curves $C(\sigma_m)$ from segments contained in the scenario and applies a window to the result. The window parameter is the larger value from (9) and (10) for $\sigma_1 \dots \sigma_m$.

$$F(\sigma_1 \dots \sigma_n) = G^n_{(\sigma_1 \dots \sigma_n)} \tag{11a}$$

$$G^0_{(\sigma_1 \dots \sigma_n)} = 0 \tag{11b}$$

$$G^m_{(\sigma_1 \dots \sigma_n)} = (G^{m-1}_{(\sigma_1 \dots \sigma_n)} \circledast C(\sigma_m)) + W_{minDur(\sigma_1 \dots \sigma_m)} \tag{11c}$$

The maximal interference, denoted by η^*, is then given as the maximal interference over all scenarios (12).

$$\eta^*(\Delta t) = \max_{l \in L} \{F(l)(\Delta t)\} \tag{12}$$

We are now able to compute an upper limit on the inter-core interference. However, the search space of scenarios in L is infinitely large. To explore it efficiently, we perform a branch and bound search. For this purpose, the scenarios are organized in a prefix tree. Each node represents a scenario $l \in L$; edges correspond to a single derivation step (4); child nodes are scenarios that possess l as a prefix. This is possible as L is a regular language.

We can terminate the exploration at a node with scenario $l \in L$ if its minimal duration exceeds the time required to evict all data from the cache. The scenario will not contribute to the maximal interference curve, thus it is safe to discard it and (by monotonicity of $minDur(\cdot)$) any scenario that possesses l as a prefix.

Note that the language L is an over-approximation of the feasible job sequences as it is constructed independently of the task periods. By considering the period of each task, we can identify infeasible scenarios. A feasible scenario has to satisfy the constraint (13). Let φ^* be the highest priority task.

$$\#\tau \le \left\lceil \frac{(\#\varphi^* + 1) \cdot P_{\varphi^*}}{P_\tau} \right\rceil \tag{13}$$

As φ^* preempts all other tasks and is released on a periodic schedule, $(\#\varphi^* + 1) \cdot P_{\varphi^*}$ is an upper bound on the execution time of the scenario. The maximal number of activations of tasks $\tau \neq \varphi^*$ in a feasible scenario is thus limited by (13).

Using η^*, an individual access to the shared cache can now be classified as a cache hit or potential miss. The worst-case inter-core interference experienced by the cache block is limited by $\eta^*(t_{max})$, where t_{max} is the maximal duration since the last access to the cache block. The access will result in a cache hit if the resilience of the cache block is larger than the inter-core interference. Multiple interfering cores can be modeled by adding the interference value of the individual cores.

VI. Cache-related Preemption Delay

In the previous section, cache accesses are categorized into cache hits and potential misses using the interference curve η^*. The classifications can be used to compute a task's WCET. However, in a preemptively scheduled system, the CRPD needs to be considered to determine the WCRT. In this section, we will discuss how the CRPD can be computed based on the presented hit classification approach.

A preemption impacts the cache behavior in two fundamental ways: there are direct and indirect preemption effects [8]. The direct effect occurs due to cache blocks aging (and being evicted) by the preempting task; the indirect effect occurs due to the increased intra-task interference after a preemption. A cache access resulting in an L1 hit without preemptions can result in an L1 miss due to the direct preemption effects. The access thus gets forwarded to the L2 cache only if a preemption has occurred previously. The additional access to the L2 can cause further evictions and lead to an even higher delay; this is the indirect effect.

Chattopadhyay and Roychoudhury [8] developed a method to compute the CRPD for non-inclusive cache hierarchies using the standard access classification method [2]. It is unsafe to use this CRPD value for the more precise hit classification we have presented in this paper. We will now discuss the necessary extension to compute a safe CRPD value.

The key difference between the standard approach and the one presented in this paper is that timing properties of paths are considered in the classification. This fundamentally challenges a basic assumption made in many cache analyses: Cache sets are assumed to operate independently of each other. This is no longer the case when the classification process is refined to include time. The timing of an access targeting a cache set influences the classification of an access to a different cache set. To safely bound the WCRT, we have to consider this fact in the CRPD computation. In addition to the CRPD value computed by [8], we account for the penalty due to timing effects by assuming that every block reuse path affected by direct or indirect preemption effects will degrade to a cache miss.

From the classifying data-flow analysis (DFA) in [7], we can determine how many hit classifications may be impacted by preemption effects at each program location p from the set of all program locations $\mathbb{P}$ (14). Let $CHMC$ ($CHMC_{STD}$) denote the classification of the presented (standard) approach.

$$LHP(p) = \left\{ acc \;\middle|\; \begin{array}{l} CHMC(acc) = Hit \wedge \\ DFA_{in}[p](acc) \neq \bot \wedge \\ CHMC_{STD}(acc) \neq Hit \end{array} \right\} \tag{14}$$

$LHP(p)$ is the set of all accesses that may degrade to a cache miss due to preemption effects for location p. The condition $CHMC(acc) = Hit$ states that the access is classified as a L2 hit. The condition $DFA_{in}[p](acc) \neq \bot$ means that the targeted cache block will not be refreshed prior to its use by acc [7]. A preemption effect occurring at p may thus cause the access to miss. We do not need to consider accesses classified as a hit by the standard method, as their contribution to the CRPD is already accounted for in [8].

A bound on the number of paths affected per preemption induced cache miss is given in (15). We take the maximum number of live hit paths $LHP(p)$ over program points $p \in \mathbb{P}$.

$$LHP_{max} = \max_{p \in \mathbb{P}} |LHP(p)| \tag{15}$$

We adopt the notation of [8] to group accesses that are susceptible to preemptions. The sets $\mathbb{M}_1(p)$ and $\mathbb{M}_2(p)$ contain accesses that are L1 hits in the absence of preemption but may miss due to a preemption for a program location p. The preemption itself and all accesses contained in $\mathbb{M}_1$ and $\mathbb{M}_2$ disturb data in the shared cache. Thus, the value $\mathbb{M}_{max} = 1 + \max_{p \in \mathbb{P}}\{|\mathbb{M}_1(p)| + |\mathbb{M}_2(p)|\}$ is an upper bound on the number of disturbances.

Consider three accesses a, b and, c where a is contained in the reuse path leading to b and the usage of b is contained in the reuse path for c. Suppose a preemption causes a to miss. The reuse duration of b is extended and may lead to a miss for b, which leads to a miss for c. This pattern can continue for an arbitrary number of accesses. To prevent such a scenario from happening, we account for the worst-case duration (due to a miss from a) in the path analysis for b. Thus, it is not necessary to account for L2 hits that are degraded to L2 misses due to a preemption, as we can account for such misses in the worst-case path duration.

The additional CRPD due to more precise access classifications is given by (16), where $Miss_{L2}$ is the L2 miss penalty.

$$CRPD_{add} = \mathbb{M}_{max} \cdot LHP_{max} \cdot Miss_{L2} \tag{16}$$

The total CRPD value is then given by the value computed by [8] plus the additional penalty $CRPD_{add}$.

VII. Evaluation

We evaluated the performance of the presented approach by implementing it in a WCET analyzer [16]. The systems consist of two ARM7TDMI cores with private L1 caches connected via a round-robin bus to a shared L2 cache. The caches utilize the LRU replacement policy; L1 caches are direct-mapped containing 256 bytes; the shared cache is set-associative with 8 ways. A cache block contains 64 bytes. We evaluated shared cache sizes from 4 KB to 32 KB. The L1-Hit / L2-Hit / L2-Miss timings are 1 / 10 / 40 cycles. We focused our evaluation on instruction caches, as the memory layout of the code is known at compile time. This is not a limitation of the analysis method. Workloads are taken from the EEMBC AutoBench 1.1 suite [17]. We randomly generated 20 systems with 2 tasks per core and 20 systems with 4 tasks per core. Task periods were generated using UUnifast for a target utilization of 0.7 per core. Priorities are assigned according to rate-monotonic scheduling. The analyses were performed on an Intel Xeon server containing 48 cores at 3.2 GHz. Each analysis used only a single processor core. We evaluated three metrics: the percentage of hit classifications, the WCET, and the WCRT.

Fig. 1 and Fig. 2 show the percentage of accesses classified as an L2 cache hit for 2 and 4 tasks, respectively. The presented analysis is shown in blue; the standard analysis is shown in orange. Almost always, no hit classification was made by the

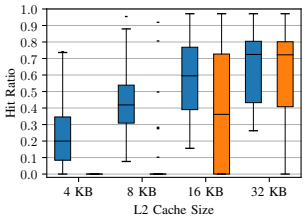


Fig. 1. Hit ratio. 2 tasks per core. The proposed analysis is in blue; the standard analysis in orange

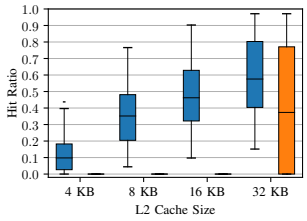


Fig. 2. Hit ratio. 4 tasks per core. The proposed analysis is in blue; the standard analysis in orange

TABLE I
AVERAGE ANALYSIS RESULTS

Tasks	Cache	WCET Red.	WCRT Red.	Time Overhead
2	4 KB	5.1%	6.3%	7.6×
	8 KB	9.6%	12.0%	8.1×
	16 KB	5.3%	6.6%	7.0×
	32 KB	1.5%	1.9%	8.0×
4	4 KB	2.2%	3.4%	7.8×
	8 KB	8.5%	11.3%	9.0×
	16 KB	11.7%	15.4%	7.4×
	32 KB	5.4%	7.5%	7.9×

standard analysis for cache sizes 4 KB – 8 KB for 2 tasks per core. And no hit classification was made for 4 KB – 16 KB for 4 tasks per core. This is due to the code size of the interfering tasks; on average 14 KB (27 KB) from the 2 (4) interfering tasks. As all potential conflicts are considered for every access, the standard analysis is almost always unable to make hit classifications in these configurations. For cache sizes that approach or exceed the code size of the interfering core, the gap between the two classification approaches shrinks.

The presented analysis increases the average hit ratio by 22.9 / 40.6 / 19.6 / 6.4 percentage points (pp) for two tasks per core and cache sizes 4 KB / 8 KB / 16 KB / 32 KB; for four tasks per core, it is increased by 11.1 / 35.6 / 47.5 / 22.1 pp. Table I shows the average analysis results for the different task set and cache sizes. The column WCET reduction shows the decrease of the WCET compared to the standard analysis. The average WCET reduction ranges from 1.5% to 9.6% for two tasks per core, with the largest reduction at the 8 KB cache configuration. For larger caches, the difference between the hit classifications reduces, which results in a smaller WCET improvement. The lowest average reduction occurred at the largest cache configuration. This cache is large enough to fit most of the code from interfering tasks and the analyzed task. For four tasks per core, WCET reduction ranges from 2.2% to 11.7%. The highest reduction was measured for the 16 KB cache. The WCRT reduction follows the same trend as the WCET reduction, while its magnitude is consistently higher. The highest average WCRT reduction is 12.0% for two tasks per core, and 15.4% for four tasks per core.

The average analysis runtime for a complete system varies with the cache size and task set size. It took on average 6.3–10.6 minutes for two tasks per core and 16.1–30.8 minutes for 4 tasks. The overhead compared to the standard approach of simply counting the number of conflicting blocks is shown in the last column of Table I. The overhead ranges between $7.0\times$–$8.1\times$ ($7.4\times$–$9.0\times$) for 2 (4) tasks. Computing the interference curves $C(\cdot)$ scaled linearly in the number of tasks. Although $\mathcal{TS}$ grows exponentially in the number of tasks, we did not observe an increase in the time to compute the interference curves for 4 tasks compared to 2 tasks per core. Both of these components are highly parallelizable, which can be leveraged to reduce the runtime.

VIII. Conclusion

We have presented a hit classification for shared caches in a preemptively scheduled multi-core systems. By modeling the scheduling decisions in a regular language, we can compute an event curve for inter-core interference. Additionally, we have integrated the hit classifications in the CRPD computation to determine the WCRT of a task. Our evaluation showed significant improvements of the cache hit classification percentage, WCET, and WCRT. In the future the scalability could be improved and a tighter bound on the CRPD could be developed.

References

[1] D. Hardy and I. Puaut, "WCET analysis of multi-level non-inclusive set-associative instruction caches," in *Proc. of RTSS*, 2008, pp. 456–466.
[2] D. Hardy, T. Piquet, and I. Puaut, "Using Bypass to Tighten WCET Estimates for Multi-Core Processors with Shared Instruction Caches," in *Proc. of RTSS*, 2009, pp. 68–77.
[3] T. Kelter, "WCET Analysis and Optimization for Multi-Core Real-Time Systems," Ph.D. dissertation, Technische Universität Dortmund, 2014.
[4] K. Nagar, "Precise analysis of Private and Shared Caches for tight WCET Estimates," Ph.D. dissertation, Indian Institute of Science Bangalore, 2016.
[5] W. Zhang, M. Lv, W. Chang, and L. Ju, "Precise and Scalable Shared Cache Contention Analysis for WCET Estimation," in *Proc. of DAC*, 2022, pp. 1267–1272.
[6] T. L. Fischer and H. Falk, "WCET Analysis of Shared Caches in Multi-Core Architectures using Event-Arrival Curves," in *Proc. of DATE*, 2023, pp. 1–2.
[7] ——, "Analysis of Shared Cache Interference in Multi-Core Systems using Event-Arrival Curves," in *Proc. of RTNS*, 2023, p. 23–33.
[8] S. Chattopadhyay and A. Roychoudhury, "Cache-Related Preemption Delay Analysis for Multilevel Noninclusive Caches," *ACM TECS*, vol. 13, no. 5s, pp. 1–29, 2014.
[9] Y. Liang, H. Ding, T. Mitra, A. Roychoudhury, Y. Li, and V. Suhendra, "Timing Analysis of Concurrent Programs Running on Shared Cache Multi-Cores," *Real-Time Systems*, vol. 48, no. 6, pp. 638–680, 2012.
[10] P. P. P. Dharishini and P. V. R. Murthy, "Precise Shared Instruction Cache Analysis to Estimate WCET of Multi-threaded Programs," in *Proc. of INDICON*, 2021, pp. 1–7.
[11] J. Xiao, Y. Shen, and A. D. Pimentel, "Cache Interference-aware Task Partitioning for Non-preemptive Real-time Multi-core Systems," *ACM TECS*, vol. 21, no. 3, pp. 1–28, 2022.
[12] D. Oehlert, S. Saidi, and H. Falk, "Compiler-Based Extraction of Event Arrival Functions for Real-Time Systems Analysis," in *Proc. of ECRTS*, 2018, pp. 4:1–4:22.
[13] S. A. Rashid, G. Nelissen, and E. Tovar, "Tightening the CRPD bound for multilevel non-inclusive caches," *Journal of Systems Architecture*, vol. 122, 2022.
[14] S. Altmeyer, "Analysis of Preemptively Scheduled Hard Real-time Systems," Ph.D. dissertation, Universität des Saarlandes, 2012.
[15] J. Liebeherr, "Duality of the Max-Plus and Min-Plus Network Calculus," *Foundations and Trends® in Networking*, vol. 11, no. 3-4, pp. 139–282, 2017.
[16] H. Falk and P. Lokuciejewski, "A Compiler Framework for the Reduction of Worst-Case Execution Times," *Real-Time Systems*, vol. 46, no. 2, pp. 251–300, 2010.
[17] J. A. Poovey, T. M. Conte, M. Levy, and S. Gal-On, "A Benchmark Characterization of the EEMBC Benchmark Suite," *IEEE Micro*, vol. 29, no. 5, pp. 18–29, 2009.

8 Towards Analysing Cache-Related Preemption Delay in Non-Inclusive Cache Hierarchies

While designing the shared cache analysis for preemptively scheduled systems, it became apparent that literature on cache-related preemption delay for multi-level caches is limited. At that time only three publications on that topic existed [CR14, ZK16, RNT22]. However, modern system architectures typically feature multiple cache levels. This gap between the state-of-the-art in CRPD analysis and modern hardware limits the application these architectures in real-time systems because their timing behavior cannot be verified. In this chapter, we address this gap in the capabilities of static timing analysis.

ACM Transactions on Embedded Computing Systems

Manuscript submitted January 31, 2024.
Revised July 3, 2024.
Accepted September 1, 2024.
Published online October 5, 2024.
Journal Volume 24, Issue 1, January 2025.
DOI: 10.1145/3695768

Towards Analysing Cache-Related Preemption Delay in Non-Inclusive Cache Hierarchies

THILO LEON FISCHER, Institute of Embedded Systems, Hamburg University of Technology, Hamburg, Germany
HEIKO FALK, Institute of Embedded Systems, Hamburg University of Technology, Hamburg, Germany

The impact of preemptions has to be considered when determining the schedulability of a task set in a preemptively scheduled system. In particular, the contents of caches can be disturbed by a preemption, thus creating context-switching costs. These context-switching costs occur when a preempted task needs to reload data from memory after a preemption. The additional delay created by this effect is termed *cache-related preemption delay* (CRPD). The analysis of CRPD has been extensively studied for single-level caches in the past. However, for two-level caches, the analysis of CRPD is still an emerging area of research. In contrast to a single-level cache, which is only affected by direct preemption effects, the second-level cache in a two-level hierarchy can be subject to *indirect interference* after a preemption. Accesses that could be served from the L1 cache in the absence of preemptions, may be forwarded to the L2 cache, as the relevant data was evicted by a preemption. These accesses create the *indirect interference* in the L2 cache and can cause further evictions. Recently, a CRPD analysis for two-level non-inclusive cache hierarchies was proposed. In this article, we show that this state-of-the-art analysis is unsafe as it potentially underestimates the CRPD. Furthermore, we show that the analysis is pessimistic and can overestimate the indirect preemption effects. To address these issues, we propose a novel analysis approach for the CRPD in a two-level non-inclusive cache hierarchy. We prove the correctness of the presented approach based on the set of feasible program execution traces. We implemented the presented approach in a *worst-case execution time* (WCET) analysis tool and compared the performance to existing analysis methods. Our evaluation shows that the presented analysis increases task set schedulability by up to 14 percentage points compared with the state-of-the-art analysis.

CCS Concepts: • **Computer systems organization** → **Real-time systems**; *Embedded software;*

Additional Key Words and Phrases: WCET analysis, multi-level caches, context-switching costs, cache-related preemption delay, preemptive scheduling

ACM Reference Format:
Thilo Leon Fischer and Heiko Falk. 2024. Towards Analysing Cache-Related Preemption Delay in Non-Inclusive Cache Hierarchies. *ACM Trans. Embedd. Comput. Syst.* 24, 1, Article 8 (October 2024), 37 pages. https://doi.org/10.1145/3695768

1 Introduction

In hard real-time systems, every task has an associated deadline. For the system to operate properly, not only the functional correctness, i.e., the correct result of a computation, but also the timeliness

Authors' Contact Information: Thilo Leon Fischer, Institute of Embedded Systems, Hamburg University of Technology, Hamburg, Germany; e-mail: thilo.leon.fischer@tuhh.de; Heiko Falk, Institute of Embedded Systems, Hamburg University of Technology, Hamburg, Germany; e-mail: Heiko.Falk@tuhh.de.

ACM 1539-9087/2024/10-ART8
https://doi.org/10.1145/3695768

of the computation has to be ensured. To verify that a system will keep all deadlines even in the worst-case scenario, it is necessary to provide an upper bound on the execution time of each task. This upper bound is called the ***worst-case execution time*** (**WCET**). An estimate for the WCET can be computed by analysing the **control-flow graph** (**CFG**) of a task and determining the path that maximizes the execution time [25].

The WCET is, however, not sufficient to decide whether all instances of a task, called jobs, will finish in time, if preemptions occur. During a preemption, the execution of a job is paused in favor of processing another job. A preemption does not only delay the execution of the preempted job but also causes further context-switching costs for the preempted job after resuming execution. This additional delay arises as the preemption may change the state of the hardware. In particular, the state of the caches can be modified by the preempting job; evicting data that is required by the preempted job. When resuming the execution of the preempted job, this data has to be reloaded into the caches, which causes the additional delay. This delay is named ***cache-related preemption delay*** (**CRPD**) [24]. The problem of CRPD has been extensively studied for single-level caches in the past [2, 5, 24, 31].

The CRPD analysis for multi-level caches is fundamentally more challenging than for a single-level cache. This is due to the so-called *indirect effect of preemptions* in a multi-level cache hierarchy [10]. The indirect effect describes the increased intra-task interference in the L2 cache after a preemption has happened. A preemption may evict useful data from the L1 cache. Thus, a cache block that would have been reused from the L1 cache without any preemptions has to be reloaded from the L2 cache due to the preemption. This additional access to the L2 cache creates intra-task interference and can potentially evict data from the L2 cache, which in turn causes further delays. Capturing the aging effects in the L2 cache caused by the indirect effect is crucial to determine a safe bound on the CRPD. We call the indirect effect of preemption *indirect interference* in this article.

Only few approaches have been proposed for the problem of CRPD in multi-level cache hierarchies. Chattopadhyay and Roychoudhury created the first analysis considering the indirect preemption effects for non-inclusive two-level cache hierarchies [10]. Two years later, Zhang and Koutsoukos analyzed CRPD in inclusive cache hierarchies [47]. Recently, Rashid et al. proposed an improvement for the analysis of non-inclusive caches [36]. In their work, they expanded the concept of **useful cache blocks** (**UCB**) to two-level caches by differentiating between L1-useful and L2-useful blocks. To the best of our knowledge, no further improvements for non-inclusive cache hierarchies have been proposed, which is why we refer to [36] as the state-of-the-art for non-inclusive cache hierarchies in the rest of the article.

In this article, we demonstrate several shortcomings of the state-of-the-art approach. We show multiple issues that may lead to the underestimation of the CRPD and a source of overestimation that causes pessimism in the analysis. To fix these issues, we present a novel analysis approach for the CRPD in a two-level non-inclusive instruction cache hierarchy. We prove the correctness of the presented analysis at the level of task execution traces, i.e., sequences of program locations and concrete system states. We integrated the presented approach in a WCET-aware compiler [13] and evaluated its performance for different task sets and system configurations. Our evaluations show that the presented approach improves the task set schedulability by up to 14 percentage points over the previous state-of-the-art.

Over the past decade, the state-of-the-art WCET research has been lagging behind the developments of commercial processor architectures. For this reason, a gap has developed between the capabilities of static worst-case analysis methods and **commercial-of-the-shelf** (**COTS**) processors.

We consider this article to be a first step towards enabling the analysis of modern hardware architectures featuring multi-level caches. Further research is needed to close the gap between

static WCET analysis methods and modern processor architectures. This is the case as the presented analysis poses strict requirements on the hardware, such as separate instruction and data caches as well as **least-recently-used** (**LRU**) replacement in both cache levels. These restrictions need to be addressed in future work.

The main contributions of this article are:

— Definition of the concrete semantics of CRPD in a two-level non-inclusive cache hierarchy to prove the correctness of the presented analysis.
— A novel analysis approach for CRPD in non-inclusive cache hierarchies, which fixes unsafe estimations and eliminates pessimism present in the state-of-the-art analysis.
— An evaluation showing significant improvements in task set schedulability using the presented CRPD analysis.

The rest of this article is structured as follows: Related work is discussed in Section 2. In Section 3, we discuss the assumptions and requirements on the system and the analyzed tasks. We present the background and state-of-the-art in Section 4. The pessimism and unsafe formulations of the state-of-the-art analysis are demonstrated in Section 5. In Section 6, we introduce the concrete semantics for a two-level non-inclusive cache hierarchy to precisely define the required concepts. We analyze the CRPD stemming from accesses that result in an L1 hit without preemptions in Section 7. In Section 8, the CRPD from accesses considered L2 hits without preemptions is analyzed. We evaluate the performance of the presented approach in Section 9. Finally, we conclude the article in Section 10.

2 Related Work

Static analysis of cache behavior is an active research field. For over two decades, many different facets of cache behavior and cache-related delays have been analyzed.

In Reference [14], Ferdinand and Wilhelm presented *may* and *must* analyses. The concrete states of the system are abstracted using *abstract interpretation* [11] to the minimal and maximal age of a cache block. Twenty years later, Touzeau et al. [41] presented an abstraction based on zero-suppressed binary decision diagrams that allows for a precise analysis of may/must information.

The may and must analyses are instances of *qualitative* analyses. Each individual access to the cache can be analyzed and classified as either a hit, a miss, or unknown. In contrast, *quantitative* analyses provide an upper bound on the total delay due to some cache behavior. The *persistence* analysis [14, 38] is a quantitative analysis that determines whether a set of accesses can result in at most one cache miss. Stock et al. [40] presented a precise persistence analysis.

If a cache is shared between multiple cores, the interference between different cores has to be considered. Hardy and Puaut [21], as well as Liang et al. [26] analyzed the interference between tasks in a shared cache by counting the number of potential conflicting cache blocks. Nagar [33] determined the worst-case placement of interfering accesses in the CFG for a quantitative analysis of shared cache interference. Zhang et al. [46] eliminated infeasible access combinations in the interference computation by considering the order of accesses. Fischer and Falk [15] presented a qualitative analysis for shared caches by bounding the time between conflicting accesses. Xiao et al. [45] performed a schedulability analysis in a non-preemptive system with a shared cache.

CRPD occurs when multiple tasks are assigned to a single core and preemptions are enabled. Busquets-Mataix et al. [9] showed how the **worst-case response time** (**WCRT**) of a task can be determined in the presence of preemption delays. Lee et al. [24] analyzed the CRPD based on the number of UCBs. Altmeyer and Burguiere [3, 5] refined the definition of a UCB. A block is only considered to be useful if it is definitely cached, as otherwise the timing analysis will already

account for the potential cache miss. Altmeyer et al. [4] introduced the concept of *resilience* to analyze set-associative LRU caches.

Cache blocks may persist in the cache between two executions of the same task. This fact can be used to reduce the WCRT estimate. Rashid et al. [35] analyzed *persistence reload* overheads that occur when persistent blocks are evicted by a preempting task.

The CRPD analyses listed above have focused on architectures with a single level cache. The analysis of CRPD in two-level caches is still an emerging area of research. Chattopadhyay and Roychoudhury [10] presented the first CRPD analysis for a two-level cache hierarchy, focusing on non-inclusive cache hierarchies.

Zhang and Koutsoukos [47] developed an analysis for inclusive cache hierarchies.

The work of Rashid et al. [36] introduced the concept of L1-UCBs and L2-UCBs and represents the state-of-the-art for CRPD analysis of non-inclusive two-level caches. The analysis poses an additional requirement, compared with Reference [10]: the L2 cache needs a higher or equal number of cache sets and associativity than the L1 cache. While [36] reports increased analysis precision compared with Reference [10], we will show in this article that the analysis contains flaws, which may cause underestimation of the actual preemption penalty.

All three approaches for multi-level CRPD analysis assume LRU replacement policy at both cache levels and instruction-only caches. Furthermore, the accesses which will be issued by the analyzed tasks must be known by the analysis. Otherwise, it is impossible to determine the possible cache states and the resulting preemption delays. This requires the architecture to have predictable fetching behavior.

The architecture requirements of the presented approach are identical to the requirements of the state-of-the-art analysis [36]. We discuss these requirements and the reasoning behind them in the following section.

3 System Model

The architectural and task model used in the presented analysis are discussed in this section. First, in Section 3.1, we focus on the hardware requirements and their impact on the applicability of the analysis to commercial processors. The properties of the executed tasks are treated in Section 3.2.

3.1 Architectural Model

We describe the target architecture to which the presented analysis can be applied in Section 3.1.1 and discuss the applicability to commercial processors in Section 3.1.2.

3.1.1 Target Architecture. We analyze an architecture that consists of a single core with a two-level instruction cache hierarchy. When processing a memory access, the L1 cache is always accessed; the L2 cache is only accessed if the requested information is not contained in the L1 cache. Both cache levels utilize the LRU replacement policy, as it offers high predictability [39] and is recommended for real-time systems in Reference [43].

As is the case in previous work on multi-level CRPD analysis [10, 36, 47], we focus on instruction caches in this article. We do not model interference from data accesses in the caches, as data accesses frequently target uncertain addresses. Thus, it is required that the L1 and L2 caches are instruction only caches, or that the caches are partitioned in such a way that the data accesses do not interfere with instructions stored in the caches.

The mapping from instructions to the corresponding cache set needs to be known at analysis time to determine which cache blocks conflict in the cache at runtime. This is the case if no virtual memory is used, or when using virtual memory one of the following is true: (1) the caches are physically indexed, or (2) the index derived from the virtual address is identical to the index derived

from the physical address. The second condition holds when the index bits are contained in the page offset of the address, or the page is colored so that the index bits match the physical address.

There are three different approaches to manage cache inclusivity in multi-level caches [6]. In an *inclusive* hierarchy, the L1 cache is enforced to contain a subset of blocks from the L2 cache. In an *exclusive* hierarchy, a cache block may only be contained in a single cache level at a time. No such condition is imposed in a *non-inclusive* hierarchy. A cache block may be contained only in the L1 cache, only in the L2 cache, or in both cache levels at the same time. We focus on *non-inclusive* cache hierarchies in this article.

As is the case in Reference [36], we impose the restriction that the number of ways in the L1 cache W_1 is less or equal to the number of ways in the L2 cache W_2. The same restriction is made for the number of sets in the L1 and L2 caches. These restrictions are not significant for real world systems, as the L2 cache is typically much larger than the L1 cache and easily satisfies these conditions.

Furthermore, we assume that the system is timing compositional [20]. This means that the WCRT, which describes the longest duration from the release of a job to its completion, may be computed using the WCETs and an additional penalty to account for the CRPD.

All memory accesses that are performed need to be known at analysis time in order to detect potential sources of CRPD. This means that the fetching behavior has to be deterministic; features like prefetching additional code on a cache miss and speculative execution are not considered in the analysis.

Branch predictors predict whether a conditional branch is taken or not. There are two main categories of branch prediction: *static* and *dynamic*. In static branch prediction, the prediction is made based on a simple rule. For example, conditional branches are assumed to be always taken or never taken. These branch prediction modes are supported, for example, by the Arm Cortex-R4 [28]. Another architecture implementing static branch prediction is the Infineon TC1.6E. 16 bit branches and 32 bit branches with backward displacement are predicted as taken, whereas 32 bit branches with forward displacement are predicted as not taken [23].

The presented analysis supports static branch prediction as the predictions made during runtime are known. Thus, it is possible to incorporate the static branch predictions into the CFG of the analyzed task by annotating the predicted instruction fetches. Dynamic branch prediction gathers information during runtime and performs the target prediction based on this information. As the prediction made during runtime is not known to the analysis, both scenarios, taking the branch and not taking the branch, need to be considered.

Another source of unpredictability is out-of-order execution. The order of accesses needs to be known to determine which conflicts occur in the caches. If the order of two accesses is not fixed, it is not possible to decide whether a potential conflict actually occurs during the analysis. Considering all possible scenarios creates high uncertainty in the analysis. For this reason, we consider an in-order pipeline.

3.1.2 Applicability to Commercial Processors. Modern architectures have evolved to contain an increasing number of features to improve the average case performance at the cost of predictability. Static worst-case analyses have failed to keep pace with these developments. This gap between the hardware architecture and static analyses have been noted already in the literature [12, 18, 32, 44]. There are two components to solving this problem: (1) ensuring predictability during the hardware design process by avoiding unpredictable features, and (2) improving the capabilities of static analyses to support more complex hardware architectures.

The former component has been addressed by Wilhelm et al. in Reference [43]. Recommendations for future architectures are made in order to enable precise and efficient analysis of the

system properties. For example, caches should use LRU replacement and be separated into instruction and data caches. Similarly, Reineke showed in Reference [37] that, for real-time systems, the LRU replacement policy is preferable over randomized replacement, which is used in the real-time focused Arm Cortex-R4, Cortex-R5, and Cortex-R7 processors [29].

Over the past years, the open-source RISC-V instruction set [42] has become increasingly popular for use in real-time systems [7, 17, 34]. The open-source nature of these architectures facilitates WCET analysis: System designers can incorporate predictability requirements into the architectural design, and researchers have access to specification details, eliminating the need for reverse-engineering of undocumented features. This trend is expected to continue, potentially increasing availability of timing-predictable processors in the future.

In this article, we focus on the latter component and aim at increasing the capabilities of static analysis methods to hardware architectures with multi-level cache hierarchies. We argue that, despite the strict requirements imposed on the architecture, the algorithms and proofs presented in this article are a first step to close the gap and advance the capabilities of static worst-case analysis.

3.2 Task Model

The tasks in the system are contained in a set T. Each task has three properties: its WCET, the period, which describes how often a new job of the task is released, and a relative deadline, before which each job has to be finished. We assume that tasks are scheduled according to a fixed priority. The set $hp(\tau) \subset T$ contains all tasks that may preempt $\tau \in T$. To denote the task currently being analyzed, we use the symbol τ; preempting tasks are denoted by φ or ψ.

The set $\mathbb{P}$ contains all program locations of τ. The CFG $(\mathbb{P}, E)$ of τ is given by the set of program locations and edges E that connect the locations. We associate every memory reference in the code to a location $p \in \mathbb{P}$. The reference associated to a location p causes a memory access when the control transfers to a new location p' with $(p, p') \in E$.

In the following, the set of all memory blocks is denoted by $\mathbb{M}$. The function $set_l : \mathbb{M} \to \mathbb{N}$ denotes the cache set for a block $m \in \mathbb{M}$ at the level $l \in \{1, 2\}$. The cache miss timing penalty of the L1 / L2 cache is written as d_{L1} / d_{L2}.

We assume that tasks do not share code. Sharing code between tasks is problematic when analyzing CRPD as the access classification for the second level cache can be invalidated by a preempting task. An access that is an L1 cache miss, without any preemptions, will reach the L2 cache and promote the targeted cache block to the youngest position in the L2 cache. We can thus be sure that, without preemptions, the target cache block is refreshed in the L2 cache. However, when this block is also used by another task, it can be loaded into the L1 cache during a preemption. As a result, the L2 cache may not be accessed for that particular request. It is no longer possible to make statements on the age of the shared cache block in the L2 cache. We have excluded code sharing for this reason in the presented analysis.

When determining the schedulability of a task set T, we assume that the timing overhead of the scheduler is negligible and the state of the caches are only affected by the preempting task and not the scheduler itself.

4 Background

In this section, we establish the background for cache analysis in the context of CRPD and give an overview of the current state-of-the-art analysis for non-inclusive cache hierarchies.

To determine whether a cache block is definitely cached at a particular location in the program, a *must* analysis is performed [14]. The *must* analysis determines an upper bound on the LRU age of each cache block for every program location. If the maximal age is less than the associativity of the cache, the block is guaranteed to reside in the cache. We denote the result of the *must* analysis

for the cache level $l \in \{1, 2\}$, at location p, for the cache block m by $MustAge_l^p(m)$. Note, that for LRU caches, the age of a cache block can be represented using the set $\{0, \ldots, W-1\} \cup \{\infty\}$, where W is the associativity of the cache and ∞ signifies that the block is not contained in the cache. When computing an upper bound on the cache age, we regard values larger than $W-1$ to be equivalent to ∞, as all values larger than $W-1$ show that the block is not definitely contained in the cache. The complementary analysis to the *must* analysis is the *may* analysis. It keeps track of the minimal age of cache blocks and is used to decide whether a cache block may be contained in the cache. The correctness of the *may* and *must* analyses are based on the theory of abstract interpretation [11].

Together, the results of the *may*/*must* analyses for the first-level cache can be used to determine the so-called ***cache-access-classification*** (**CAC**) for the second level cache [22]. The CAC of an access can take three values: *always* (A): the access will always reach the L2, i.e., it is a definite L1 cache miss; *never* (N): the access will never reach the L2, i.e., it is a definite L1 cache hit; *unknown* (U): the access may reach the L2. Note that the first-level cache is always accessed and that these classifications refer to an execution without any preemptions.

The analysis of CRPD is commonly based on the notion of *useful* and ***evicting*** **cache blocks** (**ECB**) [31]. To capture the effects of a preempting task, the cache blocks that may be stored in the cache by the preempting task are determined. These cache blocks are called ECB.

Definition 4.1 (Evicting Cache Block [4]). An evicting cache block is a cache block that may be accessed by a task. The set of ECB of task φ, cache level $l \in \{1, 2\}$, and cache set s is denoted by $ECB_l^{\varphi}(s)$.

When τ is preempted by φ, φ may also be preempted by all tasks $\psi \in hp(\varphi)$. Thus, the ECB of φ and $\psi \in hp(\varphi)$ may work together to evict data from τ. Such nested preemptions can be modeled by considering the union of all ECBs of potentially preempting tasks [4]. We introduce the following notation for the ECBs considering nested preemptions:

$$\widehat{ECB}_l^m = \left| \bigcup_{\psi \in \{\varphi\} \cup hp(\varphi)} ECB_l^{\psi}(set_l(m)) \right|. \tag{1}$$

The value $\widehat{ECB}_l^m$ is the number of cache blocks that preempting tasks may store in the same cache set as $m \in \mathbb{M}$ at the cache level $l \in \{1, 2\}$.

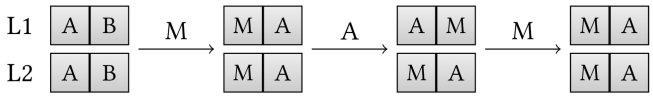
The ECB of a preempting task can evict data from the preempted task. To determine whether the evicted data contributes to the preemption penalty, cache blocks of the preempted task have been classified as UCB.

Definition 4.2 (Useful Cache Block [2]). A cache block $m \in \mathbb{M}$ is useful at a program location $p \in \mathbb{P}$ iff it is definitely cached at p and may be reused without eviction at a location reachable from p.

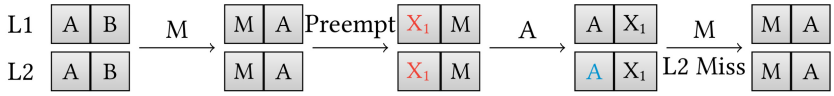
Definition 4.2 has been utilized for analyzing single-level caches. However, this definition is unsuited for multiple cache levels as it does not differentiate between the cache levels. For this reason, Rashid et al. [36] introduced the notion of *L1-useful* and *L2-useful* cache blocks.

Definition 4.3 (L1-Useful Cache Block [36]). A cache block $m \in \mathbb{M}$ is L1-useful at a program location $p \in \mathbb{P}$ iff it is definitely cached in L1 at p and may be reused without eviction from the L1 cache at a location reachable from p.

Definition 4.4 (L2-Useful Cache Block [36]). A cache block $m \in \mathbb{M}$ is L2-useful at a program location $p \in \mathbb{P}$ iff it is definitely cached in L2 at p and may be reused without eviction from the L2 cache at a location reachable from p. Furthermore, m must not be a L1-UCB at p.



(a) Unpreempted execution. The access to A and the second access to M result in an L1 hit.



(b) Execution with a preemption before the access to A. The preemption loads an ECB X_1 into the caches. The creation of indirect interference from A is highlighted in blue. The second access to M results in an L2 miss.

Fig. 1. Example of indirect interference. The access sequence M, A, M is shown: (a) without preemption and (b) with preemption before the access to A.

We denote the set of L1-useful / L2-useful blocks at location p by UCB_1^p / UCB_2^p. Using the Definitions 4.3 and 4.4, we can determine from which cache level a block may be reused and how high the penalty will be if it has to be reloaded after a preemption. In the remainder of this section, we will explain how a bound on the CRPD is computed using the state-of-the-art [36] approach.

An important concept in [36] is the so-called *maximal LRU age* of the block m. Note that, even though [10] and [36] use the name *CU* for similar concepts, the definition of this value is slightly different in [10]. Following the definition from [36], the value $CU_l^p(m)$ is the maximal age of $m \in \mathbb{M}$ in cache level l at the first reference, reachable from p, that accesses m. If there are multiple references that could result in the next access to m with different cache ages for m, the maximal age that is smaller than W_l is taken. This value essentially captures the minimal resilience of the cache block m to preemption effects to create an increase in the execution time (cf. [4]).

For example, consider a cache hierarchy with $W_1 = 2$ and $W_2 = 4$. Suppose a cache block m has the values $CU_1^p(m) = 1$ and $CU_2^p(m) = 2$ for some location $p \in \mathbb{P}$. This shows us that there exists a path from p to an access to m that hits in the L1 cache and the L1 age of m at that access is 1. It would require a single evicting cache block to evict m from the L1 cache on that path.

Furthermore, we know that there exists a path to an access to m where m has age 2 in the L2 cache. It would require 2 distinct interferences, either from ECB or indirect interference, to evict m from the L2 on this path. Thus, after a preemption that loads one ECB in the L1 cache, m can contribute one L1 miss to the CRPD. If the preemption loads two or more ECBs into the L2 cache, an L2 access to m could result in an L2 cache miss. Note that the paths that realize the maximal LRU ages for the L1 and L2 may be different. Thus, this is an approximation as both cache levels are considered in isolation for the CU_l^p value.

Using these definitions, we can now focus on the indirect effect of preemption in two-level caches.

Definition 4.5 (Indirect Effect of Preemption [10]). The indirect effect of preemption, called *indirect interference* in this article, is defined as the additional L2 cache conflicts that are caused by L1 cache misses due to preemption.

Figure 1 shows an example for indirect interference and its consequences. The analyzed caches are 2-way associative. We utilize capital letters to denote cache blocks. Initially, the L1 and L2 cache contain the cache blocks A, B; ordered in ascending LRU age. We analyze the access sequence M, A, M. The unpreempted scenario is shown in Figure 1(a). The access to A and the second access to M both result in an L1 cache hit.

ALGORITHM 1: Indirect Interference $Ind^{p}_{m_y}$ due to Preemption at Location $p \in \mathbb{P}$ [36]

Result: The indirect effect suffered by every $m_y \in \mathbb{M}$

1 **for** $m_y \in \mathbb{M}$ **do**
2 $\quad Ind^{p}_{m_y} \leftarrow \emptyset$
3 **end**
4 **for** $m_x \in UCB^{p}_{1}$ **do**
5 $\quad$ **if** $CU^{p}_{1}(m_x) + \widehat{ECB}^{m_x}_{1} \geq W_1$ **then**
6 $\quad\quad FA^{p}_{m_x} \leftarrow GetFirstAccess(m_x, p)$
7 $\quad\quad$ **for** $m_y \in \mathbb{M} \setminus \{m_x\}$ **do**
8 $\quad\quad\quad$ **if** $MustAge_2^{FA^{p}_{m_x}}(m_y) \neq \infty \wedge set_2(m_y) = set_2(m_x) \wedge$
9 $\quad\quad\quad MustAge_2^{FA^{p}_{m_x}}(m_x) > MustAge_2^{FA^{p}_{m_x}}(m_y)$ **then**
10 $\quad\quad\quad\quad Ind^{p}_{m_y} \leftarrow Ind^{p}_{m_y} \cup \{m_x\}$
11 $\quad\quad\quad$ **end**
12 $\quad\quad$ **end**
13 $\quad$ **end**
14 **end**

Now consider the preempted scenario from Figure 1(b). In this example, the block A causes indirect interference to M. The preemption occurs after the first access to M, before the access to A. The preempting task stores an ECB X_1 in the caches. The ECB is marked in red in Figure 1(b). Due to the preemption, the access to A results in an L1 cache miss. The block A is thus accessed in the L2 cache. This ages M in the L2 cache and causes its eviction from the L2 cache. Thus, A causes indirect interference to M, as in the scenario without preemption, A does not interfere with M in the L2 cache. This indirect interference is highlighted in blue in Figure 1(b). In consequence, the final access to M results in an L2 cache miss, even though it was not directly evicted by the ECB from the preemption.

Such indirect interference has to be considered in the CRPD estimation to arrive at a safe upper bound. We will now give an overview of the state-of-the-art approach [36] to analyze indirect interference and compute the CRPD in two-level non-inclusive caches.

The state-of-the-art method utilizes Algorithm 1 to determine the potential indirect interference, which causes useful blocks to age in the L2 cache, due to a preemption at the program location $p \in \mathbb{P}$. The cache blocks causing indirect interference for the block m_y due to a preemption at p are stored in the set $Ind^{p}_{m_y} \subset \mathbb{M}$.

The $Ind^{p}_{m_y}$ values are initialized as the empty set (lines 1–3). All L1-UCBs blocks at p, $m_x \in UCB^{p}_{1}$, are considered as potential sources of indirect interference (line 4). This is done because, in the absence of preemptions, these blocks can be served from the L1 cache without accessing the L2 cache. If they are evicted and have to be reloaded from L2 or memory, they create additional intra-task interference in the L2 cache.

Line 5 of Algorithm 1 checks whether the block m_x may be evicted from the L1 cache due to the ECBs of the preempting tasks. This is a necessary condition for m_x to contribute to the indirect interference for another block $m_y \neq m_x$. In Line 6, the function $GetFirstAccess(m_x, p)$ is used. This function "determine[s] the first reachable reference" [36] to m_x from P. No formal definition is given for this function in Reference [36]. As we will see in Section 5, the ambiguity inherent in this function description causes the analysis to compute unsafe bounds on the CRPD.

It is then determined for which $m_y \neq m_x$ the reloading of m_x causes indirect interference in the L2 cache (lines 8–11). This condition consists of three parts: (1) on the first access to m_x, m_y must

ALGORITHM 2: Indirect Interference $ColInd^p_{m_y}$ due to Multiple Preemptions, the First Preemption Occuring at $p \in \mathbb{P}$ [36]

Result: The indirect interference suffered by $m_y \in \mathbb{M}$ when multiple preemptions collaborate.

1 **for** $m_y \in \mathbb{M}$ **do**
2 $\quad ColInd^p_{m_y} \leftarrow \emptyset$
3 $\quad FA^p_{m_y} \leftarrow GetFirstAccess(m_y, p)$
4 $\quad PP \leftarrow GetProgramPoints(p, FA^p_{m_y})$
5 $\quad$ **for** $p' \in PP$ **do**
6 $\quad\quad ColInd^p_{m_y} \leftarrow ColInd^p_{m_y} \cup Ind^{p'}_{m_y}$
7 $\quad$ **end**
8 **end**

be cached in the L2, (2) m_x and m_y must be mapped to the same cache set, and (3) m_x must be older than m_y in the L2 cache. (This includes m_x not being cached in the L2.) If these conditions are true, the block m_x potentially causes indirect interference to m_y and is added to $Ind^p_{m_y}$ in line 10.

The authors of [36] showed that multiple preemptions may collaborate to create a higher CRPD than each preemption could in isolation. To consider this fact in the analysis, Algorithm 1 is augmented to check all program locations, starting from p and leading to the first access of the analyzed block m_y. The pseudocode for this algorithm is shown in Algorithm 2. The set of all program locations contained in paths starting from p and ending in $FA^p_{m_y}$ is given by the function $GetProgramPoints(p, FA^p_{m_y})$. The cache blocks causing indirect interference are accumulated for all of these program locations to yield an upper bound on the indirect aging events $ColInd^p_{m_y}$. This upper bound captures the potential indirect interferences from multiple preemptions between p and $FA^p_{m_y}$.

Using this upper bound, the CRPD from L1 hits being degraded due to preemptions can be bounded. An L1 hit may either degrade to an L2 hit or an L2 miss. The former case is accounted for by γ^p_{L1} in Equation (2), while the latter penalty is accumulated by γ^p_{L1+L2} in Equation (3). For an access to contribute in one of these two ways, the targeted cache block m_y has to be an L1-UCB. To result in an L1 miss, the ECB have to be able to evict the block from the L1 cache, i.e., $CU^p_1(m_y) + \widehat{ECB}^{m_y}_1 \geq W_1$. This condition is the same for both Equations (2) and (3). The difference between an L2 hit and miss is whether the L2 ECBs and the indirect interference can evict the block also from the L2 cache. If the interference due to ECBs and indirect interference causes the maximal LRU age to be greater or equal to W_2, m_y may also be evicted from the L2 cache.

$$\gamma^p_{L1} = d_{L1} \cdot \left| \left\{ m_y \in UCB^p_1 \;\middle|\; CU^p_1(m_y) + \widehat{ECB}^{m_y}_1 \geq W_1 \wedge CU^p_2(m_y) + \widehat{ECB}^{m_y}_2 + |ColInd^p_{m_y}| < W_2 \right\} \right|, \tag{2}$$

$$\gamma^p_{L1+L2} = (d_{L1} + d_{L2}) \cdot \left| \left\{ m_y \in UCB^p_1 \;\middle|\; CU^p_1(m_y) + \widehat{ECB}^{m_y}_1 \geq W_1 \wedge CU^p_2(m_y) + \widehat{ECB}^{m_y}_2 + |ColInd^p_{m_y}| \geq W_2 \right\} \right|. \tag{3}$$

For each block contributing in this way to the CRPD, a cache miss penalty of d_{L1} cycles is added for each L1 cache miss and d_{L2} is added for each L2 cache miss. The value $\gamma^p_{L1} + \gamma^p_{L1+L2}$ thus gives an upper bound on the delay caused by reloading UCB into the L1 cache.

There is a third type of preemption delay in a two-level cache hierarchy that is not covered by Equations (2) and (3). References that result in an L1 miss but L2 hit may be degraded to an L2

ALGORITHM 3: CRPD from L2 Cache Misses [36]

Result: The delay suffered from L2 hits that are degraded to an L2 miss due to preemption.

1 $\gamma^{p}_{L2} \leftarrow 0$
2 **for** $m_y \in \widehat{UCB}^{p}_{2}$ **do**
3 $\quad \gamma_{m_y} \leftarrow 0$
4 $\quad \mathbb{R} \leftarrow GetHitLocations^{p}(m_y)$
5 $\quad$ **if** $MustAge^{R^1}_{2}(m_y) + \widehat{ECB}^{m_y}_{2} + \left|ColInd^{p}_{m_y}\right| \geq W_2$ **then**
6 $\quad\quad \gamma_{m_y} \leftarrow \gamma_{m_y} + 1$
7 $\quad$ **end**
8 $\quad$ **if** R^1 *is in loop* **then**
9 $\quad\quad$ **if** $MustAge^{R^1}_{2}(m_y) + \left|ColInd^{p}_{m_y}\right| \geq W_2$ **then**
10 $\quad\quad\quad \gamma_{m_y} \leftarrow \gamma_{m_y} + 1$
11 $\quad\quad$ **end**
12 $\quad$ **end**
13 $\quad$ **for** $1 < n \leq |\mathbb{R}|$ **do**
14 $\quad\quad$ **if** $MustAge^{R^n}_{2}(m_y) + \left|ColInd^{p}_{m_y}\right| \geq W_2$ **then**
15 $\quad\quad\quad \gamma_{m_y} \leftarrow \gamma_{m_y} + 1$
16 $\quad\quad$ **end**
17 $\quad$ **end**
18 $\quad \gamma_{m_y} \leftarrow \min\{\gamma_{m_y}, |\mathbb{R}|, |ColInd^{p}_{m_y}| + 1\}$
19 $\quad \gamma^{p}_{L2} \leftarrow \gamma^{p}_{L2} + (\gamma_{m_y} \cdot d_{L2})$
20 **end**

miss due to a preemption. In the state-of-the-art method, this component of the CRPD is computed using Algorithm 3.

The set $\widehat{UCB}^{p}_{2}$ contains all memory blocks that are an L2-UCB in any location reachable from p. We have to consider all blocks that may become L2-UCBs even after the preemption because the indirect effect of a previous preemption can still cause these blocks to miss in the L2 cache. In contrast, we only had to consider the first access to an L1-UCB after the preemption, as after the first access the block will be the youngest block in the L1 cache and does not experience further interference from the preemption in the L1 cache.

For each memory block $m_y \in \widehat{UCB}^{p}_{2}$, the locations where an L2 hit takes place is stored in $\mathbb{R}$ by the function $GetHitLocations^{p}$ (line 4). The symbol $\mathbb{R}$ represents a sequence of references $(R^1, R^2, \ldots)$ that target the block m_y and result in an L2 hit. Note that no formal definition of this function is given in Reference [36].

In the following lines 5–17, it is checked whether these L2 hits may degrade to an L2 miss due to the preemption effects. The algorithm differentiates between the first L2 hit R^1 after a preemption and later accesses. The first reference to m_y after preemption may also experience direct interference from the ECBs of the preempting task. The potential eviction of the first reference is handled in lines 5–7. If the first reference may be reached again due to a loop, it can also contribute to the CRPD solely due to indirect interference. This is covered in lines 8–12. For all later accesses, only the indirect interference plays a role in the eviction of the target block (lines 13 — 17). The total contribution of m_y to this component of the CRPD is limited by $\min\{|\mathbb{R}|, |ColInd^{p}_{m_y}| + 1\}$ [36] (line 18). The total delay from L2-UCBs is accumulated in γ^{p}_{L2} (line 19).

According to Reference [36], Equations (2),(3), and Algorithm 3, which computes γ^p_{L2}, allow us to compute all three components of the preemption delay. The total CRPD γ is computed as the maximal sum of the three components γ^p_{L1}, γ^p_{L1+L2}, and γ^p_{L2} for any program location $p \in \mathbb{P}$:

$$\gamma = \max_{p \in \mathbb{P}} \left(\gamma^p_{L1} + \gamma^p_{L1+L2} + \gamma^p_{L2} \right). \tag{4}$$

However, as we will discuss in the following section, this approach contains several issues that may lead to an unsafe CRPD estimate.

5 Safety Issues and Pessimism in the State-Of-The-Art

In this section, we will analyze the state-of-the-art and highlight issues inherent in the approach. This is structured as follows:

First, in Section 5.1, we show that the indirect effect of a preemption may be underestimated in Algorithm 2. The algorithm considers the potential for indirect interference only up to the first reference. However, the algorithm does not consider accesses that have an uncertain cache-access classification, i.e., accesses that may not reach the L2 cache in every circumstance. Considering the CAC of accesses to the L2 cache is crucial in the analysis, as an access that does not reach the L2 cache will not refresh or load the targeted cache block to the youngest LRU position in the L2 cache. The indirect effects may thus still affect future L2 accesses to the same memory block. We show that multiple preemptions can create an L2 cache miss, that is not detected by the state-of-the-art analysis due to this issue. This can violate the safety of the analysis as the CRPD is underestimated.

In Section 5.2, we identify pessimism in the analysis of indirect interference. Algorithm 1 does not consider whether the access creating indirect interference actually occurs before the analyzed access. Thus, cache blocks are considered to create indirect interference even though they will always be accessed *after* the analyzed block. The indirect interference may thus be overestimated, which causes a pessimistic CRPD bound.

Section 5.3 discusses Algorithm 3. The algorithm is incomplete in the sense that it is not able to process all possible CFGs. In particular, the algorithm assumes that there will be one distinct reference to an L2-UCB which will perform the first access to the cache block. However, as the CFG may split into multiple branches, this is not always the case. We demonstrate such a scenario in Section 5.3.1. Finally, in Section 5.3.2, we show that the CAC for accesses to L2-UCBs is not correctly integrated in Algorithm 3. The algorithm thus overlooks potential L2 cache misses, resulting in an unsafe CRPD estimate.

5.1 Unsafe Estimations in Algorithm 2

It is possible for Algorithm 2 to underestimate the indirect interference due to multiple preemptions. This causes the resulting CRPD bound to be unsafe. The issue originates from the use of the $GetFirstAccess(m_y, p)$ function on line 3 in Algorithm 2.

Recall the definition of $GetFirstAccess()$ from [36]: "$GetFirstAccess(m_x, p)$ [...] is used to determine the first reachable reference to every m_x". No formal definition of this function is given. Note in particular that the CAC of the first reference is not considered.

Algorithm 2 checks all locations from the preemption at p to the first reference of the memory block and collects all memory blocks that may create indirect interference. In a two-level cache hierarchy, an access to a memory block does not guarantee to refresh the block at the L2 cache. Thus, indirect interference that has accumulated in the second level cache is not removed if the first access hits in the L1 cache.

An access beyond which indirect interference does not propagate has been termed a *firewall* in [47]. To be a firewall for data in the L2 cache, an access has to have an L2 CAC of *always*. This fact is not considered in Algorithm 2. Thus, there are scenarios in which the state-of-the-art method does not recognize a potential cache miss, leading to an unsafe CRPD bound.

We will now construct such a scenario in Figure 2. Consider the following cache configuration: the L1 and L2 caches are 4-way associative, while the L2 possesses more sets than the L1. We utilize capital letters to represent cache blocks. In the initial state, the L1 cache contains the blocks B, M, D, A, ordered in ascending LRU age. The L2 cache initially contains M, D, A, C. The access sequence to cache blocks is A, M, B, E, F, S, B, M. The critical access we focus on in this example is the final access to the block M.

Two different scenarios, without and with preemptions, are shown in Figure 2(a) and 2(b) respectively. The cache state is depicted by two rows of cache blocks. The first row represents the L1 cache, the second row represents the L2 cache. The blocks are ordered in ascending LRU age; the youngest block is placed at the left most position. We only depict a single set of the L2 for clarity. Note that the block S is mapped to a different L2 set. It thus only appears in the L1 cache set but not in the shown L2 cache set. We can see that during normal execution, the final access to M will result in an L2 hit.

It is possible to place two preemptions in this access sequence to cause the second access to M to miss. In Figure 2(b), a preemption happens directly at the beginning of the sequence and another preemption happens before the second reference to B. Each preemption causes one evicting cache block X to be stored in the L1 cache; it does not affect the shown L2 cache set. This is possible if X is mapped to a different L2 cache set, which is not shown in the figure. In Figure 2(b), the insertion of ECB is marked in red.

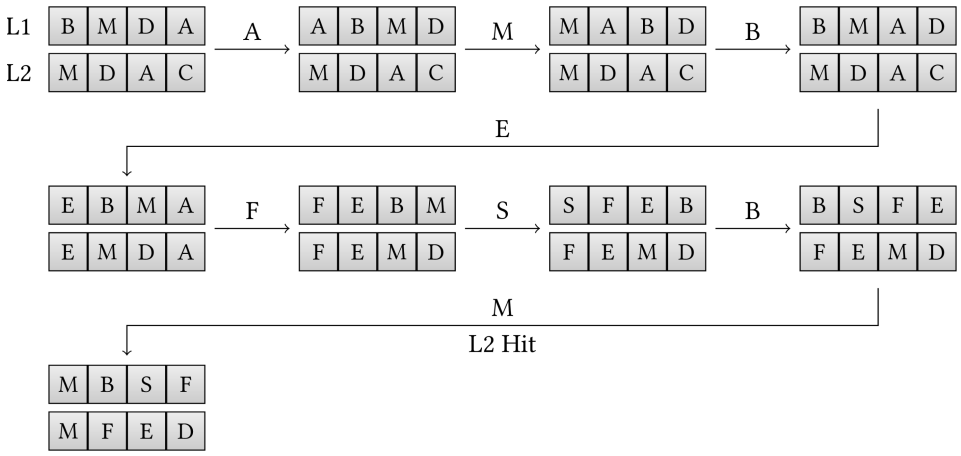
The first preemption causes the block A to be evicted from the L1 cache. Thus, when A is accessed after the preemption, it creates indirect interference for M in the L2 cache. In the figure, the creation of indirect interference is marked in blue. The L2 age of M increases from 0 (as in the unpreempted scenario) to 1. As the first reference to M does not reach the L2 cache, the age of M in the L2 cache remains 1 after the first reference to M. The second preemption evicts B from the L1 cache. B is accessed after the preemption and causes indirect interference to M. The L2 access for B actually causes M to be evicted from the L2 cache, thus incurring an L2 miss for the final access to M.

The final access to M experiences two indirect interferences. The age of M, in the unpreempted case, at the final access is 2. Adding the indirect interference to the cache age shows that M will be evicted as $2 + 2 \geq 4$ (cf. line 14 of Algorithm 3).

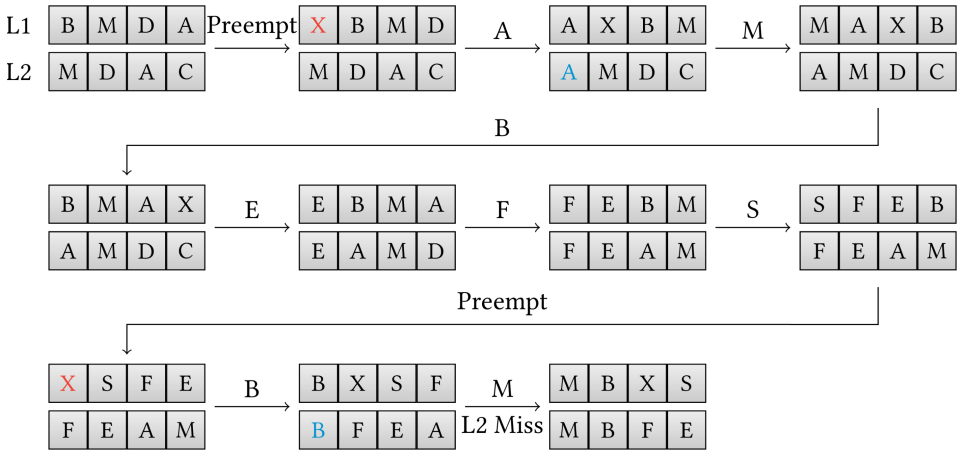
However, this L2 miss is not detected by the state-of-the-art method, as it stops considering the indirect interference after the first access to M (see Algorithm 2 lines 4–7). While the indirect interference of A is detected, the interference from B is missed as B's L1 age is 2 at the first acccess to B (see line 5 in Algorithm 1). B will thus not be evicted by the first preemption and, according to Algorithm 2 not cause indirect interference for later L2 hits for M. Consequently, the state-of-the-art CRPD computation is unsafe in this situation.

5.2 Pessimism in Algorithm 2

In addition to the optimism discussed in the previous section, the state-of-the-art method may also overestimate the indirect interference, leading to an overly pessimistic CRPD estimation. The overestimation originates from the fact that Algorithm 1 does not consider the ordering of accesses. In line 6 of Algorithm 1, the first access to the block m_x is determined. It is then checked whether m_x may cause indirect interference on another block m_y in lines 7–12. However, it is not considered whether an access to m_y will happen before or after the first access to m_x. m_x will only contribute



(a) Simulation of the access sequence A, M, B, E, F, S, B, M without preemptions. The final access to M results in an L2 hit.



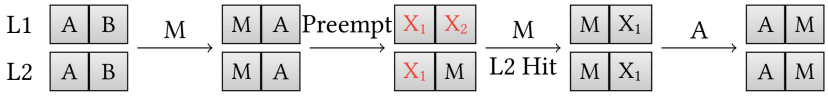
(b) Simulation of the access sequence A, M, B, E, F, S, B, M with two preemptions. The first preemption occurs before the access to A and the second preemption occurs before the second access to B. Both preemptions cause the ECB X to be loaded into the L1 cache. The insertion of ECBs is marked in red, while the creation of indirect interference is marked in blue. The final access to M results in an L2 miss.

Fig. 2. Simulation of the access sequence A, M, B, E, F, S, B, M: (a) without any preemption and (b) with two preemptions.

to the indirect interference, if it happens prior to the access to m_y. Thus, a block m_x may be pessimistically included in $Ind^p_{m_y}$ even though it will not cause m_y to age prior to the next access targeting m_y.

We will now provide an example of such pessimistic behavior in Figure 3. Consider the following cache configuration: the L1 and L2 caches are both 2-way associative, while the L2 cache has more sets than the L1 cache. The access sequence is M, M, A. Both analyzed cache sets of L1 and L2 start with the initial state A, B. The access we focus on is the second access to M. Without any preemptions, the second access to M results in an L1 hit. This behavior is shown in Figure 3(a).

(a) Simulation of the access sequence M, M, A. The second access to M and the access to A result in an L1 hit.



(b) Simulation of the access sequence M, M, A with a preemption before the second access to M. The second access to M results in an L2 hit.

Fig. 3. Simulation of the access sequence M, M, A: (a) without any preemptions and (b) with a preemption. The L1 and L2 caches are 2-way associative. The L2 cache has an additional set, which is not shown here, that X_2 is mapped to.

Let us now assume a preemption occurs after the first access to M. Let p denote that location. The preemption causes two blocks X_1 and X_2 to be loaded into the L1 set. X_1 is mapped to the same L2 set as M and A, while X_2 is mapped to a different L2 set. X_2 is thus not shown in the L2 cache in this depiction. The resulting sequence of cache states is shown in Figure 3(b). After the preemption, the second reference to M will not result in an L1 hit, as it has been evicted by X_1 and X_2. However, it will still result in an L2 hit, as only X_1 caused interference for M in the L2 cache. The preemption causes a delay of one L1 miss penalty due to the second reference to M and a full reload penalty for A.

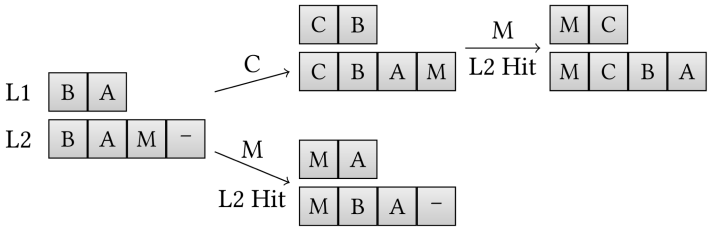
We will now compute the indirect effect on M using the state-of-the-art method. For this, we have to check all L1-UCBs at the site of preemption (see Algorithm 1, line 4). A is definitely cached in the L1 prior to the preemption and will be reused without eviction. Consequently, it is an L1-UCB. Furthermore, its maximal L1 LRU age is 1. The number of L1 ECBs is 2, meaning that A may be evicted due to the preemption from the L1 cache. It is thus a candidate for indirect interference with M. Algorithm 1 checks whether on the next access to A (line 6), the block M is cached in the L2 and has a lower LRU age compared with A (lines 8–9). This is the case in the unpreempted scenario. Hence, Algorithm 1 regards A as a source of indirect aging for M (line 10) and sets $Ind^p_M = \{A\}$. As the preemption happens directly before the next reference to M it follows that $ColInd^p_M = Ind^p_M = \{A\}$.

As M is an L1-UCB, it may contribute to the CRPD in two different ways. It may be degraded to an L2 hit or an L2 miss. The conditions for these penalties are given in Equations (2) and (3), respectively. As is the case for A, M may also be evicted from the L1 cache due to the L1 ECBs. When the execution is preempted, the maximal L2 age of M is 0. Thus, it may not be evicted from the L2 cache solely due to the L2 ECB X_1.

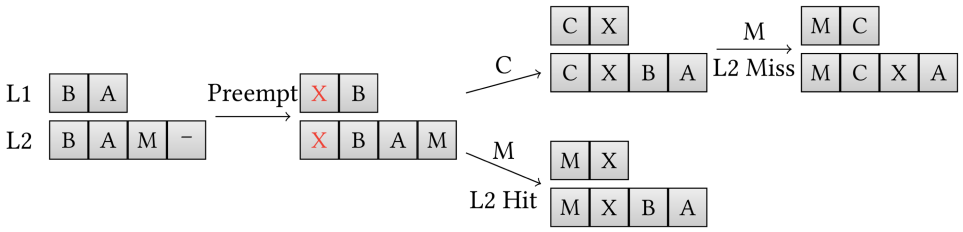
However, due to the potential indirect interference from A, as computed by Algorithm 2, M is considered to be potentially degraded to an L2 miss:

$$CU^p_2(M) + \widehat{ECB}^M_2 + \left|ColInd^p_M\right| = 0 + 1 + 1 \geq 2. \tag{5}$$

Consequently, Equation (3) adds a delay of $(d_{L1} + d_{L2})$ cycles to the CRPD value due to a preemption at p, caused by the need to potentially reload M into both cache levels. However, it is clear that A will not create indirect interference for the second reference to M as A will only be accessed after M. In reality, the access to M will not result in an L2 miss but an L2 hit. This can be seen clearly in Figure 3(b). The state-of-the-art method overestimates the CRPD penalty in this situation because it does not consider the ordering of accesses when computing indirect interference.



(a) Analysing both paths without any preemptions shows that both first accesses to M would result in an L2 cache hit.



(b) Considering a preemption at the beginning of the sequence causes different behavior for the upper and lower path. On the upper path, the first access to M results in an L2 miss, while on the lower path the first access to M results in an L2 hit.

Fig. 4. A branch in the control flow creates multiple possible *first accesses* to M. There exist two different paths leading to first accesses of M, with different L2 cache ages. Both paths need to be considered to compute a safe CRPD estimate.

5.3 Analysis of Algorithm 3

The preemption delay caused by L2 hits being degraded to L2 misses is computed using Algorithm 3 in the state-of-the-art method. In this section, we will closely examine this algorithm and highlight two distinct issues.

5.3.1 Incompleteness of Algorithm 3. A key component of Algorithm 3 is $GetHitLocations^p(m_y)$, which returns a list of all locations containing a reference to m_y that result in an L2 hit and are reachable from p (line 4).

The first reachable reference is handled differently from the later references, as it may also be subject to direct interference from the loaded ECBs. However, the approach does not consider that there may be multiple references that potentially execute the first access to the block m_y after the preemption. This can happen if there is a branch in the control flow, creating two distinct paths to different first references of the cache block. Thus, there may be multiple execution traces that have a different reference as the first reference to the analyzed cache block.

Such a scenario is shown in Figure 4. In the example, the L1 cache is 2-way associative and the L2 cache has 4-way associativity. The initial state contains B, A in the L1 cache and B, A, M in the L2 cache. The control flow splits into the upper and lower branch. In the upper branch, an additional access to C is performed. Thus, there are two possible paths, which have an access sequence of C, M and M, respectively. If there are no preemptions, the reference to M results in an L2 hit on both paths, as can be seen in Figure 4(a). In Figure 4(b), the same access sequences are analyzed with a preemption happening before the control-flow branch. The preemption causes the block X to be stored in the L1 and L2 cache. This additional interference affects the upper and lower path in different ways. While in the lower path, the reference to M still results in an L2 hit, in the upper path M is evicted prior to the access.

As Algorithm 3 does not account for a situation with multiple possible first accesses, the example is not analyzable with the state-of-the-art method.

5.3.2 Handling of Uncertain L2 Accesses. Algorithm 3 treats the first reference to a memory block after a preemption differently from subsequent references. This distinction is motivated by the observation that after an access to a memory block, it will be younger than the ECB stored in the cache by the preemption. Thus, later accesses will only be subject to indirect interference.

Lemma 2 from Reference [36] tries to apply this observation and states that "[...] it is only the first reference to m_y after P [...] that can be directly impacted due to preemption at P." However, this justification for the structure of Algorithm 3 overlooks the fact that the first reference may not reach the L2 cache and the block m_y might not be refreshed. This situation occurs when the cache access classification of the first reference to m_y after P is *unknown*, i.e., m_y may be contained in the L1, but it is not certain that it is. If this situation occurs, not only the first reference may be impacted by the direct interference in the L2 cache. Also the reference that actually causes the first L2 access to the block m_y can be impacted by direct interference. As Algorithm 3 only checks if m_y may be evicted from the L2 cache at the first reference to m_y, it can underestimate the CRPD.

We will demonstrate this using an example. Consider the access sequence B, A, M, [A, B,] M, C, D, M. The two references [A, B] denote a split in the CFG. These references may be executed, but the control can also skip these accesses, e.g., by a conditional jump. We will now analyze this sequence in the following cache configuration: The caches are 2-way associative; the L1 cache has a single set, while the L2 cache contains two sets.

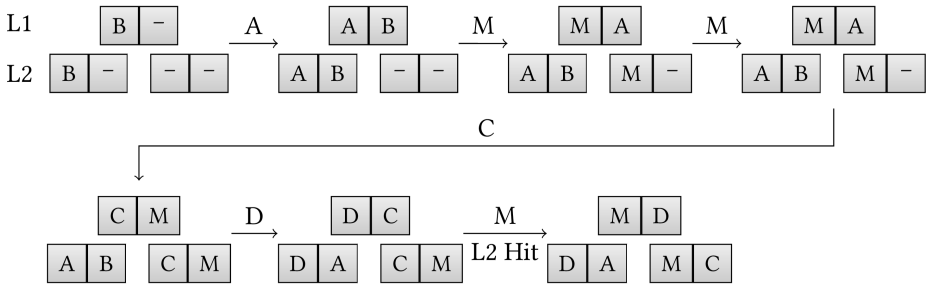
In Figure 5, the cache state simulation for an execution that skips the references [A, B] is shown. In Figure 5(a), the cache state sequence without any preemptions is shown. In between the two references to M, M is contained in the L1 cache, but it is not an L1-UCB. M is not considered to be an L1-UCB as the two optional references [A, B] may evict M from the L1 cache. Directly before the second access to M, M is not definitely cached in the L1 cache, prohibiting its classification as an L1-UCB. In both situations, regardless whether [A, B] is executed or not, M will be in the L2 cache. Thus, M is classified as an L2-UCB (see Definitions 4.3 and 4.4). As M will be in the L2 cache in all cases, the timing analysis will assume that the final reference to M results in an L2 hit. Thus, a potential L2 cache miss must be captured in the CRPD computation.

In Figure 5(b), the effects of a preemption prior to the second reference to M are shown. The preemption stores a single block X in the L1 cache and the second L2 cache set. The preemption does not evict M from the L1 cache. The second reference to M will result in an L1 hit. This situation creates the issue in Algorithm 3. As the first reference to M after preemption resulted in an L1 hit, M is not refreshed in the L2 cache. The direct interference created by X still exists in the L2 cache.

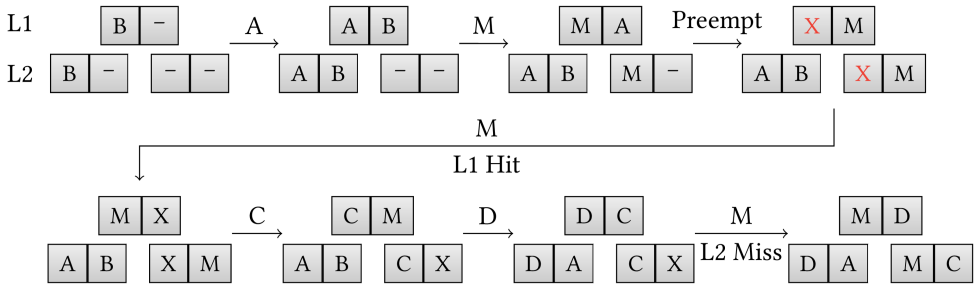
Finally, we can compare the behavior of the third access to M in the unpreempted and preempted scenario. In Figure 5(a), the third reference to M at the end of the sequence results in an L2 hit. However, for the preemption shown in Figure 5(b), as the direct interference still exists, the final access to M will cause an L2 miss. This potential L2 miss is not detected by Algorithm 3, because the direct interference of the preemption is only evaluated for the second reference to M. At that stage, M will not be evicted from the L2 due to the ECBs. However, the direct interference still applies to the third reference to M and causes an L2 miss, which leads to an unsafe CRPD bound for the shown preemption.

6 CRPD in Concrete Semantics

The previous section has demonstrated that the interaction between the L1 and L2 cache after a preemption is non-trivial. In order to be confident that an analysis does not overlook any edge case and thus computes a safe bound on the CRPD, a formal foundation for the key concepts is



(a) Simulation of the access sequence B, A, M, M, C, D, M without any preemptions. The third reference to M results in an L2 hit.



(b) Simulation of the access sequence B, A, M, M, C, D, M, with a preemption between the first and second reference to M. The direct interference by X contributes to the eviction of M prior to the third reference to M, causing an L2 miss.

Fig. 5. Simulation of the access sequence B, A, M, M, C, D, M: in (a) without preemption, in (b) with a preemption. The caches are 2-way associative; the L1 has one set and the L2 has two sets.

required. In this section, we formalize the indirect interference that may occur due to a preemption. To capture all possible interference scenarios, we operate at the level of concrete execution traces in this section. An execution trace is a sequence of program locations and the concrete state of both cache levels at those locations. The formalization established in this section lays the foundation for the safe CRPD analysis we will construct in the following Sections 7 and 8.

In a concrete situation, an LRU cache set contains W_l cache blocks, where l is the level of the cache and W_l is the associativity. The contained blocks are ordered by the LRU property. The cache state can thus be represented as a sequence of W_l blocks. Let C^l be the set that contains all possible concrete cache states:[1]

$$C^l = \left\{(m_0, \ldots, m_{W_l-1}) \in \mathbb{M}^{W_l} \;\middle|\; i \neq j \implies m_i \neq m_j\right\}. \tag{6}$$

As cache sets operate independently of each other, we formalize the state of the system for a single cache set. The state of the two-level hierarchy can be represented using elements from $C^1 \times C^2$. If an access to the memory block m transforms $(c_1^1, c_1^2) \in C^1 \times C^2$ into $(c_2^1, c_2^2) \in C^1 \times C^2$ according to the semantics of a two-level non-inclusive LRU cache hierarchy, we write $(c_1^1, c_1^2) \rightarrow^m (c_2^1, c_2^2)$. Furthermore, we use $c^l(m)$ as a shorthand for the position of m in the concrete cache state c^l, i.e., the age of m; starting at 0, going up to $W_l - 1$. The value of $c^l(m)$ is ∞ if m is not contained in c^l.

[1]Note that a (partially) empty cache can be modelled by adding "empty" blocks to the set $\mathbb{M}$ to fill up the cache ways.

We represent the concrete state of the analyzed task using the set S:

$$S = \mathbb{P} \times C^1 \times C^2. \tag{7}$$

A tuple $(p, c^1, c^2) \in S$ corresponds to the system state directly before executing the access associated to p. In the state $(p, c^1, c^2) \in S$, the access associated to p, targeting the cache block m_p, results in an L1 hit if $c^1(m_p) < W_1$; an L2 hit if $c^1(m_p) = \infty \wedge c^2(m_p) < W_2$; an L2 miss otherwise.

For $s_1, s_2 \in S$ we write $(p_1, c_1^1, c_1^2) \rightarrow (p_2, c_2^1, c_2^2)$ iff $(p_1, p_2) \in E$ and $(c_1^1, c_1^2) \rightarrow^{m_{p_1}} (c_2^1, c_2^2)$, where E is the set of edges in the CFG. This allows us to create the set of all possible execution traces, starting from the initial program location $p^0 \in \mathbb{P}$ and arbitrary cache states c^1, c^2:

$$\mathcal{S} = \{(s_1, \ldots, s_n) \mid s_1 = (p^0, c^1, c^2) \in S \wedge 1 \leq i < n : s_i \rightarrow s_{i+1}\}. \tag{8}$$

6.1 Analyzing L1 Hits in Concrete Semantics

We can now define the notion of the first access in the concrete semantics.

Definition 6.1 (First Access in Concrete Semantics). In an execution trace $(s_1, \ldots, s_n) \in \mathcal{S}$, the first access to $m \in \mathbb{M}$ after p_i, $s_i = (p_i, c_i^1, c_i^2)$, is defined as the location p_j, $s_j = (p_j, c_j^1, c_j^2)$, with the smallest $j, i \leq j \leq n$ such that the reference associated to p_j targets m.

Using the concept of the first access, we can determine the indirect interference affecting accesses that result in an L1 hit in the absence of preemption.

Lemma 6.2. *The indirect interference in a trace $s = (s_1, \ldots, (p_i, c_i^1, c_i^2), \ldots, (p_k, c_k^1, c_k^2), \ldots, s_n) \in \mathcal{S}$ experienced by an L1 access to m associated to p_k due to a preemption at $p = p_i$ is bounded by $L1I_m^p$.*

$$L1I_m^p(s) = \left\{ m_j \in \mathbb{M} \;\middle|\; \begin{array}{l} p_k \text{ is the first access to } m \text{ after } p_i \text{ in } s \wedge \\ \forall j, i \leq j < k : \\ c_j^1(m_j) < W_1 \wedge c_j^1(m_j) + \widehat{ECB}_1^{m_j} \geq W_1 \wedge \\ c_j^2(m_j) > c_j^2(m_k) \end{array} \right\} \tag{9}$$

Proof. Indirect interference from a preemption at p_i for an access resulting in an L1 hit in the absence of preemptions is limited to the first reference of the memory block [10]. Any accesses happening after the preemption at location p_i and before the first reference to m in p_k may contribute to the indirect interference. For this reason, the index j is limited to $i \leq j < k$.

As noted by Rashid et al. in Reference [36], multiple preemptions prior to p_k may collaborate to increase the indirect interference. Thus, all states in the trace s between p_i and p_k are considered.

The condition for an access to cause indirect interference is that the targeted cache block resides in the L1 cache in the absence of preemptions and may be evicted by the preemptions. Furthermore, its L2 age has to be larger than the age of m_k to cause m_k's age to increase. Thus, $L1I_m^p$ is an upper bound on the indirect interference for the access associated to p_k in the particular trace s. □

We write $L1I_m^p(\mathcal{S})$ for $\bigcup_{s \in \mathcal{S}} L1I_m^p(s)$. We will use Lemma 6.2 to determine an upper bound on the aging from indirect interference over all feasible traces using a **data-flow analysis** (**DFA**) in Section 7.

6.2 Analyzing L2 Hits in Concrete Semantics

Similar to the notion of first access, we define the first L2 access and first L2 hit to a memory block. These definitions are needed to formalize the analysis of accesses that result in an L2 hit in the absence of preemptions.

Definition 6.3 (First L2 Access in Concrete Semantics). In an execution trace $(s_1, \ldots, s_n) \in \mathcal{S}$, the first L2 access to m after p_i, $s_i = (p_i, c_i^1, c_i^2)$, is defined as the location p_j, $s_j = (p_j, c_j^1, c_j^2)$, with the smallest $j, i \leq j \leq n$ such that the reference associated to p_j targets m and $c_j^1(m) = \infty$.

Definition 6.4 (First L2 Hit in Concrete Semantics). In an execution trace $(s_1, \ldots, s_n) \in \mathcal{S}$, the first L2 access to m after p_i, $s_i = (p_i, c_i^1, c_i^2)$, is defined as the location p_j, $s_j = (p_j, c_j^1, c_j^2)$, with the smallest $j, i \leq j \leq n$ such that the reference associated to p_j targets m and $c_j^1(m) = \infty \wedge c_j^2(m) < W_2$.

In order to identify which references will hit in the L2 cache and are possibly affected by direct interference from ECBs, we determine the set of reachable first L2 hits.

Definition 6.5 (Reachable First L2 Hits). For a location $p \in \mathbb{P}$, the set of reachable first L2 hits to block $m \in \mathbb{M}$ is given by

$$F_m^p = \left\{ p_j \in \mathbb{P} \;\middle|\; \begin{array}{l} \exists\, s = (s_1, \ldots, (p_i, c_i^1, c_i^2), \ldots, (p_j, c_j^1, c_j^2), \ldots, s_n) \in \mathcal{S} : \\ p_i = p \wedge \\ p_j \text{ is the first L2 access and first L2 hit for } m \text{ after } p_i \text{ in } s \end{array} \right\}. \tag{10}$$

Lemma 6.6. *The set F_m^p contains all locations that are reachable from $p \in \mathbb{P}$, where an access targeting $m \in \mathbb{M}$ and resulting in an L2 hit may experience direct and indirect interference in the L2 cache due to the preemption at p.*

Proof. There are several conditions in order for an access to be impacted by direct interference. The reference at p_j must reach the L2 cache and m must be contained in the L2 cache at the time of the preemption. Otherwise, m will not age due to the ECBs. m will be cached in the L2 at p_i because p_j performs the first L2 access to m after p_i and this access results in an L2 hit. This can only be the case if m was already cached in L2 at p_i. If p_j does not execute the first L2 access to m in the trace, another prior access will have eliminated the direct interference by refreshing m in the L2.

These conditions are evaluated for every feasible program trace. Thus, the set F_m^p contains all locations where direct and indirect interference may contribute to the eviction of m from the L2 cache. □

Similar to the notion of the reachable first L2 hits are the reachable later L2 hits. These accesses may still experience indirect interference from a preemption but are isolated from the direct effects of the ECBs as they are protected by another access that occurs first.

Definition 6.7 (Reachable Later L2 Hits). For a location $p \in \mathbb{P}$, the set of reachable later L2 hits to block $m \in \mathbb{M}$ is given by

$$L_m^p = \left\{ p_j \in \mathbb{P} \;\middle|\; \begin{array}{l} \exists\, s = (s_1, \ldots, (p_i, c_i^1, c_i^2), \ldots, (p_j, c_j^1, c_j^2), \ldots, s_n) \in \mathcal{S} : \\ p_i = p \wedge \\ p_j \text{ is not the first L2 access to } m \text{ after } p_i \text{ in } s \wedge \\ c_j^1(m) = \infty \wedge c_j^2(m) < W_2 \end{array} \right\}. \tag{11}$$

Lemma 6.8. *The set L_m^p contains all locations reachable from $p \in \mathbb{P}$, where an access targeting $m \in \mathbb{M}$, resulting in an L2 hit, may experience indirect interference without direct interference due to the preemption at p.*

Proof. All feasible program traces in $\mathcal{S}$ are considered to build the set L_m^p. A location p_j is included in the set only if the access associated to that location results in an L1 miss and an L2 hit. As p_j is guaranteed to not issue the first L2 access targeting m after p, m will have been refreshed

in the L2 cache previously. Thus, the access associated to p_j will not experience any direct interference. However, it may still be subject to indirect interference that occured since the last L2 access to m prior to p_j. □

In order to compute the CRPD from accesses classified as an L2 hit by the timing analysis, we have to check every location in F_m^p and L_m^p and observe whether it may result in an L2 miss due to preemption effects. We now bound the indirect effect of preemption for a particular execution trace $s \in \mathcal{S}$.

Lemma 6.9. *The indirect interference in a trace*

$$s = (s_1, \ldots, (p_i, c_i^1, c_i^2), \ldots, (p_j, c_j^1, c_j^2), \ldots, (p', c^1, c^2), \ldots, s_n) \in \mathcal{S}$$

experienced by an L2 access to m at p' due to a preemption at $p = p_i$ is bounded by $L2I_{p'}^p(s)$. Let P be the set of all locations p_j (accessing block m_j) after p_i, $i \leq j$, such that: p' is the first access to m after p_j and p_j is the first access to m_j after p_i.

$$L2I_{p'}^p(s) = \left\{ m_j \;\middle|\; \begin{array}{l} (p_j, c_j^1, c_j^2) \in P \wedge \\ c_j^1(m_j) < W_1 \wedge c_j^1(m_j) + \widehat{ECB}_1^{m_j} \geq W_1 \wedge \\ c_j^2(m_j) > c_j^2(m) \end{array} \right\}. \tag{12}$$

Proof. Indirect interference affecting the access at p' will accumulate only after the last L2 access to m prior to p'. For this reason, the states $(p_j, c_j^1, c_j^2) \in P$ are determined, such that the access from p' will be the first L2 access to m after p_j. Any accesses not associated to states in P will not contribute to the indirect interference. It will occur either after p' or the indirect interference will be eliminated by another access to m prior to p'. Note that an access to a memory block m_j will only contribute to the indirect interference if it results in an L1 hit in the absence of preemptions. Furthermore, m_j is only included in $L2I_{p'}^p$ if its cache age is larger than m in the L2 cache, as otherwise it will not cause m to age in the L2 cache. This condition is only checked for the first access to m_j after p_i, as later accesses will not be affected by the ECBs in the L1 cache. Thus, $L2I_{p'}^p(s)$ is an upper bound on the indirect interference for references resulting in an L2 hit in the absence of preemptions for the particular trace s. □

We write $L2I_{p'}^p(\mathcal{S})$ for $\bigcup_{s \in \mathcal{S}} L2I_{p'}^p(s)$. In Section 8.2, we utilize Lemma 6.9 to compute an upper bound on the aging from indirect interference for L2-UCBs.

7 CRPD Analysis of L1-UCBs

In this section, we will show how an upper bound on the aging from indirect interference for references to L1-UCBs can be computed. The state-of-the-art analysis is pessimistic regarding L1-UCBs, as we have discussed in Section 5.2. We present a novel analysis method for indirect interference on L1-UCBs that eliminates this pessimism. Note that while the previous section operated on concrete cache states and system traces, the analysis uses the efficient abstraction of potential cache states to the maximal block age [14] and thus computes CRPD bounds independent of the particular path taken through the CFG.

To determine the indirect interference for an L1-UCB m at p, we have to determine a safe bound on the aging caused by blocks in $L1I_m^p(\mathcal{S})$. To this end, we construct a DFA [1, p.597ff]. By formulating the problem as a DFA, we can directly solve the overestimation issue of the indirect interference Algorithm 1. The pessimism of the state-of-the-art algorithm originated from the fact that memory references occurring after the analyzed reference are considered to cause indirect

interference. This overestimation is avoided by our approach as the correct ordering of accesses is inherent in the DFA.

We use $\mathcal{P}(\mathbb{M})$ to denote the power set of $\mathbb{M}$. The domain $\mathcal{D}_{Ind}$ of the analysis is the function space of all mappings from a memory block to subsets of memory blocks:

$$\mathcal{D}_{Ind} = \{\mathbb{M} \to \mathcal{P}(\mathbb{M})\}\,. \tag{13}$$

An element $d_1 \in \mathcal{D}_{Ind}$ expresses which blocks may cause indirect interference to m_y on the next access to m_y by the value $d_1(m_y)$. Joining two elements of the domain is performed by taking the union of the interference sets:

$$d_1 \sqcup d_2 = \{(m, d_1(m) \cup d_2(m)) \mid m \in \mathbb{M}\}. \tag{14}$$

The analysis is performed in the backward direction. When formulating the DFA, we use the words *in* (*out*) to refer to *incoming* (*outgoing*) edges of a node in the sense of the regular control-flow.

The data-flow information d_m regarding a block m is transferred over a program location $p \in \mathbb{P}$ using the function $f^p(m, d_m)$, defined in Equation (15), where m_p is the memory block targeted by the reference associated to p.

$$f^p(m, d_m) = \begin{cases} \emptyset & \text{if } m_p = m \\ d_m \cup \{m_p\} & \text{else if } set_2(m_p) = set_2(m) \wedge \\ & \qquad MustAge_1^p(m_p) < W_1 \wedge \\ & \qquad MustAge_1^p(m_p) + \widehat{ECB}_1^{m_p} \geq W_1 \wedge \\ & \qquad MustAge_2^p(m_p) > MustAge_2^p(m) \\ d_m & \text{otherwise} \end{cases} \tag{15}$$

The first case in Equation (15) checks if the location p accesses the analyzed memory block m. If this is the case, we reset the potential for indirect interference to the empty set. This is possible, as the L1 cache is always accessed for every memory access. This means that after an access to the memory block m it is definitely the youngest block in the L1 cache. Thus, further accesses to this block may not result in an L1 miss due to a previous preemption. The indirect interference in the L2 cache need not be transferred for L1-UCBs beyond this point.

The second case checks whether the block m_p may cause indirect interference for m. For m_p to cause indirect interference, it has to be mapped to the same L2 set as m. Directly before the access to m_p it has to be contained in the L1 cache and the timing analysis must consider the access to result in an L1 hit, i.e., the maximal LRU age of m_p must be less than the number of ways in the L1 cache. Additionally, to cause indirect interference, m_p must potentially be evicted from the L1 cache by the ECBs. Finally, the age of m_p must be higher than m in the L2 cache to cause m to age. If these conditions are met, the set d_m is expanded by m_p. Otherwise, d_m is unaffected by the access to m_p.

To propagate information backward over a location p, the function f^p is applied to every memory block, as shown in Equation (16), while the information at an outgoing edge is computed from multiple successor blocks $p' \in succ(p)$ by joining the individual elements, as shown in Equation (17).

$$in_{IndL1}[p] = \{(m, f^p(m, d_m)) \mid (m, d_m) \in out_{IndL1}[p]\} \tag{16}$$

$$out_{IndL1}[p] = \bigsqcup_{p' \in succ(p)} in_{IndL1}[p'] \tag{17}$$

The initial value is the mapping that assigns every block the empty set. The values $in[p]$ and $out[p]$ are computed iteratively for all $p \in \mathbb{P}$ until a fixed point is reached. Termination of the analysis is guaranteed as the transfer function f^p is monotonic and the domain $\mathcal{D}_{Ind}$ is finite. Only a finite number of updates may be performed for each program location, thus ensuring termination.

Theorem 7.1. *Given a preemption that occurs at $p \in \mathbb{P}$: m will be in the L2 cache at the first access to m after p if $CU_2^p(m) + \widehat{ECB}_2^m + |in_{IndL1}[p](m)| < W_2$ holds.*

Proof. The value $CU_2^p(m)$ gives the maximal age of the block m in the L2 cache at the first access to m after p in the absence of preemptions. The direct aging of m due to the preemption at p is limited by $\widehat{ECB}_2^m$. Thus, $CU_2^p(m) + \widehat{ECB}_2^m$ is an upper bound on the L2 age of m including direct preemption effects.

A block m_j may only cause indirect interference to m if $m_j \in L1I_m^p(\mathcal{S})$ (see Lemma 6.2). In case $L1I_m^p(\mathcal{S}) \subseteq in_{IndL1}[p](m)$, the theorem holds due to Lemma 6.2. We show that for $m_j \in L1I_m^p(\mathcal{S}) \setminus in_{IndL1}[p](m)$ the age of m will not increase past $CU_2^p(m)$ due to m_j.

Assume that $\exists\, m_j \in L1I_m^p(\mathcal{S}) \setminus in_{IndL1}[p](m)$. This means that there exists a trace $s \in \mathcal{S}$ where $m_j \in L1I_m^p(s)$, i.e., the conditions listed in Equation (9) hold for m_j on the trace s. Let p_j denote the location where m_j is accessed.

We will now examine the three components of the second case in Equation (15):

(1) $MustAge_1^{p_j}(m_j) < W_1$
(2) $MustAge_1^{p_j}(m_j) + \widehat{ECB}_1^{m_j} \geq W_1$
(3) $MustAge_2^{p_j}(m_j) > MustAge_2^{p_j}(m)$.

As $m_j \notin in_{IndL1}[p](m)$, one of these conditions must be false. If the first part of the condition is false, the access to m_j will not be considered an L1 hit by the cache analysis and already contributes to $CU_2^p(m)$. m_j is thus not a source of indirect interference in this scenario.

The second part of the condition is directly implied by the fact that $m_j \in L1I_m^p(\mathcal{S})$ as $MustAge_1^{p_j}(m_j)$ is an upper bound on the L1 cache age of m_j at p_j on any trace. It will never be false given $m_j \in L1I_m^p(\mathcal{S})$.

If the third part of the condition is false, there exists another trace $s' \in \mathcal{S}, s' \neq s$ that contains the same references to m_j and m as s, where m is older than m_j in the L2 cache. In this situation, an L1 cache miss to m_j will not cause m to age in the L2 cache as m is already older than m_j. The potential interference of m_j toward m is already captured in the value $CU_2^p(m)$.

Hence, the interference from m_j is accounted for either in $CU_2^p(m)$ or by $m_j \in in_{IndL1}[p](m)$ Consequently, there is no memory block $m_j \in L1I_m^p(\mathcal{S}) \setminus in_{IndL1}[p](m)$, that could increase the age of m beyond $CU_2^p(m) + \widehat{ECB}_2^m + |in_{IndL1}[p](m)|$. We conclude that $CU_2^p(m) + \widehat{ECB}_2^m + |in_{IndL1}[p](m)| < W_2$ is a sufficient condition to guarantee that m is contained in the L2 cache at the first reference to m after the preemption at p. □

We modify Equations (2) and (3) to consider the tighter bound on indirect interference as follows. Instead of the value $ColInd_m^p$ given by Algorithm 2, the value $in_{IndL1}[p](m)$ is used to determine whether a memory block may be evicted from the L2 cache.

$$\delta_{L1}^p = d_{L1} \cdot \left| \left\{ m \in UCB_1^p \;\middle|\; CU_1^p(m) + \widehat{ECB}_1^m \geq W_1 \wedge CU_2^p(m) + \widehat{ECB}_2^m + |in_{IndL1}[p](m)| < W_2 \right\} \right|, \tag{18}$$

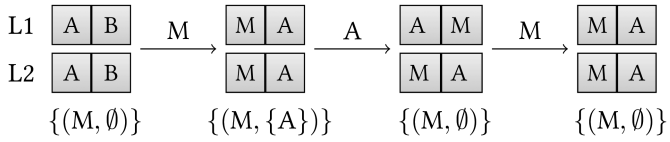


Fig. 6. Example for the indirect interference DFA for L1 cache hits.

$$\delta^p_{L1+L2} = (d_{L1} + d_{L2}) \cdot \left| \left\{ m \in UCB^p_1 \;\middle|\; CU^p_1(m) + \widehat{ECB}^m_1 \geq W_1 \wedge CU^p_2(m) + \widehat{ECB}^m_2 + |in_{IndL1}[p](m)| \geq W_2 \right\} \right|. \tag{19}$$

It follows from Theorem 7.1 that $\delta^p_{L1} + \delta^p_{L1+L2}$, as defined in Equations (18) and (19), is a safe bound on the CRPD resulting from L1 hits being degraded to L2 accesses due to a preemption at p.

Figure 6 illustrates the execution of the DFA on the scenario used in Figure 1, which introduced indirect interference. As before, we consider a preemption with a single interfering cache block. The data-flow information is annotated below the cache states, showing the information concerning the cache block M. The information is propagated backward along the edges, beginning at the rightmost state with the empty set. As it is propagated to the left, the indirect interference information for M remains the empty set. This corresponds to the first case of the transfer Equation (15). In the next propagation step, the set of cache blocks potentially causing indirect interference is updated to {A}. This update occurs because the second case of Equation (15) applies: A is older in the L2 cache than M, and A may be evicted from the L1 cache by a preemption at this stage. The access to A can cause indirect interference and is consequently added to the data-flow information. When propagating the information to the leftmost state, block A is removed again from the indirect interference set, since the access to M refreshes it in the L1 cache. A preemption at this point would not create indirect interference from A toward M, because the first L1 access to M happens before the access to A.

8 CRPD Analysis for L2 UCBs

In this section, we will show how the preemption penalty stemming from L2-UCBs can be computed, even if there are multiple references that potentially cause the first L2 access to a particular cache block after the preemption and accesses may not reach the L2 cache in every situation.

First, we construct a backward DFA that collects all reachable references that result in an L2 hit for every program location $p \in \mathbb{P}$ in Section 8.1. Then a bound on indirect interference for L2-UCBs is introduced in Section 8.2. Finally, in Section 8.3, an algorithm to compute the CRPD from all references to L2-UCBs is presented.

8.1 Determining L2 Access Locations

To analyze how much CRPD can originate from references that are classified as L2 hits in the absence of preemption, we have to determine where these references are located. We solve this problem by creating a backward DFA.

Only the first access to a block in the L2 cache after a preemption experiences the direct interference due to ECBs [36]. However, there may be multiple references that potentially cause the first access to the L2 cache. To safely estimate the CRPD, we have to consider all candidates for the first access. For this reason, we define the set *Loc* in Equation (20).

$$Loc = \{(f, l) \mid f, l \subseteq \mathbb{P}\}. \tag{20}$$

Loc contains pairs of program location subsets. The first element f of a pair $(f, l) \in Loc$ corresponds to the possible first L2 access locations after a preemption, while l contains all locations that may issue the second or later access to the L2 cache. Joining two tuples (f_1, l_1) and (f_2, l_2) is realized by the union of the two sets in both tuples:

$$(f_1, l_1) \sqcup (f_2, l_2) = (f_1 \cup f_2, l_1 \cup l_2). \tag{21}$$

In order to keep track of all L2 hit locations for all memory blocks, we define the DFA domain $\mathcal{D}_{FL}$ as the function space of mappings from every memory block to an element of *Loc*:

$$\mathcal{D}_{FL} = \{\mathbb{M} \rightarrow Loc\}. \tag{22}$$

When a reference resulting in an L2 hit to $m \in \mathbb{M}$ is encountered during the DFA, the mapped element $d(m), d \in \mathcal{D}_{FL}$ is modified. The function $t_m^p((f, l))$, defined in Equation (23), updates the tuple (f, l) according to the access at the location p. We write $CAC(p)$ to denote the CAC of the reference associated to p. A CAC value of A shows that the second level cache is always accessed, i.e., it is an L1 miss; a CAC value of N signifies that the L2 cache is never accessed, while the CAC value U stands for an unknown access behavior.

$$t_m^p((f, l)) = \begin{cases} (f, l) & \text{if } m_p \neq m \vee CAC(p) = N \\ (\emptyset, f \cup l) & \text{else if } CAC(p) = A \wedge MustAge_2^p(m) \geq W_2 \\ (f, f \cup l) & \text{else if } CAC(p) = U \wedge MustAge_2^p(m) \geq W_2 \\ (\{p\}, f \cup l) & \text{else if } CAC(p) = A \wedge MustAge_2^p(m) < W_2 \\ (f \cup \{p\}, f \cup l) & \text{else if } CAC(p) = U \wedge MustAge_2^p(m) < W_2 \end{cases} \tag{23}$$

In the first case, the mapped value is not changed. t_m^p leaves (f, l) unchanged under two conditions: (1) the reference associated to p targets a different memory block than m, or (2) the access never reaches the L2 cache.

The next two cases match if the reference associated to p targets the block m and will not result in a definite L2 hit. The difference between these two cases is whether the access will always reach the L2 cache or whether it might be processed at the L1 cache as a hit. If the access will definitely miss in the L1 cache and always reach the L2 cache, the function t_m^p will set the first access locations to the empty set and add the previous set of first access locations to the set of later accesses. This is done as no L2 hits may be affected by direct interference. The direct interference will be eliminated by the currently analyzed L2 miss.

Otherwise, if the access may hit in the L1 cache, we have to consider both situations. Either the access results in an L1 hit or an L1 miss. For an L1 hit, we leave the sets of first and later L2 hits unchanged (f, l). In the situation of the L1 miss, we get the same result as in the previous case $(\emptyset, f \cup l)$. Joining these two potential results gives us $(f, l) \sqcup (\emptyset, f \cup l) = (f, f \cup l)$.

The fourth case considers accesses that will definitely miss in the L1 cache and hit in the L2 cache. In this situation, the current location p becomes the location of the first L2 hit and all previous first hits are added to the set l, which contains those accesses that may only be affected by indirect interference.

In case the access may not reach the L2 cache but will result in an L2 hit if it does, we have to consider multiple scenarios. The first scenario is that the access reaches the L2 cache, this results in the tuple $(\{p\}, f \cup l)$ as in the previous case. In the second scenario, the access does not reach the L2 cache; the tuple of hit locations is thus not modified and remains (f, l). Joining the results of these scenarios gives us $(f \cup \{p\}, f \cup l)$.

The data-flow information is transferred over a location $p \in \mathbb{P}$ by applying the function t_m^p to every value (m, d_m) contained in the outgoing information:

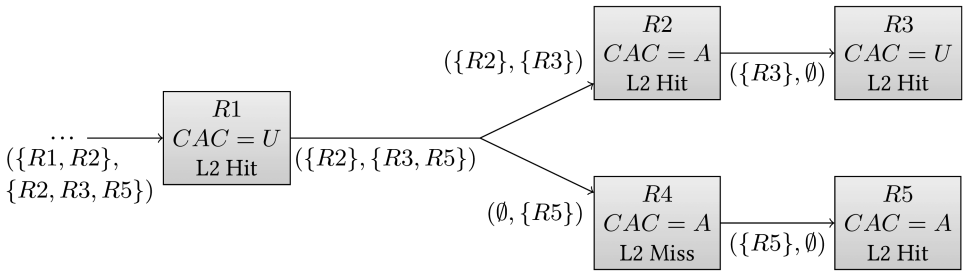


Fig. 7. Example for the first access location DFA on a simplified CFG.

$$in_{FL}[p] = \{(m, t^p_m(d_m)) \mid (m, d_m) \in out_{FL}[p]\}. \tag{24}$$

To compute the data-flow information at the outgoing edge of a node, the information from all successor nodes are joined (25), where two mappings are joined according to Equation (26).

$$out_{FL}[p] = \bigsqcup_{p' \in succ(p)} in_{FL}[p'], \tag{25}$$

$$d_1 \sqcup d_2 = \{(m, d_1(m) \sqcup d_2(m)) \mid m \in \mathbb{M}\}. \tag{26}$$

The initial information for each node is the empty mapping, which assigns each cache block a pair of empty sets. Using this DFA, it is possible to determine for each program location p which references to a memory block are L2 hits and can suffer from direct and/or indirect interference.

Theorem 8.1. *For a memory block m, the least fixed point (f, l) of $in_{FL}[p](m)$, computed by the data-flow analysis, contains all locations with references to m, that may contribute to the CRPD by being degraded from an L2 hit to an L2 miss. References associated to locations $r \in f$, may be subject to direct and indirect interference; references associated to locations $r \in l$ may be subject only to indirect interference.*

Proof. F^p_m contains all potential first L2 accesses to m after p that may be subject to direct and indirect interference (Lemma 6.6). For an access to contribute to the CRPD, it has to be considered a hit in the timing analysis. Thus, only references $r \in F^p_m$ with $MustAge^r_2(m) < W_2$ have to be considered when computing the CRPD.

Assume that $r \in F^p_m$. We know that there exists a trace $s \in \mathcal{S}$ that traverses the state (p, c^1_p, c^2_p) and continues to location (r, c^1_r, c^2_r), where the access results in an L2 hit. Hence, it is true that $c^1_r(m) = \infty \wedge c^2_r(m) < W_2$. If $MustAge^p_2(m)$ were to be greater or equal to W_2, the access would not be considered an L2 hit by the timing analysis and could not contribute to the CRPD. Thus, m is added to the set of first accesses when propagating the data-flow information over (r, c^1_r, c^2_r), as the fourth or fifth case of Equation (23) applies. Furthermore, as the access associated to r is the first L2 access to m, there is no intermediate state (q, c^1_q, c^2_q), that performs an L2 access to m that always reaches the L2 cache. Hence, for all intermediate states the first, third, or fifth case of the propagation function t^p_m applies. It follows that r is not removed from the set of first hit locations. As no elements are removed from the set on a join operation it follows that $r \in f$. The proof for references $r \in L^p_m$ is analogous, using Lemma 6.8 and the addition of an intermediate state that causes r to be added to the set of later L2 hits. □

Thus, based on the value of $in_{FL}[p]$, we can compute a safe estimate of the CRPD for a preemption happening at p.

We demonstrate the functionality of the DFA in Figure 7 on a CFG, which is simplified to only show references to a single cache block. Data-flow information for the targeted cache block is

annotated at the edges. The accesses in the CFG are named $R1$ through $R5$, and their respective access and hit classification are listed in the corresponding node.

The DFA is performed in the backward direction, meaning that the information is propagated from $R3$ and $R5$ toward $R1$. On the edge toward $R3$, $R3$ is the only potential first L2 cache hit for the analyzed cache block, with no additional L2 cache hits reachable from this location. Thus, the data-flow information is equal to $(\{R3\}, \emptyset)$. The same applies for the edge leading towards $R5$.

When the data-flow information is propagated over $R2$, $R3$ is replaced as the first L2 hit and is moved to the set of later L2 hits, while $R2$ is the new first hit: $(\{R2\}, \{R3\})$. This update corresponds to the fourth case in equation (23). Since the access $R4$ can result in a cache miss, it is not considered as a potential source of delay in the CRPD computation. However, the reference $R5$ is moved to the second set, which contains L2 cache hits unaffected by direct interference, as the CAC of $R4$ is A, which means that $R4$ will definitely miss in the L1 cache and refresh the cache block in the L2 cache. A preemption before $R4$ will not thus not create direct interference for $R5$. Case 2 of Equation (23) covers this scenario, resulting in: $(\emptyset, \{R5\})$.

The information of the upper and lower branch are joined by performing the set union, as defined in Equation (21). Finally, the tuple $(\{R2\}, \{R3, R5\})$ is propagated backward over $R1$. As $R1$ will result in an L2 hit, but the CAC is U, the tuple is updated to $(\{R1, R2\}, \{R2, R3, R5\})$, as defined in the fifth case of Equation (23). This means that a preemption at this point can impact $R1$ and $R2$ directly, while $R2$, $R3$, and $R5$ can only be affected by indirect interference.

8.2 Indirect Interference for L2 Hits

The analysis of indirect interference on L1-UCBs as presented in the previous Section 7 is unsuited to compute the indirect interference for references that are considered L2 hits. This results from the fact that it is not sufficient to only check the preemption effects on the first L2 access after the preemption for L2-UCBs. Even after multiple accesses to a memory block m, the indirect preemption effects can evict m from the L2 cache and thus cause a cache miss for the next access. For this reason, we have to compute an upper bound on the indirect interference for every reachable reference targeting m that results in an L2 hit. As we have determined the location of these references in the previous Section 8.1, we can now focus on determining an upper bound on the indirect interference.

Determining such an upper bound can be approached from different perspectives. One approach would be to modify the DFA from Section 7 to operate not on memory blocks but references and adjust the transfer function accordingly. This would allow us to compute an upper bound on the indirect interference for each individual reference. However, this approach scales poorly as the number of reachable L2 hits can be significantly larger than the number of cache blocks (e.g., due to virtual loop unrolling) and the bound has to be computed for every reachable reference that results in an L2 hit and every possible preemption location. The required computational effort of this approach would thus be much larger than for the DFA from Section 7.

We solve this problem by determining an upper bound on the indirect interference for every reference resulting in an L2 hit, irrespective of the preemption location. This approach can be implemented using an algorithm that iterates over every program location $p \in \mathbb{P}$ once and accumulates all potential indirect interference effects for every potential first L2 hit. For each location $p \in \mathbb{P}$, we only have to consider the first L2 hits reachable from p in this algorithm, because for later L2 hits the indirect interference will have been removed by a previous L2 access.

Algorithm 4 computes the indirect interference on L2 hits and starts by setting the indirect interference to the empty set for all references (lines 1–3). Then, for every program location it is determined whether the access on that location can cause indirect interference to another block (lines 5–8). To cause indirect interference, the blocks must be mapped to the same L2 cache set

ALGORITHM 4: Indirect Interference on L2 Hits

Result: The indirect interference for references that result in L2 hits.

```
1  for p' ∈ ℙ do
2  |  IndL2_{p'} ← ∅
3  end
4  for p ∈ ℙ do
5  |  for m ∈ 𝕄 do
6  |  |  if m ≠ m_p ∧ set_2(m) = set_2(m_p)∧
7  |  |  |  MustAge_1^p(m_p) < W_1 ∧ MustAge_1^p(m_p) + ÊCB_1^{m_p} ≥ W_1∧
8  |  |  |  MustAge_2^p(m_p) > MustAge_2^p(m) then
9  |  |  |  (f, l) ← in_FL[p](m)
10 |  |  |  for p' ∈ f do
11 |  |  |  |  IndL2_{p'} ← IndL2_{p'} ∪ {m_p}
12 |  |  |  end
13 |  |  end
14 |  end
15 end
```

(line 6). Furthermore, it is checked whether m_P is contained in the L1 cache, is considered a definite hit by the *must* age analysis, and may potentially be evicted due to the ECBs (line 7). The block is only considered to cause indirect interference if the L2 *must* age of m_p is greater to that of m (line 8). If these conditions hold, the block m_p is added to the indirect interference estimate for every reachable first L2 hit to m (lines 9–12). The result of the algorithm is the set $IndL2_{p'}$ for each $p' \in \mathbb{P}$ that is considered an L2 hit. It contains all memory blocks that can potentially cause indirect interference for the reference at p'.

Theorem 8.2. *Given a preemption that occurs at $p \in \mathbb{P}$:*

For a reference at location $p' \in f, (f, l) = in_{FL}[p](m)$, m will be contained in the L2 cache at p' if $MustAge_2^{p'}(m) + \widehat{ECB}_2^m + |IndL2_{p'}| < W_2$ holds.

For a reference at location $p' \in l, (f, l) = in_{FL}[p](m)$, m will be contained in the L2 cache at p' if $MustAge_2^{p'}(m) + |IndL2_{p'}| < W_2$ holds.

Proof. The proof is very similar to the proof of Theorem 7.1.

We show that it holds for $p' \in f$. The value $MustAge_2^{p'}(m) + \widehat{ECB}_2^m$ is an upper bound on the cache age of m solely based on the direct effect of preemption. A block m_h may only cause indirect interference to m if $m_h \in L2I_{p'}^p(\mathcal{S})$ (see Lemma 6.9) and the access to m_h is considered an L1 hit in the absence of preemptions. In case $L2I_{p'}^p(\mathcal{S}) \subseteq IndL2_{p'}$, the theorem holds due to Lemma 6.9.

For $m_h \in L2I_{p'}^p(\mathcal{S}) \setminus IndL2_{p'}$, the age of m will not increase over $MustAge_2^{p'}(m)$ due to m_h. For m_h to be contained in $L2I_{p'}^p(\mathcal{S}) \setminus IndL2_{p'}$, there must be a trace s, where on accessing m_h, the L2 age of m_h exceeds the L2 age of m (see the final condition in Equation (12)). m_h will not be added to $IndL2_{p'}$ if at the location p_h where m_h is referenced $MustAge_2^{p_h}(m_h) \leq MustAge_2^{p_h}(m)$ holds. Thus, there exists a different trace $s' \in \mathcal{S}, s' \neq s$, where m is older in the L2 cache than m_h. On the trace s', m_h contributes to the *must* age of m. This interference is captured by $MustAge_2^{p'}(m)$. Consequently, $MustAge_2^{p'}(m) + \widehat{ECB}_2^m + |IndL2_{p'}| < W_2$ is a sufficient condition for the access at p', targeting m, to result in an L2 hit.

The proof for the second part of the theorem for $p' \in l$ is analogous. □

ALGORITHM 5: CRPD from L2 Cache Misses due to a Preemption at p

Result: The delay suffered from L2 hits that are degraded to L2 misses due to preemption.

1 $\delta^p_{L2} \leftarrow 0$
2 **for** $m \in \widehat{UCB}^p_2$ **do**
3 $\quad \delta_m \leftarrow 0$
4 $\quad (f, l) \leftarrow in_{FL}[p](m)$
5 $\quad$ **if** $\exists\, p' \in f : MustAge^{p'}_2(m) + \widehat{ECB}^m_2 + \left|IndL2_{p'}\right| \geq W_2$ **then**
6 $\quad\quad \delta_m \leftarrow \delta_m + 1$
7 $\quad$ **end**
8 $\quad$ **for** $p' \in l$ **do**
9 $\quad\quad$ **if** $MustAge^{p'}_2(m) + \left|IndL2_{p'}\right| \geq W_2$ **then**
10 $\quad\quad\quad \delta_m \leftarrow \delta_m + 1$
11 $\quad\quad$ **end**
12 $\quad$ **end**
13 $\quad \delta^p_{L2} \leftarrow \delta^p_{L2} + (\delta_m \cdot d_{L2})$
14 **end**

8.3 CRPD Computation

The locations of references that may be impacted by direct and indirect effects or only indirect effects can be determined using the DFA from Section 8.1. Using the analysis of indirect interference from the previous Section 8.2, it is now possible to compute a bound on the CRPD. Taking the information from these two components, we construct Algorithm 5 to compute a safe estimate on the CRPD from L2 hits that does not suffer from the safety issues discussed in Section 5. In contrast to Algorithm 3, this Algorithm is capable of dealing with situations where multiple references to the same block may be the first reference to be executed after a preemption. We denote the CRPD contribution of L2 hits being degraded to L2 misses due to a preemption at p by δ^p_{L2}.

Algorithm 5 iterates over all memory blocks that are L2-UCBs or may become L2-UCBs in a location reachable from P (lines 2–14). This is done as these are exactly those blocks that may contribute to the CRPD due to a degradation from an L2 hit to an L2 miss. The CRPD contribution from each $m \in \widehat{UCB}^p_2$ is denoted by δ_m in Algorithm 5.

Instead of determining a singular reference to m that may be executed first after p, we utilize the DFA from Section 8.1, to determine all locations which are reachable from p and access m resulting in an L2 hit (line 4). The set f contains references that may be subject to direct interference from ECB, due to the preemption at p. Notably this includes also those references which occur after another reference to m that has a CAC of U (see Equation (23)). The set l contains references that may be subject to eviction solely due to the indirect effect. Note that f and l are not necessarily disjoint.

In lines 5–7, the algorithm checks whether there is a reference to m in f that may be evicted by the collaboration of direct and indirect preemption effects. m may be evicted from the L2 cache at $p' \in f$ if its maximal LRU age at p' plus the direct interferences of $\widehat{ECB}^m_2$ ECB and the indirect interference is greater or equal to the number of ways in the L2 cache (see Theorem 8.2). As there is at most one reference that can be affected by the direct effects of the preemption, it is sufficient to check for the existence of such a reference. After the first L2 access to m after a preemption, the block m will be refreshed in the L2 cache. Thus, it will be younger than all ECB. Consequently, future accesses to m are no longer affected by those ECBs. If there exists a potential first access that is degraded to an L2 miss due to the preemption, δ_m is increased by 1.

The loop in lines 8–12 checks all references in the set l. The access may result in an L2 miss if the maximal LRU age of m plus the indirect interference exceeds the number of L2 ways. For each potential L2 miss, the variable δ_m is incremented by 1. The total CRPD from L2 cache hits is then given by the sum of $\delta^p_{L2} = \sum_{m \in \widehat{UCB^p_2}} \delta_m \cdot d_{L2}$ (line 13).

An upper bound on the CRPD can thus be computed by taking the maximal sum of $\delta^p_{L1} + \delta^p_{L1+L2} + \delta^p_{L2}$ for any program location $p \in \mathbb{P}$:

$$\delta = \max_{p \in \mathbb{P}} \left(\delta^p_{L1} + \delta^p_{L1+L2} + \delta^p_{L2} \right). \tag{27}$$

As the value δ varies based on the preempted task and preempting tasks, we denote the CRPD of task τ being preempted by φ and all higher priority tasks $\psi \in hp(\varphi)$ as $\delta_{\tau,\varphi}$. The WCRT of a task τ can be computed iteratively using Equation (28) [9]. If the value $WCRT^{i+1}_\tau$ exceeds the deadline of τ, the iteration can be stopped as the task, and by extension the whole system, is not schedulable.

$$WCRT^{i+1}_\tau = WCET_\tau + \sum_{\varphi \in hp(\tau)} \left\lceil \frac{WCRT^i_\tau}{Period(\varphi)} \right\rceil \cdot \left(WCET_\varphi + \delta_{\tau,\varphi} \right). \tag{28}$$

9 Evaluation

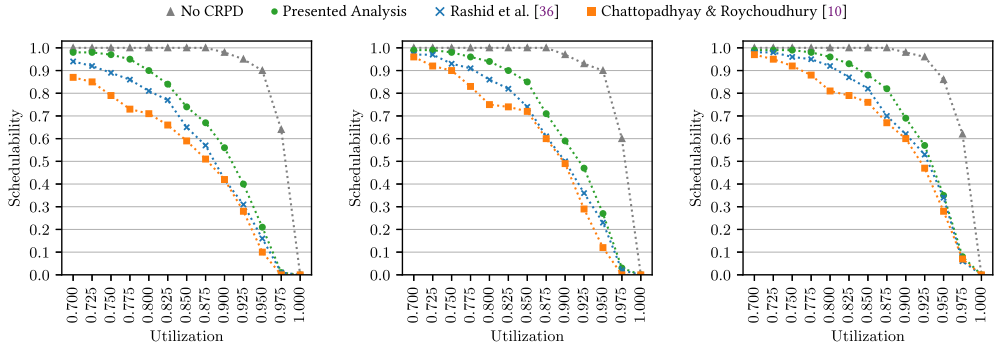
To evaluate the performance of the presented analysis, we integrated it in a WCET-aware C Compiler [13]. We compare the performance of the presented analysis to the two other approaches for non-inclusive caches: Chattopadhyay and Roychoudhury [10], which we refer to as the baseline analysis, as well as the state-of-the-art from Rashid et al. [36].

We modified the state-of-the-art analysis to be able to handle situations with multiple possible first accesses. This was done as every analyzed task set contained at least one situation in which there are multiple possible first references or first L2 accesses. As the state-of-the-art [36] approach does not consider such situations, the analysis failed, preventing us from creating a meaningful comparison to the presented analysis. We used the presented Algorithm 5 instead of Algorithm 3 to compute the contribution of L2 hits to the CRPD. We adapted Algorithm 5 for this application to use the indirect interference values as computed by the state-of-the-art. Furthermore, we modified Algorithm 1 to iterate over all potential first accesses (cf. line 6 in Algorithm 1) and accumulate the results for every potential first access. All other details of the approach were implemented as defined in [36]. These adjustments allowed us to compare the performance of the state-of-the-art approach to the presented analysis, even in the presence of multiple first accesses to a particular block (see Section 5.3.1). Note that even with these modifications, the analysis [36] is not safe as the issue described in Section 5.1 persists: the analysis does not account for the CAC of L2 cache accesses and can underestimate the indirect interference.

The target architecture in our evaluation consists of a single core, with a two-level instruction cache hierarchy. The core features a 3-stage in-order pipeline using the ARMv4T instruction set. Both cache levels use the LRU replacement policy and have a cache block size of 64 bytes. The access timings are 1 cycle for an L1 hit, 10 cycles for an L2 hit and 100 cycles for an L2 miss. The same timings were used in the evaluation of the state-of-the-art approach [36] and are typical timings according to the reference manual of the ARM PL310 cache controller [27]. We abbreviate this timing configuration as the 1/10/100 timing.

To increase the analysis precision, we activated virtual-inlining and virtual-unrolling with a maximal context number of 3. This means, that the analyzer differentiates between the first, second, and all following iterations of a loop.

As the benchmarking programs, we used tasks from the MRTC benchmark suite [19], which are also used in other literature for two-level CRPD analyses [10, 36, 47]. We randomly generated 100



(a) 1-way L1, 2-way L2 (b) 2-way L1, 4-way L2 (c) 4-way L1, 8-way L2

Fig. 8. Schedulability of 100 analyzed systems with 10 tasks per core with 1 KB L1 and 4 KB L2 cache. The y-axis shows the ratio of schedulable systems for different utilization values on the x-axis. (a) shows the results for 1-way L1 and 2-way L2 cache; (b) for 2-way L1 and 4-way L2 cache; (c) for 4-way L1 and 8-way L2 cache.

task sets containing 10 tasks. The task periods were generated using the UUnifast approach [8]. We used *rate-monotonic* scheduling to assign a priority to every task [30].

The performance of the different approaches was measured using the schedulability fraction, which describes how many of the analyzed task sets were schedulable according to Equation (28). We present our results in Figures 8–10. On the x-axis the utilization value u is shown. It is computed as $u = \sum_{\tau \in T}(WCET(\tau)/Period(\tau))$, where $WCET(\tau)$ is the WCET of τ and $Period(\tau)$ is its period. We evaluated utilization values between 0.7 and 1.0. The y-axis shows the fraction of schedulable task sets. The schedulability without considering the CRPD is depicted by gray triangles, the presented analysis as green dots, the state-of-the-art [36] as blue crosses, and the baseline [10] as orange squares.

In Figure 8, the evaluation results for a 1 KB L1 and 4 KB L2 cache are shown. We evaluated three different settings for the cache associativity. Figure 8(a) corresponds to a direct mapped L1 cache and a 2-way set associative L2 cache. In Figure 8(b), the associativity is set to 2-ways for the L1 and 4-ways for the L2 cache. The highest evaluated associativity is shown in Figure 8(c), where the L1 cache is 4-way associative and the L2 cache is 8-way associative. We call these parameter settings the low, medium, and high associativity configuration. The gray markers show the theoretical upper limit for the schedulability, if no CRPD would occur. The baseline analysis [10], shown in orange, produced the lowest schedulability ratio of the three approaches. We use percentage points (pp) to measure the increase in schedulability. For example, increasing the number of schedulable systems from 40 to 50, out of the 100 total systems, corresponds to a 10 percentage point increase.

The state-of-the-art approach [36], shown in blue, improved the average schedulability by 6.2 pp, 4.6 pp, and 4.3 pp for the low, medium, and high associativity setting over the baseline.

Additionally, it can be seen that the presented analysis yielded significant improvements over the state-of-the-art. We observed an average increase in schedulability of 6.9 pp, 5.8 pp, and 3.8 pp for the low, medium, and high associativity setting over the state-of-the-art. The maximal increase in task set schedulability lies between 11 pp and 14 pp depending on the cache associativity. The largest improvement of 14 pp was observed at $u = 0.9$ with low associativity, as seen in Figure 8(a). Using the state-of-the-art analysis, 42 systems are schedulable, while 56 systems are schedulable when computing the CRPD using the presented analysis.

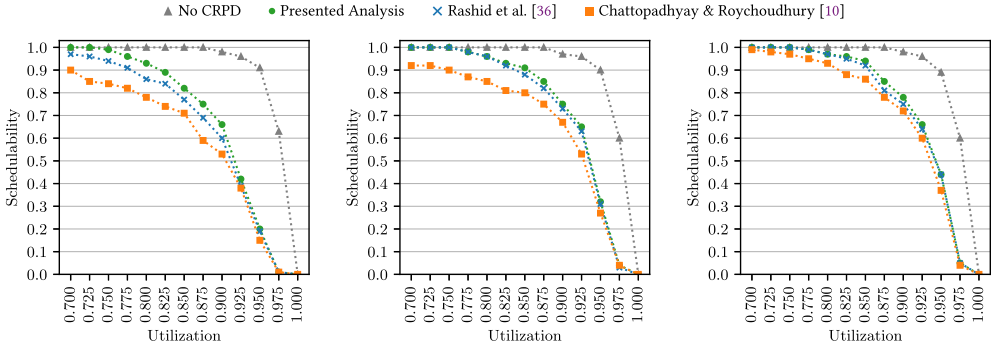
Table 1 shows the decrease of the CRPD values compared with the state-of-the-art. Note in particular that the minimal reduction for the medium associativity is negative. This means that

Table 1. CRPD Reduction Over the State-of-the-Art for 1KB L1 / 4KB L2 Caches

Associativity (L1 / L2)	Maximal Reduction	Average Reduction	Minimal Reduction
1 / 2	75.0%	6.7%	0%
2 / 4	90.9%	6.2%	−5.5%
4 / 8	84.2%	3.4%	0%

Table 2. CRPD Reduction Over the State-of-the-Art for 2KB L1 / 8KB L2 Caches

Associativity (L1 / L2)	Maximal Reduction	Average Reduction	Minimal Reduction
1 / 2	82.8%	2.7%	0%
2 / 4	76.5%	2.3%	−3.9%
4 / 8	71.7%	1.7%	−0.7%



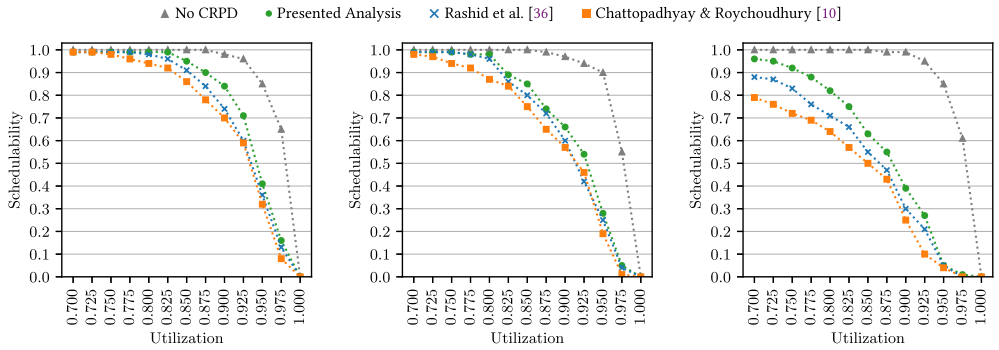
(a) 1-way L1, 2-way L2 (b) 2-way L1, 4-way L2 (c) 4-way L1, 8-way L2

Fig. 9. Schedulability of 100 analyzed systems with 10 tasks per core with 2 KB L1 and 8 KB L2 cache. The y-axis shows the ratio of schedulable systems for different utilization values on the x-axis. (a) shows the results for 1-way L1 and 2-way L2 cache; (b) for 2-way L1 and 4-way L2 cache; (c) for 4-way L1 and 8-way L2 cache.

the CRPD value computed by the presented analysis was 5.5% higher than the value given by the state-of-the-art method. This increase occurred in a system where the task *adpcm* is preempted by the task *bs*. The CRPD increased from 1629 to 1719 cycles. The increased CRPD value could be due to the unsafety of the state-of-the-art, but it may also be the case that the presented analysis is slightly more pessimistic in this particular situation.

In Figure 9, the evaluation results for a 2 KB L1 cache and 8 KB L2 cache are shown. As for the smaller cache size, we analyzed three different associativity settings in Figure 9(a)–9(c). The state-of-the-art yielded on average 6.4 pp / 7.2 pp / 3.5 pp higher schedulability over the baseline for low / medium / high associativity. In contrast to the smaller cache size, the gap between the presented analysis and the state-of-the-art is smaller. An average improvement of 3.8 pp / 1 pp / 0.9 pp for low / medium / high associativity was observed. The maximal improvement of the presented analysis over the state-of-the-art was 7 pp / 3 pp / 4 pp. Hence, we conclude that in this larger cache configuration, the overestimation of the indirect interference in the state-of-the-art is less impactful on the schedulability than for smaller caches.

Table 2 shows the CRPD reduction using the presented analysis over the state-of-the-art for the 1KB L1 / 8 KB L2 cache configuration. The presented analysis achieved an average CRPD reduction of 2.7% / 2.3% / 1.7% over the patched, but unsafe, state-of-the-art analysis. As also visible in Table 1,



(a) 1/10/50 access timing (b) 1/50/100 access timing (c) 1/10/200 access timing

Fig. 10. Schedulability of 100 analyzed systems with 10 tasks per core for different L1 hit / L2 hit / L2 miss timings. (a) shows the results for 1/10/50 cycles; (b) for 1/50/100 cycles; (c) for 1/10/200 cycles.

there are situations in which the presented analysis yields a larger CRPD value compared with the state-of-the-art. The maximal increase, from 4572 to 4752 cycles, was realized in a system where the task *cnt* is preempted by the task *bsort100*.

We note that, for all analyzed system, the presented analysis dominated the state-of-the-art. Even though the CRPD estimate was slightly increased for some task combinations, there was no system which the state-of-the-art classified as schedulable, which was unschedulable using the presented analysis.

Comparing the cache size and associativity configurations using the presented analysis, we observe that the small cache with low associativity yielded the lowest schedulability of an average 63.2%, while the highest average schedulability occurred for the larger cache with high associativity at an average of 74.2%.

We also evaluated a cache size configuration of 4 KB L1 and 16 KB L2 cache. In this configuration, the performance difference between the three approaches diminished. At most, there was a 2 pp improvement of the presented analysis compared with the state-of-the-art. As the cache size increases, the code of the benchmarks fits more easily into the cache causing fewer conflicts. Thus, the precision difference in the analysis methods plays a less important role in system schedulability. On average, the total code size of the task sets was 26.3 KB.

In addition to experiments on the cache size, we explored the impact of the cache access timings. We show the results for a 1 KB L1 and 4 KB L2 cache with 2-way / 4-way associativity in Figure 10. A timing configuration of 10 cycles for an L2 hit delay and 50 cycle L2 miss delay is evaluated in Figure 10(a). Figure 10(b) shows a 50 cycle L2 hit and 100 cycle L2 miss configuration. The final timing evaluation in Figure 10(c) was made with a 10 cycles L2 hit and 200 cycle L2 miss setting.

For these timing configurations, the average improvement in schedulability by the presented analysis over the state-of-the-art was 3.3 pp, 2.6 pp, and 6.8 pp, respectively. Recall, that the improvement for the default 1/10/100 timing was 5.8 pp. For the two timing configurations 1/10/50 and 1/50/100, the difference between an L2 hit and L2 miss is smaller than for the default 1/10/100 timing. It is also for these combinations, that the average improvement was reduced. Whereas, for the slower L2 miss timing of 200 cycles, the average improvement increased from 5.8 pp to 6.8 pp. This observation is congruent with the optimization we made to the state-of-the-art regarding indirect interference. Indirect interference causes L2 hits to be degraded to L2 misses, thus an improvement in the estimation of indirect interferences will scale with the timing ratio between an L2 hit and an L2 miss.

Table 3. Average Runtime of a CRPD Analysis for a System Containing 10 Tasks in Seconds

Size (L1 / L2)	Associativity (L1 / L2)	Baseline [10]	SotA [36]	Presented Analysis
1 KB / 4 KB	1 / 2	12.2 s	70.7 s	7.5 s
	2 / 4	11.2 s	82.0 s	7.7 s
	4 / 8	11.9 s	92.4 s	7.7 s
2 KB / 8 KB	1 / 2	10.7 s	76.0 s	8.2 s
	2 / 4	9.6 s	82.5 s	8.4 s
	4 / 8	9.7 s	90.4 s	8.3 s

To compare the required computational effort, we measured the time required to analyze each system with all three analysis approaches. We performed the evaluations on an Intel Xeon Server with 48 cores running at 3.2 GHz. Every analysis was limited to utilize only a single core. To increase the performance, the analysis could be performed in parallel for every task, as it only requires knowledge of the number of ECBs from preempting tasks for each cache set.

The results are shown in Table 3. Each measurement corresponds to one CRPD analysis of a system independent of the other analyses. The average runtime of the presented analysis is comparable to the baseline approach of Chattopadhyay and Roychoudhury [10]. The baseline analysis required, on average, between 9.6 s and 12.2 s, while the presented analysis took, on average, between 7.5 s and 8.4 s. We call these runtimes comparable, as the relatively small difference in absolute runtime may depend on technical implementation details.

In contrast, the state-of-the-art analysis required substantially more time to complete. The average overhead lies between 9.3× and 12.0× compared with the presented analysis. The main contributors to the runtime of the state-of-the-art analysis were determining the first access to a memory block and collecting all possible locations between the analyzed location and the potential first accesses to that memory block, i.e., the function $GetProgramPoints(P, FA^P_{m_y})$ (see lines 3–4, Algorithm 2). The presented analysis does not contain this bottleneck, as the set returned by $GetProgramPoints(P, FA^P_{m_y})$ is never explicitly determined. Instead, by performing a data-flow analysis to determine potential indirect interference, the relevant locations are considered implicitly by propagating the data-flow information over these locations.

10 Conclusion

In this article, we demonstrate that the state-of-the-art [36] approach for CRPD analysis in two-level non-inclusive caches is flawed. To provide a solid foundation for new analyses, we introduce the concrete semantics of CRPD in two-level non-inclusive caches. Using the concrete semantics, it is possible to argue about the impact of preemption effects at the level of program traces.

Upon this foundation, we construct a novel CRPD analysis for two-level non-inclusive caches and prove its safety. In contrast to the state-of-the-art, we consider the CAC of accesses to the L2 cache and handle scenarios in which multiple references potentially issue the first access to a cache block after a preemption. Furthermore, we eliminated pessimism in the analysis by taking the ordering of accesses into account. Thus, a tighter bound on the indirect preemption effects can be computed. In our evaluations, we observed significant improvements in task set schedulability for small cache configurations. Schedulability was increased by up to 14 percentage points compared with the state-of-the-art.

In the future, it would be interesting to compare the CRPD values computed by the presented analysis with the context-switching costs measured on a real-system or observed in a simulation. This evaluation could show how tight the bound given by the presented analysis is compared with

the actually occurring CRPD and thus show how much room there is for improvements in the analysis precision.

In order for the presented analysis to be applicable to a system it has to satisfy several requirements, as discussed in Section 3. These limitations need to be addressed in future work in order to broaden the applicability of the presented analysis to COTS processors.

This article and all previously published works on two-level CRPD analysis [10, 36, 47] have focused on instruction caches. In the future, data caches need to considered. Data caches are harder to analyze than instruction caches because the target address of memory accesses to data objects may be difficult to determine precisely. This is the case in particular for input dependent data accesses. As memory accesses with uncertain target addresses are currently not supported, the presented analysis requires that both cache levels are instruction-only caches. While current processors commonly feature separated data and instruction caches at the first level, the second level cache is often unified. Thus, further research needs to be performed to handle data caches and consider the impact of data accesses in a unified L2 cache.

The LRU replacement policy has been recommended for real-time systems due to its high predictability [43]. However, commercial processors frequently implement different strategies. An extension of the presented analysis to other replacement policies, such as first-in-first-out replacement, could increase the applicability of the presented analysis.

The presented analysis considers systems with a single processor core. Multi-core systems often share the last-level cache between multiple cores. Sharing the second-level cache causes inter-core interferences, which leads to additional L2 cache misses. Chattopadhyay and Roychoudhury [10] proposed to account for inter-core interference in the CRPD analysis by counting the number of conflicting cache blocks accessed by interfering cores. However, it has been shown that this approach can be overly pessimistic when analyzing shared cache interference [15, 16, 33, 46]. Further research is needed to integrate the precise analysis of CRPD and shared cache interference.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2007. *Compilers: Principles, Techniques & Tools* (2 ed.). Pearson Education.

[2] Sebastian Altmeyer. 2012. *Analysis of Preemptively Scheduled Hard Real-time Systems*. Ph. D. Dissertation. Universität des Saarlandes.

[3] Sebastian Altmeyer and Claire Burguière. 2009. A new notion of useful cache block to improve the bounds of cache-related preemption delay. In *21st Euromicro Conference on Real-Time Systems (ECRTS'09)*. 109–118. DOI: https://doi.org/10.1109/ECRTS.2009.21

[4] Sebastian Altmeyer, Claire Maiza, and Jan Reineke. 2010. Resilience analysis: Tightening the CRPD bound for set-associative caches. *ACM SIGPLAN Notices* 45, 4 (2010), 153–162. DOI: https://doi.org/10.1145/1755951.1755911

[5] Sebastian Altmeyer and Claire Maiza Burguière. 2011. Cache-related preemption delay via useful cache blocks: Survey and redefinition. *Journal of Systems Architecture* 57, 7 (2011), 707–719. DOI: https://doi.org/10.1016/j.sysarc.2010.08.006 Special Issue on Worst-Case Execution-Time Analysis.

[6] Luna Backes and Daniel A. Jiménez. 2019. The impact of cache inclusion policies on cache management techniques. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)*. 428–438. DOI: https://doi.org/10.1145/3357526.3357547

[7] Robert Balas and Luca Benini. 2021. RISC-V for real-time MCUs—software optimization and microarchitectural gap analysis. In *2021 Design, Automation and Test in Europe Conference (DATE'21)*. 874–877. DOI: https://doi.org/10.23919/DATE51398.2021.9474114

[8] Enrico Bini and Giorgio C. Buttazzo. 2005. Measuring the performance of schedulability tests. *Real Time Systems* 30, 1–2 (2005), 129–154. DOI: https://doi.org/10.1007/s11241-005-0507-9

[9] J. V. Busquets-Mataix, J. J. Serrano, R. Ors, P. Gil, and A. Wellings. 1996. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Proceedings Real-Time Technology and Applications*. 204–212. DOI: https://doi.org/10.1109/RTTAS.1996.509537

[10] Sudipta Chattopadhyay and Abhik Roychoudhury. 2014. Cache-related preemption delay analysis for multilevel noninclusive caches. *ACM Transactions on Embedded Computing Systems (TECS)* 13, 5s (2014), 1–29. DOI: https://doi.org/10.1145/2632156

[11] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'77)*. 238–252. DOI: https://doi.org/10.1145/512950.512973

[12] Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza, Jan Reineke, Benoît Triquet, and Reinhard Wilhelm. 2010. Predictability considerations in the design of multi-core embedded systems. *Embedded Real Time Software and Systems Conference (ERTS)* 36 (2010), 10. http://web1.see.asso.fr/erts2010/Default.aspx-Id=973-Idd=982.htm

[13] Heiko Falk and Paul Lokuciejewski. 2010. A compiler framework for the reduction of worst-case execution times. *Real-Time Systems* 46, 2 (2010), 251–300. DOI: https://doi.org/10.1007/s11241-010-9101-x

[14] Christian Ferdinand and Reinhard Wilhelm. 1999. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems* 17, 2 (1999), 131–181. DOI: https://doi.org/10.1023/A:1008186323068

[15] Thilo L. Fischer and Heiko Falk. 2023. Analysis of shared cache interference in multi-core systems using event-arrival curves. In *Proceedings of Real-Time Network and Systems (RTNS'23)*. 23–33. DOI: https://doi.org/10.1145/3575757.3593643

[16] Thilo L. Fischer and Heiko Falk. 2024. Shared cache analysis under preemptive scheduling. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE'24)*. DOI: https://doi.org/10.23919/DATE58400.2024.10546581

[17] Alban Gruin, Thomas Carle, Hugues Cassé, and Christine Rochange. 2021. Speculative execution and timing predictability in an open source RISC-V core. In *2021 IEEE Real-Time Systems Symposium (RTSS'21)*. 393–404. DOI: https://doi.org/10.1109/RTSS52674.2021.00043

[18] Daniel Grund, Jan Reineke, and Reinhard Wilhelm. 2011. A template for predictability definitions with supporting evidence. In *Bringing Theory to Practice: Predictability and Performance in Embedded Systems (Open Access Series in Informatics (OASIcs), Vol. 18)*, Philipp Lucas and Reinhard Wilhelm (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 22–31. DOI: https://doi.org/10.4230/OASIcs.PPES.2011.22

[19] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. 2010. The Mälardalen WCET benchmarks–past, present and future. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET'10)*, Björn Lisper (Ed.). OCG, Brussels, Belgium, 136–146. DOI: https://doi.org/10.4230/OASIcs.WCET.2010.136

[20] Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. 2015. Towards compositionality in execution time analysis: Definition and challenges. *ACM SIGBED Review* 12, 1 (2015), 28–36. DOI: https://doi.org/10.1145/2752801.2752805

[21] Damien Hardy, Thomas Piquet, and Isabelle Puaut. 2009. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *2009 30th IEEE Real-Time Systems Symposium (RTSS'09)*. 68–77. DOI: https://doi.org/10.1109/RTSS.2009.34

[22] Damien Hardy and Isabelle Puaut. 2008. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *2008 Real-Time Systems Symposium (RTSS'08)*. 456–466. DOI: https://doi.org/10.1109/RTSS.2008.10

[23] Infineon. 2014. *AURIX™ TC21x/TC22x/TC23x Family*. Retrieved June 24, 2024 from https://community.infineon.com/gfawx74859/attachments/gfawx74859/AURIX/5399/1/Infineon-TC21x-TC22x-TC23x-UM-v01_01-EN.pdf

[24] Chang-Gun Lee, Hoosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. 1998. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Trans. Comput.* 47, 6 (1998), 700–713. DOI: https://doi.org/10.1109/12.689649

[25] Yau-Tsun Steven Li and Sharad Malik. 1997. Performance analysis of embedded software using implicit path enumeration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 16, 12 (1997), 1477–1487. DOI: https://doi.org/10.1109/43.664229

[26] Yun Liang, Huping Ding, Tulika Mitra, Abhik Roychoudhury, Yan Li, and Vivy Suhendra. 2012. Timing analysis of concurrent programs running on shared cache multi-cores. *Real-Time Systems* 48, 6 (2012), 638–680. DOI: https://doi.org/10.1007/s11241-012-9160-2

[27] Arm Limited. 2007. *PL310 Cache Controller Technical Reference Manual r0p0*. Retrieved June 5, 2024 from https://developer.arm.com/documentation/ddi0246/a/introduction/about-the-cache-controller

[28] Arm Limited. 2011. *Cortex-R4 and Cortex-R4F Technical Reference Manual r1p4*. Retrieved June 5, 2024 from https://developer.arm.com/documentation/ddi0363/g

[29] Arm Limited. 2024. *ARM Cortex-R Series Programmer's Guide*. Retrieved June 5, 2024 from https://developer.arm.com/documentation/den0042/a/Caches/Cache-policies/Replacement-policy

[30] Chung Laung Liu and James W. Layland. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)* 20, 1 (1973), 46–61. DOI: https://doi.org/10.1145/321738.321743

[31] Mingsong Lv, Nan Guan, Jan Reineke, Reinhard Wilhelm, and Wang Yi. 2016. A survey on static cache analysis for real-time systems. *Leibniz Transactions on Embedded Systems* 3, 1 (2016), 05:1–05:48. DOI: https://doi.org/10.4230/LITES-v003-i001-a005

[32] Tulika Mitra. 2019. Time-predictable computing by design: Looking back, looking forward. In *Proceedings of the 56th Annual Design Automation Conference 2019 (DAC'19)*. Article 153, 4 pages. DOI: https://doi.org/10.1145/3316781.3323489

[33] Kartik Nagar. 2016. *Precise analysis of Private and Shared Caches for tight WCET Estimates*. Ph. D. Dissertation. Indian Institute of Science Bangalore.

[34] Michael Platzer and Peter Puschner. 2021. Vicuna: A timing-predictable RISC-V vector coprocessor for scalable parallel computation. In *33rd Euromicro Conference on Real-Time Systems (ECRTS'21) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 196)*, Björn B. Brandenburg (Ed.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 1:1–1:18. DOI: https://doi.org/10.4230/LIPIcs.ECRTS.2021.1

[35] Syed Aftab Rashid, Geoffrey Nelissen, and Eduardo Tovar. 2020. Bounding cache persistence reload overheads for set-associative caches. In *2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'20)*. 1–10. DOI: https://doi.org/10.1109/RTCSA50079.2020.9203583

[36] Syed Aftab Rashid, Geoffrey Nelissen, and Eduardo Tovar. 2022. Tightening the CRPD bound for multilevel non-inclusive caches. *Journal of Systems Architecture* 122 (2022), 102340. DOI: https://doi.org/10.1016/j.sysarc.2021.102340

[37] Jan Reineke. 2014. Randomized caches considered harmful in hard real-time systems. *Leibniz Transactions on Embedded Systems* 1, 1 (2014), 03:1–03:13. DOI: https://doi.org/10.4230/LITES-v001-i001-a003

[38] Jan Reineke. 2018. The semantic foundations and a landscape of cache-persistence analyses. *Leibniz Transactions on Embedded Systems* 5, 1 (2018), 03:1–03:52. DOI: https://doi.org/10.4230/LITES-v005-i001-a003

[39] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. 2007. Timing predictability of cache replacement policies. *Real Time Systems* 37, 2 (2007), 99–122. DOI: https://doi.org/10.1007/s11241-007-9032-3

[40] Gregory Stock, Sebastian Hahn, and Jan Reineke. 2019. Cache persistence analysis: Finally exact. In *2019 IEEE Real-Time Systems Symposium (RTSS'19)*. 481–494. DOI: https://doi.org/10.1109/RTSS46320.2019.00049

[41] Valentin Touzeau, Claire Maïza, David Monniaux, and Jan Reineke. 2019. Fast and exact analysis for LRU caches. *Proceedings of the ACM on Programming Languages (POPL)* 3 (2019), 54:1–54:29. DOI: https://doi.org/10.1145/3290367

[42] Andrew Shell Waterman. 2016. *Design of the RISC-V instruction set architecture*. Ph. D. Dissertation. University of California, Berkeley.

[43] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. 2009. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28, 7 (2009), 966–978. DOI: https://doi.org/10.1109/TCAD.2009.2013287

[44] Reinhard Wilhelm and Jan Reineke. 2012. Embedded systems: Many cores–Many problems. In *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*. 176–180. DOI: https://doi.org/10.1109/SIES.2012.6356583

[45] Jun Xiao, Sebastian Altmeyer, and Andy Pimentel. 2017. Schedulability analysis of non-preemptive real-time scheduling for multicore processors with shared caches. In *2017 IEEE Real-Time Systems Symposium (RTSS'17)*. 199–208. DOI: https://doi.org/10.1109/RTSS.2017.00026

[46] Wei Zhang, Mingsong Lv, Wanli Chang, and Lei Ju. 2022. Precise and scalable shared cache contention analysis for WCET estimation. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC'22)*. 1267–1272. DOI: https://doi.org/10.1145/3489517.3530613

[47] Zhenkai Zhang and Xenofon Koutsoukos. 2016. Cache-related preemption delay analysis for multi-level inclusive caches. In *Proceedings of the 13th International Conference on Embedded Software (EMSOFT'16)*. Article 16, 10 pages. DOI: https://doi.org/10.1145/2968478.2968481

Received 31 January 2024; revised 3 July 2024; accepted 1 September 2024

9 Work in Progress: Optimizing Schedulability using Cache-Bypassing

Whereas the previous chapters have focused on the analysis of the worst-case timing behavior, this chapter presents an optimization to improve system schedulability. The optimization is applicable to single-core systems featuring a two-level cache hierarchy. The CRPD analysis introduced in the previous chapter is used to determine the WCRTs and system schedulability. The optimization objective is to increase the *breakdown utilization*, which is the highest utilization for which the system is still schedulable.

The optimization applies cache bypassing to improve the worst-case performance. This means parts of the address space are not accessed via the caches but will always be served directly from memory. The cache bypassing is realized by partitioning the memory address space into two sections: the cached and the uncached section. The optimization operates by allocating parts of the application code to uncached memory sections during compilation.

In essence, this approach performs a tradeoff between the WCET of the tasks and the context-switching costs, as well as the intra-task cache interference. As the process to determine the schedulability is a complex process, it is extremely challenging to build an analytical model, which captures all caching effects accurately (cf. [LKF16]).

For this reason, the presented optimization procedure utilizes *metaheuristic* algorithms to make the bypassing decisions. In particular, the efficacy of *simulated annealing* and the *strength pareto evolutionary algorithm* for the given optimization problem are explored.

31st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)

Submitted February 18, 2025.
Accepted March 17, 2025.
Presented in Irvine, United States of America, May 6 – 9, 2025.
DOI: 10.1109/RTAS65571.2025.00015

Work in Progress: Optimizing Schedulability using Cache-Bypassing

***Abstract*—We present an optimization technique to improve system schedulability using selective cache bypassing. By allocating parts of the application to uncached memory sections during compilation, context-switching costs and intra-task cache interference are reduced, leading to improved schedulability. We compare the performance of *simulated annealing* and the *strength-pareto evolutionary algorithm* (SPEA) for the optimization problem. Our evaluation demonstrates an increase in schedulability by up to 20% using SPEA.**

I. Introduction

Modern architectures utilize caches to overcome the performance gap between the faster processor core and slower main memory. Caches improve the average memory access latency by leveraging temporal locality, i.e. the reuse of data, and spatial locality, i.e. the tendency of accesses to target data in close proximity to previously accessed data. However, for real-time systems, caches introduce additional complexity to the verification process as the worst-case timing behavior of the cache has to be determined.

To verify that no deadlines are violated, the *worst-case response time* (WCRT) is determined. The WCRT corresponds to the maximal duration from the release of a task to its completion. The system is *schedulable* if the WCRT of all tasks is smaller than the associated deadline.

If a system is not schedulable, its implementation has to be optimized to meet the timing requirements. Since caches are a significant source of unpredictability in the microarchitectural analysis, optimizing the usage of this component is a promising strategy to improve the worst-case system performance.

One approach to reduce this unpredictability is to restrict cache utilization. Examples of this approach are *cache locking* [1], [2], where important data is loaded into the cache and locked into place, preventing its eviction, and *cache partitioning* [3], [4], where each task is assigned a dedicated subset of the cache, eliminating interference between different tasks. Other research has focused on architectures featuring additional *scratchpad memory* (SPM) [5], [6], which features predictable low latencies but is constrained by its size. Moving memory objects to the SPM reduces the traffic to the cache and thus increases the predictability of the cache behavior.

In this paper, we pursue an orthogonal approach: we selectively bypass caching for parts of the application to improve the task set schedulability. The optimization strategically places code fragments in uncached memory sections to reduce context-switching costs and intra-task cache interference. While moving code to uncached memory can increase a task's execution time, our evaluations demonstrate that this tradeoff can be performed such that overall system schedulability is improved.

Compared to cache locking and cache partitioning, we do not modify the core functionality of the cache. In contrast to SPM allocation techniques, the presented optimization does not require a dedicated scratchpad memory. SPM-based optimizations identify *critical* memory objects and allocate these objects to the scratchpad to reduce their access latency. The presented approach identifies *uncritical* memory objects and bypasses caching for those objects. While the access latency to the uncached objects increases, cache conflicts are reduced, which, as we will show, improves system performance.

The remainder of the paper is structured as follows: Section II reviews related work. Section III outlines the system architecture and provides background information. Section IV discusses the optimization problem. Section V presents evaluation results, demonstrating the effectiveness of our approach. Section VI concludes the paper and highlights future work.

II. Related Work

The context-switching costs caused by caches are called *cache-related preemption delay* (CRPD). CRPD occurs when tasks assigned to the same core share cache space and preemptions are enabled. During a preemption, the preempting task can evict data from the cache, forcing a reload of the data following the preemption. This creates a delay in the execution of the preempted task, which directly impacts the schedulability of the system. To quantify the CRPD, we use the analysis presented by Fischer and Falk [7].

Verma et al. [5] presented a static scratchpad allocation optimization, which models cache conflicts in a graph. By moving instructions from regular memory to scratchpad memory, the number of cache misses is reduced. As cache misses are a major contributor to energy usage, the overall energy consumption decreases.

Falk and Kotthaus [8] optimized the memory layout of a task to reduce the number of cache misses. The cache behavior is modelled using a conflict graph. The memory layout is modified iteratively by positioning the code fragments causing the highest number of cache misses contiguously in memory.

Luppold et al. [6] presented a cache-aware SPM allocation optimization to reduce the *worst-case execution time* (WCET), which is the maximal execution time of a task without interference from other tasks. The ILP model approximates cache conflicts to make allocation decisions.

Gebhard et al. [9] modify the placement of tasks in memory to improve performance in a preemptively scheduled system.

The approach determines the optimal starting address of each task considering the placement of other tasks in the system. The tasks are not modified, only the placement in memory is adjusted. The goal of the layout optimization is to reduce the threat of evictions caused by a preemption. They show that the optimal placement is NP-hard and solve a simplified problem using *simulated annealing* (SA) [10].

Lunniss et al. [11] present another placement optimization technique, inspired by [9]. The analysis is more precise, as it accounts for *useful* cache blocks and *evicting* cache blocks. Using SA, the schedulability is optimized. The nature of the analysis limits it to direct-mapped caches; set-associative caches are noted as future work.

In contrast to [9], [11], which optimized task ordering, the presented approach bypasses caches by moving parts of the application to uncached memory. And unlike [11], the presented approach is not restricted to direct-mapped caches.

The previously mentioned work focuses on single-level caches. Hardy et al. [12] proposed an optimization for multi-core systems with a shared second-level cache. Cache blocks accessed only once bypass the shared cache, decreasing shared cache pollution. The lower inter-core interference in the shared cache leads to improved performance. Bypassing not only the shared cache but also private caches is noted as potential future work. In this paper, we perform cache bypassing of a two-level cache hierarchy. In contrast to [12], we target a single-core system and evaluate the impact on system schedulability for preemptive scheduling.

III. System Model & Background

The hardware architecture considered in this paper consists of a single processor core with a two-level instruction cache hierarchy and main memory. The instruction memory hierarchy is visualized in Fig. 1. Previous research has shown that memory layout optimization can improve system schedulability and reduce energy consumption. While earlier work focused on single-level caches, modern systems typically incorporate multiple cache layers. For this reason, we apply the presented optimization to a two-level cache hierarchy. The presented approach of strategically bypassing caches, is applicable to instruction, data, and unified caches.

When a portion of the code is moved to a different memory location, all jumps targeting the moved code have to be adjusted to point to the new location. This process can necessitate inserting additional instructions, which is referred to as *jump correction code*. These corrections increase the total code size and introduce delays, potentially offsetting some of the gains from the optimized layout. As modifying application code is inherently more complex than relocating data objects to an uncached region, we focus on instruction caches in this paper to demonstrate that the optimization technique is beneficial even when additional jump correction code is needed.

The instruction memory consists of two sections: a cached section (`.text_cached`), and an uncached section (`.text_uncached`). Memory accesses to the cached section are handled by the L1 cache, the L2 cache, or memory. In contrast, an access to the uncached memory section bypasses both cache levels and is directly served from main memory. The optimization is applicable to systems with virtual memory if the CRPD can be determined [7].

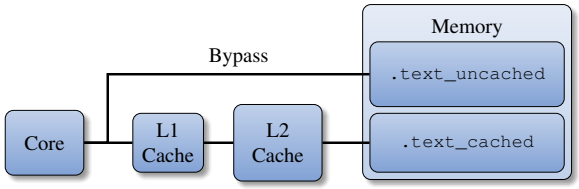


Fig. 1: Instruction Memory Hierarchy

When using caches, memory address ranges are grouped into cache blocks. The CRPD is determined using the concept of *useful cache blocks* (UCB) and *evicting cache blocks* (ECB) [13], [7]. A cache block of the preempted task is a UCB at a location in the program iff it is definitely cached and may be reused in the future without eviction. A cache block of the preempting task is an ECB iff it may be accessed by the task.

We denote the set of tasks to be executed on the system by T. Each task $\tau \in T$ has a minimal inter-arrival time P_τ, and a deadline $D_\tau \leq P_\tau$. The schedulability of a system depends on the WCRT R_τ of tasks $\tau \in T$. The WCRT can be determined from the WCET of each task and the CRPDs [14]. As the presented optimization modifies the memory layout, the WCET and context-switching costs of tasks are not fixed but depend on the currently considered memory layout. Let $\mathcal{B}$ denote the set containing all basic blocks. We define a memory layout as a mapping $l : \mathcal{B} \to \{0, 1\}$, which describes whether each basic block is assigned to cached (0) or uncached (1) memory. The set of all memory layouts is denoted by $\mathcal{L}$.

Let C_τ denote the WCET of τ; its dependence on the memory layout is explicitly shown by the parameter l: $C_\tau(l)$. Let $\delta_{\tau,\varphi}(l)$ denote the CRPD for $\tau \in T$ due to a preemption by $\varphi \in T \setminus \{\tau\}$. The function $hp : T \to 2^T$ returns the set of higher priority tasks, which may preempt τ. The response time of τ for a specific memory layout l is the fixpoint solution of:

$$R_\tau(l) = C_\tau(l) + \sum_{\varphi \in hp(\tau)} \left\lceil \frac{R_\tau(l)}{P_\varphi} \right\rceil \cdot \left(C_\varphi(l) + \delta_{\tau,\varphi}(l)\right). \quad (1)$$

A task τ is schedulable if the iterative solution of Eq. (1) converges to a value $\leq D_\tau$ [14]. The utilization U of a system for a particular memory layout l is given by $U(l) = \sum_{\tau \in T} \frac{C_\tau(l)}{P_\tau}$.

IV. Memory Layout Optimization

The schedulability of a system is a binary property: either a system is schedulable for the given task periods, or it may violate some deadline and is thus not schedulable.

To assess the performance of a memory layout, we use the concept of the *breakdown utilization*. This value describes the highest utilization value for which the system is still schedulable. To determine the breakdown utilization, task periods are scaled until the system is barely schedulable, meaning any further increase of the task's execution frequency would render the system unschedulable.

For applications where the task periods are given in physical time units, which are predetermined due to physical processes, the scaling of task periods can be interpreted as a change in the required processor clock frequency.

A. Optimization Rationale

Although it seems counter-intuitive to expect a speedup from moving code to uncached memory, it is possible to improve system performance by selectively bypassing caches. As can be seen in Eq. (1), the schedulability depends on two components: 1) task WCETs, and 2) context-switching costs.

The obvious drawback of bypassing caches for certain code fragments is the significant latency incurred when accessing those fragments relative to cached accesses. This additional latency can inflate the WCET if the uncached memory objects are required to execute the *worst-case execution path*, which is the path that realizes the WCET. Considered on its own, this effect potentially decreases the system schedulability.

However, there are significant benefits to selective cache bypassing. The context-switching costs depend on two factors: the UCBs of the preempted task, and the ECBs of the preempting tasks. Consider a task τ that is preempted by another task $\varphi \in hp(\tau)$. Not caching parts of the preempting task φ can directly reduce the number of evicting cache blocks of φ. This means that a preemption by φ with a modified memory layout, where φ is (partially) stored in uncached memory, causes τ to experience less CRPD compared to the layout where φ is stored completely in cached memory. This is the case as executing φ using the modified layout causes fewer evictions of useful blocks from τ. Hence, selective cache bypassing of a high priority task has beneficial effects on all lower priority tasks, which can be preempted by the high priority task.

Similarly, there are two distinct beneficial effects on τ, when some of its code is not cached. First, it can reduce the number of UCBs contributing to the CRPD penalty for τ, as the previously considered useful code needs not be reloaded into the cache after the preemption. Second, when a task $\tau \in T$ is partially moved to uncached memory, the intra-task cache interference reduces. This happens as fewer cache blocks are stored in the caches by τ. Thus, the cached code is less likely to be evicted from internal interference, preventing conflict misses, and is more resilient to eviction from the interference effects of a preemption by a higher priority task. Note that this effect can also lead to novel UCB classifications for τ: Cache blocks, which were previously not classified as useful as they were evicted by intra-task interference, can become useful in the modified layout.

We demonstrate in our evaluation that these effects contribute to an improved cache behavior, leading to an increased breakdown utilization.

B. Search Space

All elements $l \in \mathcal{L}$ are possible solutions to the optimization problem. The intuitive approach is to make a bypassing decision for each individual basic block. As the number of basic blocks correlates with the total code size, there may be a vast number of solution candidates. For this reason, we also evaluate the effectiveness of smaller search spaces that abstract from the concrete memory layout. This allows us to control the granularity of the optimization by adjusting the dimensionality of the search space and guide the optimization using heuristics. We propose two search space restrictions.

First, a reduced search space can be defined by making the allocation decision at the cache-block level. Instead of making a decision for each basic block, basic blocks can be moved to the uncached region based on whether they are part of a specific cache block in the initial layout. In this way multiple basic blocks can be moved to the uncached region by a single decision variable.

Second, we introduce the heuristic that it is beneficial to only bypass basic blocks that are not part of a loop in a function. If a basic block is part of a loop, it will potentially be executed many times. The intuition is that bypassing such a basic block can result in a large WCET increase. This heuristic is reminiscent of the approach from Hardy et al. [12], which bypassed single-use data around the shared L2 cache.

C. Optimization Algorithms

We have considered two different algorithms to optimize the breakdown utilization: *simulated annealing* [10], and the *strength pareto evolutionary algorithm* (SPEA) [15].

SA mimics the physical process of cooling a material to improve its structure, and has been used in previous layout optimizations [9], [11]. It is a single-objective algorithm, and we use the breakdown utilization as the metric. We resolve ties so that the candidate with less cache bypassing is preferable.

SPEA is an evolutionary algorithm, which operates by grouping multiple individuals in a population. New populations are generated by combining previous individuals. It can be applied to multi-objective problems, and we use the breakdown utilization and the amount of cached code as the two metrics to optimize.

V. Evaluation

We evaluated the effectiveness of the cache bypassing optimization using the WCET-aware C Compiler *WCC* [16]. The target architecture consists of a 3-stage, in-order pipeline using the ARMv4 instruction set. The core is connected to the main memory via a two-level instruction cache hierarchy. The caches use the least-recently-used replacement policy. The cache line size is 64 bytes. The L1 cache is 2-way associative and 1 KiB in size. The L2 cache is 4-way associative and 4 KiB in size. The memory access latencies are: 1 cycle if the access is served by the L1 cache (L1 hit); 10 cycles if the access is served by the L2 cache (L2 hit); 100 cycles if the main memory is accessed (L2 miss or bypassed access).

We generated 100 task sets containing 10 tasks from the MRTC benchmark suite [17]. Task periods were generated using UUnifast [18]. Deadlines are implicit. Tasks are scheduled using rate-monotonic preemptive scheduling [19].

We performed SA with an initial temperature of 100, a stopping temperature of 0.05, and a cooling rate of 0.98, leading

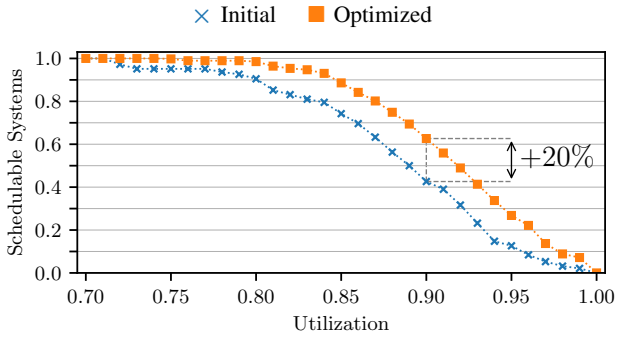


Fig. 2: Schedulable systems before and after optimization. Results of the SPEA algorithm making bypassing decisions for each basic block, without considering blocks in loops.

to 378 iterations, as in [11]. For SPEA, we configured the population size to 10 individuals and performed the optimization over 38 generations, leading to 380 evaluated individuals. The crossover probability was set to 0.8. The mutation rate of bypassing decisions was $\frac{1}{|\mathcal{B}'|}$, where $|\mathcal{B}'|$ is the dimension of the (restricted) search space. Each system is optimized 5 times to account for the randomized nature of the SA and SPEA algorithms. We aggregated the results of all runs. We evaluated all combinations of the two heuristics: moving multiple basic blocks, which belonged to the same cache block in the initial layout, and prohibiting the bypassing of basic blocks in loops. Coarsening the decision from individual basic blocks to cache blocks did not improve the optimization performance.

The best performance was exhibited by the SPEA algorithm when basic blocks are moved individually and blocks in loops are not considered. Fig. 2 shows the schedulability improvement achieved by this optimization. The x-axis shows system utilization and the y-axis shows the fraction of schedulable systems. For a system utilization of 0.9, schedulability is increased by 20%. Prior to optimization, 43% of systems were schedulable, whereas 63% of optimized systems were schedulable. In this configuration, cache bypassing improved breakdown utilization for 75% of systems. On average, for an optimized system, the breakdown utilization was increased by 0.042. The best performance of SA was achieved for the same configuration. However, in our evaluation, SPEA consistently outperformed SA. The maximal schedulability increase using SA was 8.65%.

VI. Conclusion & Outlook

Our results indicate that bypassing caches is a promising direction for system optimization, which needs to be explored in greater detail.

In the future, the task positioning optimization proposed by [11] could be extended to the considered system architecture and incorporated into the presented optimization. This would allow the optimization to simultaneously consider the position of tasks in memory and selective cache bypassing to improve system schedulability.

While SA has been used in previous layout optimizations [9], [11], we observed that SPEA yielded higher schedulability improvements in our evaluation. This motivates the application of SPEA to related optimization problems. Further experiments could evaluate other algorithms such as flower pollination.

Furthermore, we want to explore an extension of the presented approach, where the initial memory layout of cached code is preserved. Currently, moving a block to the uncached section modifies the address of all following blocks, as they slide up to fill the gap. Pinning the blocks in the cached section at fixed addresses could help the optimization algorithm discover better solutions. Additionally, the effectiveness of cache bypassing in preemptively scheduled multi-core systems with shared caches could be evaluated.

References

[1] S. Mittal, "A Survey of Techniques for Cache Locking," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 21, no. 3, May 2016.
[2] H. Falk, S. Plazar, and H. Theiling, "Compile-Time Decided Instruction Cache Locking Using Worst-Case Execution Paths," in *Proc. of CODES+ISSS*, 2007, pp. 143–148.
[3] S. Altmeyer, R. Douma, W. Lunniss, and R. I. Davis, "On the effectiveness of cache partitioning in hard real-time systems," *Real-Time Systems*, vol. 52, pp. 598–643, 2016.
[4] B. Sun, T. Kloda, S. Arribas Garcia, G. Gracioli, and M. Caccamo, "Minimizing Cache Usage for Real-time Systems," in *Proc. of RTNS*, 2023, p. 200–211.
[5] M. Verma, L. Wehmeyer, and P. Marwedel, "Cache-aware scratchpad allocation algorithm," in *Proc. of DATE*, 2004, pp. 1264–1269 Vol.2.
[6] A. Luppold, C. Kittsteiner, and H. Falk, "Cache-Aware Instruction SPM Allocation for Hard Real-Time Systems," in *Proc. of SCOPES*, 2016, p. 77–85.
[7] T. L. Fischer and H. Falk, "Towards Analysing Cache-Related Preemption Delay in Non-Inclusive Cache Hierarchies," *ACM Trans. Embed. Comput. Syst.*, vol. 24, no. 1, Oct. 2024.
[8] H. Falk and H. Kotthaus, "WCET-driven Cache-aware Code Positioning," in *Proc. of CASES*, 2011, p. 145–154.
[9] G. Gebhard and S. Altmeyer, "Optimal Task Placement to Improve Cache Performance," in *Proc. of EMSOFT*, 2007, p. 259–268.
[10] S. Kirkpatrick, C. D. Gelatt Jr, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, pp. 671–680, 1983.
[11] W. Lunniss, S. Altmeyer, and R. I. Davis, "Optimising Task Layout to Increase Schedulability via Reduced Cache Related Pre-emption Delays," in *Proc. of RTNS*, 2012, p. 161–170.
[12] D. Hardy, T. Piquet, and I. Puaut, "Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches," in *Proc. of RTSS*, 2009, pp. 68–77.
[13] S. Altmeyer and C. Maiza Burguière, "Cache-related preemption delay via useful cache blocks: Survey and redefinition," *Journal of Systems Architecture*, vol. 57, no. 7, pp. 707–719, 2011.
[14] J. Busquets-Mataix, J. Serrano, R. Ors, P. Gil, and A. Wellings, "Adding Instruction Cache Effect to Schedulability Analysis of Preemptive Real-Time Systems," in *Proc. of RTTAS*, 1996, pp. 204–212.
[15] E. Zitzler and L. Thiele, "Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach," *IEEE Transactions on Evolutionary Computation*, vol. 3, no. 4, pp. 257–271, 1999.
[16] H. Falk and P. Lokuciejewski, "A Compiler Framework for the Reduction of Worst-Case Execution Times," *Real-Time Systems*, vol. 46, no. 2, pp. 251–300, 2010.
[17] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET Benchmarks – Past, Present and Future," in *Proc. of WCET*, B. Lisper, Ed., 2010, pp. 136–146.
[18] E. Bini and G. C. Buttazzo, "Measuring the Performance of Schedulability Tests," *Real-Time Systems*, vol. 30, pp. 129–154, 2005.
[19] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

Summary & Outlook

10

In this final chapter, we summarize the contributions of the previous chapters in Section 10.1, discuss limitations in Section 10.2, and highlight potential future work in Section 10.3.

10.1 Summary

This thesis investigates the timing behavior of real-time systems featuring multi-level caches. The timing behavior is analyzed using static timing analyses, meaning the application code is examined without directly executing it. The presented analyses make use of abstract interpretation to enhance analysis efficiency, which is required to achieve manageable analysis overhead. While the utilized abstractions sacrifice some precision, the derived results are provably safe and the respective evaluations demonstrate the practical utility of the presented analyses. In comparison to previously existing analyses, the methods introduced in this thesis offer improved precision across many system configurations.

First, an analysis of shared cache interference in multi-core systems is presented. Shared cache interference arises when tasks execute in parallel on multiple cores and access a shared cache. This inter-core interference causes additional cache misses, which would not have occurred if the tasks were executed in isolation. Consequently, the classification of accesses to the shared cache has to consider inter-core interference to derive safe cache hit classifications.

In Chapter 6, a novel perspective on shared cache interference is presented. Interference is quantified using event-arrival curves. The interference curves are functions over a time domain, which capture the maximal interference within a given time frame. This allows the analysis to make classification decisions based on the temporal recency of previous accesses to data in the shared cache. The analysis, as presented in Chapter 6, is applicable to systems with a single task per core and to non-preemptively scheduled systems. Interference from multiple tasks being executed non-preemptively on the same core is bounded by convolving the interference curves in the max-plus algebra using a novel algorithm.

In Chapter 7, the shared cache analysis is extended to support preemptive scheduling. The preemptive scheduling of tasks is modelled using a formal language. The language makes use of the fixed priority assignments of tasks. As the tasks executing on the system are known during the analysis, the constructed language is regular. The worst-case cache interference is determined by searching for words in the language, which represent scheduling scenarios, that maximize the amount of accesses to distinct cache blocks over a given time frame.

Second, in Chapter 8, cache-related preemption delay in multi-level caches is analyzed. CRPD occurs when tasks are scheduled preemptively, i.e., a task may be interrupted in favor of a higher priority task. In systems featuring caches, preemptions cause delays for the

preempted task, as the cache contents has to be restored after the preemption. In particular, non-inclusive cache hierarchies, which neither enforce nor prohibit data duplication in different cache levels are targeted in this thesis. Example scenarios, which are not handled correctly by the previous state-of-the-art are presented. Motivated by the complexity of CRPD analysis in multi-level caches, we introduce a formal foundation on the analysis of indirect preemption effects. Then, a novel CRPD analysis for two-level non-inclusive cache hierarchies is presented. Using the previously established formal foundation, we establish the safety of the presented analysis. By eliminating pessimism of earlier approaches, the system schedulability is improved.

To conclude the thesis, an application of the CRPD analysis from Chapter 8 is presented in Chapter 9. The presented optimization leverages cache bypassing to improve system schedulability. The decisions, which parts of the application to bypass around the caches, are made using metaheuristic algorithms. The optimization algorithms evaluate the fitness of a particular memory layout using the CRPD analysis presented in Chapter 8.

10.2 Discussion

This thesis aimed at improving the capabilities of static timing analyses for real-time systems featuring multi-level cache hierarchies. This goal was achieved by 1. improving the precision with which shared caches in multi-core systems may be analyzed, and 2. by advancing the state-of-the-art analysis for context-switching costs occurring due to multi-level cache hierarchies.

However, the presented analyses are limited in some capacities, which reduce their applicability to commercial-of-the-shelf hardware architectures:

LRU Replacement Policy The analyses presented in this thesis assume the utilization of the LRU replacement policy. In research on static timing analysis, LRU replacement is the de-facto standard of cache replacement policies due to its highly predictable nature [WEE+08, CFG+10, Rei14]. However, LRU replacement is costly to implement in hardware. Hardware manufacturers of commercial processor often opt for different replacement policies, such as random replacement [Arm14] or proprietary algorithms [AR14]. This disconnect negatively impacts the applicability of the presented analyses to commercial system architectures.

For the classification of accesses to a shared cache, the key property used in the analysis is the resilience of cache blocks. This resilience to inter-core interference may also be determined for different replacement policies. Consequently, the classification could potentially be adapted to replacement policies other than LRU with some changes. However, the required changes may significantly degrade the achievable analysis precision. For example, depending on the replacement policy not all inter-core interference may be eliminated after an access to a particular cache block in the shared cache.

In contrast, the process to derive event-arrival curves is not particular to LRU replacement. The derivation procedure only makes use of the BCET of individual basic blocks, and the information which cache blocks may be accessed during the execution of a particular block.

The approaches to determine CRPD presented in this analysis, are coupled to the LRU replacement policy. This is the case, as the analysis assumes that an access to the cache block will "refresh" the block in the cache and restore its resilience to the cache's associativity. Thus, an extension to other replacement policies will require changes in the analysis technique, potentially resulting in lower analysis precision.

Timing Compositionality Timing compositionality is the property of a system decomposition into multiple components stating that the timing of the complete system can be determined by analyzing the components individually [HRW15]. In particular, the response time analysis presented in Section 4.3 requires timing compositionality to safely account for context-switching costs. Furthermore, the iterative application of the cache hit classification, as proposed in Chapter 6 requires timing compositionality, if a microarchitectural analysis is not performed after each classification run. While timing compositionality is a common requirement of timing analyses [HJR16], it is crucial to consider this property when analyzing a given hardware platform.

Non-compositional architectures require a different analysis approach which integrates several components into the analysis as it is not possible to determine the system's timing behavior from its isolated components. This is fundamentally at odds with the approach to determine CRPD in isolation, as is done in Chapters 7 and 8. Consequently, expanding the presented analyses to a non-compositional architecture may not be feasible.

10.3 Outlook

Based on the insights gained in this thesis, there are several promising directions for future research:

Analysis Overhead The overhead of the shared cache analysis is substantial. The analysis overhead results mainly from the requirement to derive event-arrival curves. Event-arrival curves are derived for each task and cache set. High analysis overhead could reduce the utility of the analysis approach. In order to reduce analysis overhead, the construction of the event-arrival curves could be safely approximated. Another approach to ease the derivation of event-arrival curves would be to enforce certain cache access rate limits in hardware [ZW16]. Then, the event-arrival could be derived directly from hardware properties, reducing the analysis overhead. Evaluating such adaptations and potential precision tradeoffs could be evaluated in future research.

Interconnect Architecture The shared cache analysis targets centralized shared caches, which are accessed via a shared bus. This means that there is a single bottleneck which can limit the system performance, as all cores have to access the same shared resource. This can be especially problematic as the core count will likely increase in future architectures. Recently, the *network-on-chip* (NoC) paradigm is becoming increasingly popular [HRGAEG17]. NoC architectures eliminate the central shared bus by implementing a network between components with multiple possible routes to distribute the traffic. In future work, the pre-

sented results on event-arrival curve combination to model scheduling decisions could be applied in the analysis of NoC architectures.

Combined Analysis The shared cache analysis for preemptively scheduled tasks is based on the analysis of Chattopadhyay and Roychoudhury to determine CRPD. A combined approach of the analyses presented in this thesis could be developed in the future. Based on the results of Chapters 7 and 8, such a combined approach would likely further increase system schedulability. In an application, the increased schedulability would enable the utilization of less powerful, and thus cheaper, hardware, or support the execution of more complex application code.

CRPD Analysis of Data Caches The CRPD analysis presented in this thesis focuses on instruction caches. This is also the case for all other existing analysis approaches for multi-level caches. Accesses to instruction caches are predictable during a static analysis, as the control-flow of the application is generally well known. Data caches differentiate themselves from instruction caches due to the high unpredictability of target addresses of cache accesses. For example, consider the array access statement `arr[i];`. Depending on the location of `arr` in memory and the value of the index `i`, the accessed memory location varies. If the location of `arr` and the value of `i` are not known precisely, different scenarios have to be considered in the cache analysis. While an approach to handle such contingent accesses for the May and Must analyses is described in Section 5.1.3, the precise CRPD analysis of data and unified caches remains an unexplored problem.

System Optimization If the analysis of a system reveals that the worst-case performance is not satisfying the requirements, the implementation of the system has to be improved. This thesis has mainly focused on the analysis of the worst-case behavior. The insights gained in these analyses could be utilized to perform optimizations. A first step in this direction was taken in Chapter 9. The presented optimization uses metaheuristic algorithms to decide which parts of an application should be bypassed around the caches. Note that this optimization does not directly utilize detailed information made available by the CRPD analysis. Creating a precise model for complex architectures has proven to be challenging in the past (see [LAD12, LKF16]). Thus, creating optimizations, which make full use of the information provided by the presented analyses, are a topic for future research.

Bibliography

[AB08] Björn Andersson and Konstantinos Bletsas. Sporadic Multiprocessor Scheduling with Few Preemptions. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, page 243–252, 2008. doi:10.1109/ECRTS.2008.9.

[AB09] Sebastian Altmeyer and Claire Burguière. A New Notion of Useful Cache Block to Improve the Bounds of Cache-Related Preemption Delay. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, pages 109–118, 2009. doi:10.1109/ECRTS.2009.21.

[ABD05] James H. Anderson, Vasile Bud, and UmaMaheswari C. Devi. An EDF-based scheduling algorithm for multiprocessor soft real-time systems. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, pages 199–208, 2005. doi:10.1109/ECRTS.2005.6.

[ABR+93] Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[ABRW91] N.C. Audsley, A. Burns, M.F. Richardson, and A.J. Wellings. Hard real-time scheduling: The deadline-monotonic approach. *IFAC Proceedings Volumes*, 24(2):127–132, 1991. IFAC/IFIP Workshop on Real Time Programming, Atlanta, GA, USA, 15-17 May 1991. doi:10.1016/S1474-6670(17)51283-5.

[AG04] Andrew W. Appel and Maia Ginsburg. *Modern compiler implementation in C*. Cambridge University Press, 2004.

[AHQ+15] Jaume Abella, Carles Hernandez, Eduardo Quiñones, Francisco J. Cazorla, Philippa Ryan Conmy, Mikel Azkarate-askasua, Jon Perez, Enrico Mezzetti, and Tullio Vardanega. WCET Analysis Methods: Pitfalls and Challenges on their Trustworthiness. In *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–10, 2015. doi:10.1109/SIES.2015.7185039.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques & Tools*. Addison-Wesley Longman Publishing Co., Inc., 2 edition, 2006.

[Alt12] Sebastian Altmeyer. *Analysis of Preemptively Scheduled Hard Real-time Systems*. PhD thesis, Universität des Saarlandes, 2012.

[AM11] Sebastian Altmeyer and Claire Maiza Burguière. Cache-related preemption delay via useful cache blocks: Survey and redefinition. *Journal of Systems Architecture*, 57(7):707–719, 2011. doi:10.1016/j.sysarc.2010.08.006.

[AMR10] Sebastian Altmeyer, Claire Maiza, and Jan Reineke. Resilience Analysis: Tightening the CRPD Bound for Set-Associative Caches. *ACM SIGPLAN Notices*, 45(4):153–162, 2010. doi:10.1145/1755951.1755911.

[AR14] Andreas Abel and Jan Reineke. Reverse Engineering of Cache Replacement Policies in Intel Microprocessors and Their Evaluation. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 141–142, 2014. doi:10.1109/ISPASS.2014.6844475.

[Arm14] Arm Ltd. ARM Cortex-R Series Programmer's Guide, 2014. Accessed: March 17, 2025. URL: https://developer.arm.com/documentation/den0042/a/Caches/Cache-policies/Replacement-policy.

[Arm20] Arm Ltd. Arm Cortex-M0 Processor Datasheet, 2020. Accessed: March 17, 2025. URL: https://www.arm.com/-/media/Arm%20Developer%20Community/PDF/Processor%20Datasheets/Arm_Cortex-M0_Processor_Datasheet.pdf.

[BBA11] Andrea Bastoni, Bjorn B. Brandenburg, and James H. Anderson. Is Semi-Partitioned Scheduling Practical? In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS)*, pages 125–135, 2011. doi:10.1109/ECRTS.2011.20.

[BCOQ01] François Baccelli, Guy Cohen, Geert Jan Olsder, and Jean-Pierre Quadrat. *Synchronization and Linearity: An Algebra for Discrete Event Systems*. Wiley, 2001.

[BG16] Björn B. Brandenburg and Mahircan Gül. Global Scheduling Not Required: Simple, Near-Optimal Multiprocessor Real-Time Scheduling with Semi-Partitioned Reservations. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, pages 99–110, 2016. doi:10.1109/RTSS.2016.019.

[BJ19] Luna Backes and Daniel A. Jiménez. The Impact of Cache Inclusion Policies on Cache Management Techniques. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)*, page 428–438, 2019. doi:10.1145/3357526.3357547.

[BMSO+96] José V. Busquets-Mataix, J.J. Serrano, R. Ors, P. Gil, and A. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium*, pages 204–212, 1996. doi:10.1109/RTTAS.1996.509537.

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, page 238–252, 1977. doi:10.1145/512950.512973.

[CC04] Patrick Cousot and Radhia Cousot. Basic concepts of abstract interpretation. In *Building the Information Society*, pages 359–366. Springer US, 2004. doi:10.1007/978-1-4020-8157-6_27.

[CD97a] Michel Cekleov and Michel Dubois. Virtual-Address Caches. Part 1: Problems and Solutions in Uniprocessors. *IEEE Micro*, 17(5):64–71, 1997. doi:10.1109/40.621215.

[CD97b] Michel Cekleov and Michel Dubois. Virtual-Address Caches. Part 2: Multiprocessor issues. *IEEE Micro*, 17(6):69–74, 1997. doi:10.1109/40.641599.

[CFG+10] Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza, Jan Reineke, Benoît Triquet, and Reinhard Wilhelm. Predictability Considerations in the Design of Multi-Core Embedded Systems. In *Proceedings of Embedded Real Time Software and Systems (ERTS^2)*, 2010.

[CKT03] Samarjit Chakraborty, Simon Künzli, and Lothar Thiele. A General Framework for Analysing System Properties in Platform-Based Embedded System Designs. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 190–195, 2003. doi:10.1109/DATE.2003.1253607.

[CR14] Sudipta Chattopadhyay and Abhik Roychoudhury. Cache-related preemption delay analysis for multilevel noninclusive caches. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(5s):1–29, 2014. doi:10.1145/2632156.

[Cru91a] Rene L. Cruz. A Calculus for Network Delay, Part 1: Network Elements in Isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, 1991. doi:10.1109/18.61109.

[Cru91b] Rene L. Cruz. A Calculus for Network Delay, Part 2: Network Analysis. *IEEE Transactions on Information Theory*, 37(1):132–141, 1991. doi:10.1109/18.61110.

[Dal04] William James Dally. *Principles and practices of interconnection networks*. Elsevier, 2004.

[ESG+07] Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper. Loop Bound Analysis based on a Combination of Program Slicing, Abstract Interpretation, and Invariant Analysis. In *Proceedings of the 7th International Workshop on Worst-Case Execution Time Analysis (WCET)*, volume 6, pages 1–6, 2007. doi:10.4230/OASIcs.WCET.2007.1194.

[FF23a] Thilo L. Fischer and Heiko Falk. Analysis of Shared Cache Interference in Multi-Core Systems using Event-Arrival Curves. In *Proceedings of Real-Time Network and Systems (RTNS)*, pages 23–33, 2023. doi:10.1145/3575757.3593643.

[FF23b] Thilo L. Fischer and Heiko Falk. Wcet analysis of shared caches in multi-core architectures using event-arrival curves. In *Proceedings of Design, Automation & Test in Europe Conference (DATE)*, 2023. doi:10.23919/DATE56975.2023.10137034.

[FF24a] Thilo L. Fischer and Heiko Falk. Shared Cache Analysis under Preemptive Scheduling. In *Proceedings of Design, Automation & Test in Europe Conference (DATE)*, 2024. doi:10.23919/DATE58400.2024.10546581.

[FF24b] Thilo Leon Fischer and Heiko Falk. Timing-aware shared cache analysis for non-preemptive scheduling. *Real-Time Systems*, 60(4):570–624, October 2024. doi:10.1007/s11241-024-09430-8.

[FF24c] Thilo Leon Fischer and Heiko Falk. Towards analysing cache-related preemption delay in non-inclusive cache hierarchies. *ACM Trans. Embed. Comput. Syst.*, 24(1), October 2024. doi:10.1145/3695768.

[FF25] Thilo L. Fischer and Heiko Falk. Work in progress: Optimizing schedulability using cache-bypassing. In *2025 IEEE 31st Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 418–421, 2025. © 2025 IEEE. Reprinted, with permission, from this source. doi:10.1109/RTAS65571.2025.00015.

[FGGH23] Christoph Funda, Pablo Marín García, Reinhard German, and Kai-Steffen Hielscher. Arrival and service curve measurement-based estimation methods to analyze and design soft real-time streaming systems with network calculus. In *3rd International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME)*, pages 1–8, 2023. doi:10.1109/ICECCME57830.2023.10253001.

[FH04] Christian Ferdinand and Reinhold Heckmann. aiT: Worst-Case Execution Time Prediction by Static Program Analysis. In *Building the Information Society*, pages 377–383. Springer US, 2004. doi:10.1007/978-1-4020-8157-6_29.

[Fid10] Markus Fidler. Survey of Deterministic and Stochastic Service Curve Models in the Network Calculus. *IEEE Communications Surveys & Tutorials*, 12(1):59–86, 2010. doi:10.1109/SURV.2010.020110.00019.

[FL10] Heiko Falk and Paul Lokuciejewski. A Compiler Framework for the Reduction of Worst-Case Execution Times. *Real-Time Systems*, 46(2):251–300, 2010. doi:10.1007/s11241-010-9101-x.

[FW99] Christian Ferdinand and Reinhard Wilhelm. Efficient and Precise Cache Behavior Prediction for Real-Time Systems. *Real-Time Systems*, 17(2):131–181, 1999. doi:10.1023/A:1008186323068.

[GFPF13] Giovani Gracioli, Antônio Augusto Fröhlich, Rodolfo Pellizzoni, and Sebastian Fischmeister. Implementation and evaluation of global and partitioned scheduling in a real-time os. *Real-Time Systems*, 49:669–714, 2013. doi:10.1007/s11241-013-9183-3.

[GLSB03] Jan Gustafsson, Björn Lisper, Christer Sandberg, and Nerina Bermudo. A Tool for Automatic Flow Analysis of C-programs for WCET Calculation. In *Proceedings of the Eighth International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS).*, pages 106–112, 2003. doi:10.1109/WORDS.2003.1218072.

[HJH+22] Sebastian Hahn, Michael Jacobs, Nils Hölscher, Kuan-Hsun Chen, Jian-Jia Chen, and Jan Reineke. LLVMTA: An LLVM-Based WCET Analysis Tool. In *Proceedings of the 20th International Workshop on Worst-Case Execution Time Analysis (WCET)*, pages 2:1–2:17, 2022. doi:10.4230/OASIcs.WCET.2022.2.

[HJR16] Sebastian Hahn, Michael Jacobs, and Jan Reineke. Enabling Compositionality for Multicore Timing Analysis. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems (RTNS)*, page 299–308, 2016. doi:10.1145/2997465.2997471.

[Hor74] W. A. Horn. Some Simple Scheduling Algorithms. *Naval Research Logistics Quarterly*, 21(1):177–185, 1974. doi:10.1002/nav.3800210113.

[HP08] Damien Hardy and Isabelle Puaut. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *Proceedings of the Real-Time Systems Symposium (RTSS)*, pages 456–466, 2008. doi:10.1109/RTSS.2008.10.

[HP11] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, 5 edition, 2011.

[HPP09] Damien Hardy, Thomas Piquet, and Isabelle Puaut. Using Bypass to Tighten WCET Estimates for Multi-Core Processors with Shared Instruction Caches. In *Proceedingso of the 30th IEEE Real-Time Systems Symposium (RTSS)*, pages 68–77, 2009. doi:10.1109/RTSS.2009.34.

[HRGAEG17] Salma Hesham, Jens Rettkowski, Diana Goehringer, and Mohamed A. Abd El Ghany. Survey on Real-Time Networks-on-Chip. *IEEE Transactions on Parallel and Distributed Systems*, 28(5):1500–1517, 2017. doi:10.1109/TPDS.2016.2623619.

[HRP17] Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. The Heptane Static Worst-Case Execution Time Estimation Tool. In *Proceedings of the 17th International Workshop on Worst-Case Execution Time Analysis (WCET)*, volume 57, pages 8:1–8:12, 2017. doi:10.4230/OASIcs.WCET.2017.8.

[HRW15] Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. Towards Compositionality in Execution Time Analysis: Definition and Challenges. *ACM SIGBED Review*, 12(1):28–36, 2015. doi:10.1145/2752801.2752805.

[JNWR08] Bruce Jacob, Spencer Ng, David Wang, and Samuel Rodriguez. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2008.

[KAQC13] Leonidas Kosmidis, Jaume Abella, Eduardo Quiñones, and Francisco J. Cazorla. A Cache Design for Probabilistically Analysable Real-Time Systems. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 513–518, 2013. doi:10.7873/DATE.2013.116.

[Kel14] Timon Kelter. *WCET Analysis and Optimization for Multi-Core Real-Time Systems*. PhD thesis, Technische Universität Dortmund, 2014.

[KFM+11] Timon Kelter, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay, and Abhik Roychoudhury. Bus-Aware Multicore WCET Analysis through TDMA Offset Bounds. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS)*, pages 3–12. IEEE, 2011. doi:10.1109/ECRTS.2011.9.

[KFM+14] Timon Kelter, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay, and Abhik Roychoudhury. Static analysis of multi-core TDMA resource arbitration delays. *Real-Time Systems*, 50:185–229, 2014. doi:10.1007/s11241-013-9189-x.

[LAD12] Will Lunniss, Sebastian Altmeyer, and Robert I. Davis. Optimising Task Layout to Increase Schedulability via Reduced Cache Related Pre-emption Delays. In *Proceedings of the 20th International Conference on Real-Time Networks and Systems (RTNS)*, page 161–170, 2012. doi:10.1145/2392987.2393008.

[LCFM09] P. Lokuciejewski, D. Cordes, H. Falk, and P. Marwedel. A Fast and Precise Static Loop Analysis Based on Abstract Interpretation, Program Slicing and Polytope Models. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 136–146, 2009. doi:10.1109/CGO.2009.17.

[LDM+12] Yun Liang, Huping Ding, Tulika Mitra, Abhik Roychoudhury, Yan Li, and Vivy Suhendra. Timing Analysis of Concurrent Programs Running on Shared Cache Multi-Cores. *Real-Time Systems*, 48(6):638–680, 2012. doi:10.1007/s11241-012-9160-2.

[Leu04] Joseph Y. T. Leung. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. Chapman and Hall/CRC, 2004.

[Lie17] Jörg Liebeherr. Duality of the Max-Plus and Min-Plus Network Calculus. *Foundations and Trends® in Networking*, 11(3-4):139–282, 2017. doi:10.1561/1300000059.

[LKF16] Arno Luppold, Christina Kittsteiner, and Heiko Falk. Cache-Aware Instruction SPM Allocation for Hard Real-Time Systems. In *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, page 77–85, 2016. doi:10.1145/2906363.2906369.

[LL73] Chung Laung Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973. doi:10.1145/321738.321743.

[LLF+15] Jing Li, Zheng Luo, David Ferry, Kunal Agrawal, Christopher Gill, and Chenyang Lu. Global EDF scheduling for parallel real-time tasks. *Real-Time Systems*, 51:395–439, 2015. doi:10.1007/s11241-014-9213-9.

[LLMR07] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69(1):56–67, 2007. doi:10.1016/j.scico.2007.01.014.

[LM97] Yau-Tsun Steven Li and Sharad Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(12):1477–1487, 1997. doi:10.1109/43.664229.

[LM11] Paul Lokuciejewski and Peter Marwedel. *Worst-Case Execution Time Aware Compilation Techniques for Real-Time Systems*. Springer Science & Business Media, 1 edition, 2011. doi:10.1007/978-90-481-9929-7.

[LSD89] John Lehoczky, Lui Sha, and Yuqin Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of the Real-Time Systems Symposium (RTSS)*, pages 166–171, 1989. doi:10.1109/REAL.1989.63567.

[Mar21] Peter Marwedel. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things*. Springer Nature, 4 edition, 2021. doi:10.1007/978-3-030-60910-8.

[MAWF98] Florian Martin, Martin Alt, Reinhard Wilhelm, and Christian Ferdinand. Analysis of Loops. In *Proceedings of Compiler Construction (CC)*, pages 80–94. Springer, 1998. doi:10.1007/BFb0026424.

[McK04] Sally A. McKee. Reflections on the memory wall. In *Proceedings of the 1st Conference on Computing Frontiers (CF)*, pages 162–167, 2004. doi:10.1145/977091.977115.

[MH21] Anna Minaeva and Zdeněk Hanzálek. Survey on Periodic Scheduling for Time-triggered Hard Real-time Systems. *ACM Computing Surveys*, 54(1), 2021. doi:10.1145/3431232.

[MIM16] Guilherme Madalozzo, Leandro S. Indrusiak, and Fernando G. Moraes. Mapping of real-time applications on a packet switching NoC-based MPSoC. In *Proceedings of the IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 640–643, 2016. doi:10.1109/ICECS.2016.7841283.

[MTK+11] Peter Marwedel, Jürgen Teich, Georgia Kouveli, Iuliana Bacivarov, Lothar Thiele, Soonhoi Ha, Chanhee Lee, Qiang Xu, and Lin Huang. Mapping of applications to MPSoCs. In *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, page 109–118, 2011. doi:10.1145/2039370.2039390.

[Muc97] Steven S. Muchnick. *Advanced compiler design implementation*. Morgan Kaufmann Publishers Inc., 1997.

[Nag16] Kartik Nagar. *Precise analysis of Private and Shared Caches for tight WCET Estimates*. PhD thesis, Indian Institute of Science Bangalore, 2016.

[Oeh21] Dominic Oehlert. *Worst Case Execution Time Oriented Code Optimization of Hard Real-Time Multicore Systems*. PhD thesis, Technische Universität Hamburg, 2021. doi:10.15480/882.3855.

[OSF18] Dominic Oehlert, Selma Saidi, and Heiko Falk. Compiler-Based Extraction of Event Arrival Functions for Real-Time Systems Analysis. In *Proceedings of the 30th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 4:1–4:22, 2018. doi:10.4230/LIPIcs.ECRTS.2018.4.

[PH12] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware Software Interface*. Morgan Kaufmann, 4 edition, 2012.

[Rei14] Jan Reineke. Randomized Caches Considered Harmful in Hard Real-Time Systems. *Leibniz Transactions on Embedded Systems (LITES)*, 1(1):03:1–03:13, 2014. doi:10.4230/LITES-v001-i001-a003.

[RGBW07] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37:99–122, 2007. doi:10.1007/s11241-007-9032-3.

[RNT22] Syed Aftab Rashid, Geoffrey Nelissen, and Eduardo Tovar. Tightening the CRPD bound for multilevel non-inclusive caches. *Journal of Systems Architecture*, 122, 2022. doi:10.1016/j.sysarc.2021.102340.

[SCR+14] Hardik Shah, Andrew Coombes, Andreas Raabe, Kai Huang, and Alois Knoll. Measurement based wcet analysis for multi-core architectures. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems (RTNS)*, page 257–266, 2014. doi:10.1145/2659787.2659819.

[SCT10] Andreas Schranzhofer, Jian-Jia Chen, and Lothar Thiele. Timing Analysis for TDMA Arbitration in Resource Sharing Systems. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTSS)*, pages 215–224, 2010. doi:10.1109/RTAS.2010.24.

[SHR18] Darshit Shah, Sebastian Hahn, and Jan Reineke. Experimental Evaluation of Cache-Related Preemption Delay Aware Timing Analysis. In *Proceedings of the 18th International Workshop on Worst-Case Execution Time Analysis (WCET)*, volume 63, pages 7:1–7:11, 2018. doi:10.4230/OASIcs.WCET.2018.7.

[SKA+14] Mladen Slijepcevic, Leonidas Kosmidis, Jaume Abella, Eduardo Quiñones, and Francisco J. Cazorla. Time-Analysable Non-Partitioned Shared Caches for Real-Time Multicore Systems. In *Proceedings of the 51st Annual Design Automation Conference (DAC)*, 2014. doi:10.1145/2593069.2593235.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[TCN00] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 101–104, 2000. doi:10.1109/ISCAS.2000.858698.

[The04] Stephan Thesing. *Safe and precise WCET determination by abstract interpretation of pipeline models.* PhD thesis, Universität des Saarlandes, 2004. doi:10.22028/D291-25797.

[TMMR19] Valentin Touzeau, Claire Maïza, David Monniaux, and Jan Reineke. Fast and Exact Analysis for LRU Caches. *Proceedings of the ACM on Programming Languages (POPL)*, 3:54:1–54:29, 2019. doi:10.1145/3290367.

[Tou19] Valentin Touzeau. *Static analysis of least recently used caches: complexity, optimal analysis, and applications to worst-case execution time and security.* PhD thesis, Université Grenoble Alpes, 2019.

[Tur37] Alan M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1), 1937. doi:10.1112/plms/s2-42.1.230.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3), 2008. doi:10.1145/1347375.1347389.

[WGR+09] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):966–978, 2009. doi:10.1109/TCAD.2009.2013287.

[Wil07] Stephan Wilhelm. Efficient Analysis of Pipeline Models for WCET Computation. In *Proceedings of the 5th International Workshop on Worst-Case Execution Time Analysis (WCET)*, volume 1, pages 37–40, 2007. doi:10.4230/OASIcs.WCET.2005.814.

[WM95] Wm. A. Wulf and Sally A. McKee. Hitting the Memory Wall: Implications of the Obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, 1995. doi:10.1145/216585.216588.

[YHZ+09] Ying Yi, Wei Han, Xin Zhao, Ahmet T. Erdogan, and Tughrul Arslan. An ILP formulation for task mapping and scheduling on multi-core architectures. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 33–38, 2009. doi:10.1109/DATE.2009.5090629.

[ZGW+17] Ying Zhang, Zhishan Guo, Lingxiang Wang, Haoyi Xiong, and Zhenkai Zhang. Integrating Cache-Related Preemption Delay into GEDF Analysis for Multiprocessor Scheduling with On-chip Cache. In *Proceedings of IEEE Trustcom/BigDataSE/ICESS*, pages 815–822, 2017. doi:10.1109/Trustcom/BigDataSE/ICESS.2017.317.

[ZK16] Zhenkai Zhang and Xenofon Koutsoukos. Cache-related preemption delay analysis for multi-level inclusive caches. In *Proceedings of the 13th International Conference on Embedded Software (EMSOFT)*, 2016. doi:10.1145/2968478.2968481.

[ZLCJ22] Wei Zhang, Mingsong Lv, Wanli Chang, and Lei Ju. Precise and Scalable Shared Cache Contention Analysis for WCET Estimation. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC)*, pages 1267–1272, 2022. doi:10.1145/3489517.3530613.

[ZW16] Yanqi Zhou and David Wentzlaff. Mitts: memory inter-arrival time traffic shaping. *ACM SIGARCH Computer Architecture News*, 44(3):532–544, 2016. doi:10.1145/3007787.3001193.

Acronyms

BCET Best-Case Execution Time.

CAC Cache Access Classification.

CFG Control-Flow Graph.

CHMC Cache-Hit-Miss-Classification.

CRPD Cache-Related Preemption Delay.

DFA Data-Flow Analysis.

EDF Earliest Deadline First.

FIFO First-In First-Out.

GEDF Global Earliest Deadline First.

ILP Integer Linear Program.

IPET Implicit Path Enumeration Technique.

LLC Last-Level Cache.

LRU Least Recently Used.

PIPT Physically Indexed, Physically Tagged Cache.

PTA Probabilistic Timing Analysis.

RM Rate Monotonic.

RTC Real-Time Calculus.

SDTA Static Deterministic Timing Analysis.

TDMA Time-Division Multiple Access.

VIPT Virtually Indexed, Physically Tagged Cache.

VIVT Virtually Indexed, Virtually Tagged Cache.

VIVU Virtual-Inlining and Virtual-Unrolling.

WCEP Worst-Case Execution Path.

WCET Worst-Case Execution Time.

WCRT Worst-Case Response Time.

List of Figures

Appendices

WCET Analysis of Shared Caches in Multi-Core Architectures using Event-Arrival Curves

A

2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)

Submitted September 25, 2022.
Accepted November 17, 2022.
Presented in Antwerp, Belgium, April 17 – 19, 2023.
DOI: 10.23919/DATE56975.2023.10137034

WCET Analysis of Shared Caches in Multi-Core Architectures using Event-Arrival Curves

Thilo L. Fischer
Institute of Embedded Systems
Hamburg University of Technology
Hamburg, Germany
thilo.leon.fischer@tuhh.de

Heiko Falk
Institute of Embedded Systems
Hamburg University of Technology
Hamburg, Germany
heiko.falk@tuhh.de

***Abstract*—We propose a novel analysis approach for shared LRU caches to classify accesses as definitive cache hits or potential misses. In this approach inter-core cache interference is modelled as an event stream. Thus, by analyzing the timing between subsequent accesses to a particular cache block, it is possible to bound the inter-core interference. This perspective allows us to classify accesses as cache hits or potential misses using a data-flow analysis. We compare the performance of the presented approach to a partitioning of the shared cache.**

***Index Terms*—shared cache, WCET analysis, multi-core**

I. Introduction

In a multi-core system, tasks are subject to inter-core interference due to the concurrent execution of multiple tasks. Specifically, data stored in a shared cache may be evicted by interfering memory accesses. As a consequence of multiple tasks competing for cache space, the worst-case execution time (WCET) is negatively impacted. To derive safe estimations of a task's WCET, the inter-core interference has to be taken into account. The standard method [1] of accounting for shared cache interference is to assume that all conflicting cache blocks may be accessed at any time by an interfering task. This is obviously pessimistic. An alternative approach is to eliminate inter-core interference by isolating tasks from each other. This can be achieved by locking the contents of the cache or by partitioning the cache to enforce isolation [2]. Another approach is to bypass the shared cache for requests on infrequently used data to avoid altering the cache's contents [3]. However, implementing such approaches requires hardware or software support and, in the case of cache partitioning, reduces the maximal cache size available to each task. Thus, to fully benefit from the presence of shared caches without restrictions on cache usage, a detailed analysis of inter-core interference is required.

We propose a novel analysis approach for inter-core interference on set-associative shared caches using the LRU replacement policy. To quantify inter-core interference, we view cache accesses issued by each core as an event stream. Thus, we can examine the inter-arrival time of cache access events. This perspective essentially induces a *time-to-live* (TTL) for information stored in the shared cache. The TTL of a cache block corresponds to the time frame in which interfering tasks can not issue a sufficient number of conflicting accesses to cause its eviction. Consequently, if a block is accessed before its TTL has expired, the access will result in a cache hit.

The key contributions of this paper are:

- We describe a novel perspective on inter-core interference for shared caches by interpreting cache accesses as event streams.
- We provide a cache hit classification criterion for shared caches accessed via a TDMA bus based on event-arrival curves.
- The technique is scalable and attains WCET performance similar to a partitioned cache.

II. Event-Arrival Curve Perspective

The system architecture considered in this paper consists of multiple cores with private caches, which are connected to a shared cache via a TDMA bus. The associativity of the shared cache is denoted by $\mathcal{A}$. Each core processes a single task $\tau \in \mathrm{T}$. We quantify inter-task interference by determining the event-arrival curves for cache access events which may evict data from the cache. The number of accesses to pair-wise different cache blocks issued from task $\tau \in \mathrm{T}$ during a time frame of Δt cycles is denoted by the event-arrival curve $\eta_\tau : \mathbb{N} \to \mathbb{N}$. Deriving the curve $\eta_\tau(\Delta t)$ from a given control-flow graph involves finding the path containing the maximal number of interfering accesses with duration $\leq \Delta t$. This problem can be solved using an IPET model. Due to space constraints, the full model is not included here. It is then possible to compute the interference caused by a task during a single TDMA slot (consisting of S cycles) by evaluating $\eta_\tau(S)$. We can thus make the following observation:

Observation 1. *The cumulative interference a task τ experiences over the course of j TDMA slots is bounded by $\gamma_\tau : \mathbb{N} \to \mathbb{N}$:*

$$\gamma_\tau(j) = j \cdot \sum_{\phi \neq \tau} \eta_\phi(S) \tag{1}$$

Note that this is a generalization of way-based cache partitioning. For the purpose of cache hit classifications, the reduced associativity in an equally partitioned cache can be represented using a constant interference function:

$$\gamma_\tau^{Part} = \sum_{\phi \neq \tau} \frac{\mathcal{A}}{|\mathrm{T}|} = \frac{|\mathrm{T}| - 1}{|\mathrm{T}|} \cdot \mathcal{A} \tag{2}$$

III. Cache Hit Classification

We now construct a cache hit classification criterion for LRU caches based on the duration (and thus the number of interfering TDMA slots) between accesses to the same cache block. The eviction distance ξ of a cache block is the minimal number of interfering accesses to cause its eviction. We call a path π between two subsequent accesses to the same cache block with $\xi(\pi) > 0$ a *potential-hit* path.

To capture the inter-task interference along a potential-hit path π, the maximal number of TDMA slots granted to interfering tasks during its execution has to be determined. The path duration $\delta(\pi)$ can be derived from a WCET analysis. Note that $\delta(\pi)$ depends on the initial TDMA offset of π computed during the WCET analysis. For a different initial offset, the path duration may diverge from $\delta(\pi)$. However, this divergence is limited to $|\mathrm{T}| \cdot S$ cycles due to the *offset relocation* Lemma from Kelter et al. [4]. These considerations give rise to an upper limit:

Lemma 1. *While executing a path π, the number of TDMA slots granted to an interfering core is limited by:*

$$J(\pi) = \left\lfloor \frac{\delta(\pi) + |\mathrm{T}| \cdot S + (S-1)}{|\mathrm{T}| \cdot S} \right\rfloor \tag{3}$$

Combining Lemma 1 and the cumulative interference function γ_τ (1) allows us to construct an *always-hit* classification criterion.

Theorem 1. *An access will always result in a cache hit if it may only be reached by traversing potential-hit paths π satisfying:*

$$\gamma_\tau(J(\pi)) < \xi(\pi) \tag{4}$$

It is now possible to check whether a particular cache access will result in a cache-hit. The paths leading to an access have to be analyzed to determine whether condition (4) holds. This can be done using a data-flow analysis. The formal definition of the data-flow analysis is not shown here due to space constraints.

IV. Evaluation

We evaluated the performance of the presented analysis approach using the WCC compiler [5]. We considered 10 dual-core and 10 quad-core systems. For each system, we randomly assigned a task from the EEMBC AutoBench 1.1 suite [6] to each core. The access timings for an L1-Hit/L2-Hit/L2-Miss were set to 1/8/20. The L1 cache was 512 bytes, direct-mapped, and block size of 16 bytes. The L2 cache was 4KB to 16KB, 8-way (dual-core) / 16-way (quad-core) associative, and block size of 32 bytes. The TDMA slots were 80 cycles long. The evaluations were conducted on an Intel Xeon Server containing 46 cores at 3.2GHz. All analyses were configured to only use a single processor core. Creating the ILPs from Section II and solving them took only 6 minutes on average for quad-core systems. The DFA to classify accesses took < 1 second for every system. We compared the computed WCET values to an equally partitioned cache. The results are shown in Fig. 1. The y-axis shows the relative WCET using a shared cache compared to the partitioning. The WCET is evaluated for each task and grouped for each system configuration. It can be seen that the performance of a shared cache analyzed using event-arrival curves tracks closely to the partitioned cache. For dual-core systems the median for all configurations is only 1.04. Overall, for quad-core systems the median WCET is 1.11. Notably, sharing the cache instead of partitioning it yielded WCET reductions of up to 26%. In that case, leveraging the timing information of cache access patterns allowed the analysis to classify 28% more accesses as cache hits than for the partitioned cache configuration.

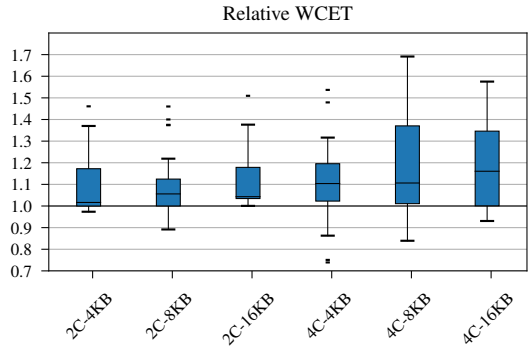


Fig. 1. Relative WCET using the event-arrival curve based classification in relation to a partitioned cache.

V. Conclusion

We presented a novel analysis approach to quantify inter-core interference on shared caches using the LRU replacement policy in multi-core architectures. By leveraging timing information and the properties of TDMA arbitration, we formulated a cache hit classification criterion for accesses to the shared cache. The evaluation showed that the event-arrival perspective reduced the unpredictability inherent to shared caches. Additional experiments could be done to further evaluate the performance of the analysis. In future work, the technique could be applied to more complex systems, e.g. multiple tasks being scheduled to run on each core or different bus arbitration techniques.

References

[1] Y. Liang, H. Ding, T. Mitra, A. Roychoudhury, Y. Li, and V. Suhendra, "Timing Analysis of Concurrent Programs Running on Shared Cache Multi-Cores," *Real-Time Systems*, vol. 48, no. 6, pp. 638–680, 2012.

[2] V. Suhendra and T. Mitra, "Exploring Locking & Partitioning for Predictable Shared Caches on Multi-Cores," in *Proc. of DAC*, 2008, pp. 300–303.

[3] D. Hardy, T. Piquet, and I. Puaut, "Using Bypass to Tighten WCET Estimates for Multi-Core Processors with Shared Instruction Caches," in *Proc. of RTSS*, 2009, pp. 68–77.

[4] T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury, "Static Analysis of Multi-Core TDMA Resource Arbitration Delays," *Real-Time Systems*, vol. 50, no. 2, pp. 185–229, 2014.

[5] H. Falk and P. Lokuciejewski, "A Compiler Framework for the Reduction of Worst-Case Execution Times," *Real-Time Systems*, vol. 46, no. 2, pp. 251–300, 2010.

[6] The Embedded Microprocessor Benchmark Consortium. About the EEMBC AutoBench™ Performance Benchmark Suite. Accessed 2022-07-05. [Online]. Available: https://www.eembc.org/autobench/

B

Analysis of Shared Cache Interference in Multi-Core Systems using Event-Arrival Curves

Proceedings of the 31st International Conference on Real-Time Networks and Systems (RTNS)

Submitted January 23, 2023.
Accepted March 17, 2023.
Presented in Dortmund, Germany, June 7 – 8, 2023.
DOI: 10.1145/3575757.3593643

Analysis of Shared Cache Interference in Multi-Core Systems using Event-Arrival Curves

Thilo L. Fischer
Hamburg University of Technology
Hamburg, Germany
thilo.leon.fischer@tuhh.de

Heiko Falk
Hamburg University of Technology
Hamburg, Germany
heiko.falk@tuhh.de

ABSTRACT

Caches are used to bridge the gap between main memory and the significantly faster processor cores. In multi-core architectures, the last-level cache is often shared between cores. However, sharing a cache causes inter-core interference to emerge. Concurrently running tasks will experience additional cache misses as the competing tasks issue interfering accesses and trigger the eviction of data contained in the shared cache. Thus, to compute a task's worst-case execution time (WCET), a safe bound on the effects of inter-core cache interference has to be determined. In this paper, we propose a novel analysis approach for shared caches using the least recently used (LRU) replacement policy. The presented analysis leverages timing information to produce tight bounds on the worst-case interference. We describe how inter-core cache interference may be expressed as a function of time using event-arrival curves. Thus, by determining the maximal duration between subsequent accesses to a cache block, it is possible to bound the inter-core interference. This enables us to classify accesses as cache hits or potential misses. We implemented the analysis in a WCET analyzer and evaluated its performance for multi-core systems containing 2, 4, and 8 cores using shared caches from 4 KB to 32 KB. The analysis achieves significant improvements compared to a standard interference analysis with WCET reductions of up to 60%. The average WCET reduction is 9% for dual-core, 15% for quad-core, and 11% for octa-core systems. The analysis runtime overhead ranges from a factor of 4× to 7× compared to the baseline analysis.

CCS CONCEPTS

• **Computer systems organization** → **Real-time systems**; • **Software and its engineering** → **Formal software verification**.

KEYWORDS

shared cache, WCET analysis, multi-core, event-arrival curve

ACM Reference Format:
Thilo L. Fischer and Heiko Falk. 2023. Analysis of Shared Cache Interference in Multi-Core Systems using Event-Arrival Curves. In *The 31st International Conference on Real-Time Networks and Systems (RTNS 2023), June 7–8, 2023, Dortmund, Germany*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3575757.3593643

RTNS 2023, June 7–8, 2023, Dortmund, Germany

ACM ISBN 978-1-4503-9983-8/23/06…$15.00
https://doi.org/10.1145/3575757.3593643

1 INTRODUCTION

In multi-core systems, the concurrent execution of multiple tasks influences the state of shared resources, such as shared busses and shared caches. A worst-case execution time (WCET) analysis has to take these effects into account. However, inter-core interference on shared caches is notoriously difficult to quantify. This unpredictability creates the potential for large over-estimation of the worst-case timing behavior. To avoid this over-estimation, a tight bound on the effects of inter-core cache interference is required.

To analyze the behavior of a cache, *cache-hit-miss-classifications* (CHMC) are used to describe whether an access will be a hit, a miss, or result in unknown behavior. For private caches, methods [4] [16] based on the well established framework of abstract interpretation [1] exist to determine these access classifications. For shared caches however, the inter-core interference has to be included in the classification process. An intuitive approach to account for inter-core interference when deriving the CHMC of an access is to consider all potentially interfering cache blocks to actually interfere with the analyzed access. The number of interfering blocks is called the *cache block conflict number* (CCN).

This approach was presented by Hardy et al. in [6] and Liang et al. in [8]. The pessimism in this approach is apparent, as interfering tasks may not access all potentially interfering cache blocks simultaneously, repeatedly, and at any point in time.

Instead of classifying accesses to the shared cache individually, the authors of [11] and [17] focused on determining an upper bound for the additional execution time caused by inter-core interference. The upper bound for the additional execution time is termed *worst-case-extra-execution-time* (WCEET) in [17]. The actual WCET of a task in a multi-core environment is then given as the sum of the WCET without interference plus the extra execution time due to inter-core interference. The problem of classifying each individual access is more complex than determining the WCEET in the sense that given a classification of individual accesses, a safe WCEET value can be determined, but not vice versa.

In this paper, we propose a novel analysis approach to derive CHMCs for individual accesses while considering inter-core interference. The approach operates on set-associative shared caches using the least recently used (LRU) replacement policy. To quantify inter-core interference, we view cache accesses issued by each core as an event stream. These event streams can be characterized using event-arrival curves. Thus, we can examine the inter-arrival time between multiple cache access events. This perspective essentially induces a *time-to-live* (TTL) for information stored in the shared

cache. The TTL of a cache block corresponds to the time frame in which interfering tasks can not issue a sufficient number of conflicting accesses to cause its eviction. Consequently, if a block is accessed before its TTL has expired, the access will result in a cache hit.

The key contributions of this paper are:

- We formulate an ILP model to derive event-arrival curves expressing inter-core cache interference.
- We provide a cache hit classification criterion for shared caches.
- We develop a data-flow analysis which leverages timing information to classify shared cache accesses as cache hits or potential misses.
- Our evaluation shows that the analysis is scalable and provides significant improvements upon the standard CCN approach.

In Section 2, related research is discussed. Section 3 gives an overview of the analysis workflow. We introduce the event-arrival perspective on cache interference in Section 4. In Section 5, a cache hit classification criterion is constructed. A data-flow analysis to discern definite hits from potential misses is formulated in Section 6. An evaluation using realistic workloads is presented in Section 7, while Section 8 concludes the paper.

2 RELATED WORK

The static analysis of worst-case behavior for complex systems is an active field of research. A survey of multi-core analysis techniques was published by Maiza et al. in [10], while analyses specifically focused on caches were surveyed by Lv et al. in [9].

Hardy et al. [6] analyzed shared caches in multi-core systems by counting the number of potentially interfering cache blocks. This value is called the *cache block conflict number* (CCN). To classify an access to a cache block as a hit, the number of conflicting blocks has to be less than the associativity minus the block age. In [8], Liang et al. combine this information with a lifetime analysis. The lifetime analysis determines which tasks are running concurrently, as tasks with a disjoint lifetime do not create any mutual interference on the cache. While this approach yields safe results, it is pessimistic as an interfering task is assumed to potentially issue all interfering accesses concurrently, at any point in time.

In a recent paper, Zhang et al. [17] aim to eliminate some of this pessimism by excluding infeasible interferences. Memory accesses are grouped based on their location in the control-flow graph. This creates a *happens-before* partial order on all accesses contained in a task. Using this ordering, infeasible combinations of interfering accesses are excluded from the interference estimation. The additional execution time caused by cache misses due to interference is computed using the cumulative execution count of all accesses to potentially evicted cache blocks. The approach is evaluated for dual-core systems and compared to the CCN approach. Using the MRTC benchmark suite [5], an average WCET reduction of 13% is reported. However, in the median only a 1% improvement is achieved.

A similar approach was used by the authors of [2] to analyze multi-threaded programs running on multi-core systems. Synchronization points between threads are used to determine which sections of the program may run in parallel in different threads. This information is then used to reduce the number of potentially occurring conflicts.

Nagar et al. [12] [11] attempt to tackle the cache interference problem from a different perspective. Nagar developed a shared cache analysis by capturing inter-task interference in an ILP model. The total amount of possible interfering accesses originating from competing tasks is statically determined and used to limit the interference experienced by the analysed task. The worst-case distribution of interfering accesses along the control-flow graph (CFG), which results in the largest increase in execution time, is then determined by solving the ILP. This approach does not depend on the exact interleaving of memory accesses issued by different tasks and yielded more precise WCET estimations than the CCN classification method.

In this paper, we pursue an orthogonal approach. Instead of limiting the total number of interfering accesses, we examine how quickly a sequence of multiple accesses may be issued.

The two techniques presented in [17] and [11] do not produce a hit or miss classification for any single cache access but only bound the overall increase in execution time for the complete program. In this paper, we tackle the harder problem of classifying each individual access as a cache hit or potential miss.

In [14] [13], Oehlert et al. present a method to quantify memory accesses issued by a task using event-arrival curves. Memory access events are derived from the program code at the level of basic blocks. To compute an upper bound on the number of events arriving in a given time frame, an IPET ILP model is developed. The event-arrival curves are used to analyze, and subsequently improved, the bus contention in multi-core systems. Based on this work, we develop an ILP model to quantify inter-core cache interference. Instead of quantifying the total number of memory accesses, we analyze how much time is required for a task to accesses a given number of interfering cache blocks.

3 ANALYSIS OVERVIEW

The system architecture considered in this paper consists of multiple cores with private caches, which are connected to a shared cache via a shared bus. We assume that each core processes a single task $\tau \in \mathrm{T}$. Consequently, inter-core and inter-task interference are synonymous in this context. The shared bus is managed using round-robin arbitration.

We analyze set-associative shared caches employing the LRU replacement policy with associativity $\mathcal{A}$. As cache-sets operate independently of one another, we may consider them in isolation during the analysis. While our analysis is general and applies to both data and instruction caches, we concentrate on instruction caches in this paper. We focus on instruction caches because the memory layout of the program code is known at compile time. This eliminates the need for a value analysis to determine the target address of data accesses.

In the remainder of this section, we provide an overview of the proposed shared cache analysis. The analysis consists of the following steps:

(1) Initial BCET and WCET analysis for each task
(2) Determining cache interference using event-arrival curves
(3) Data-flow analysis to compute cache hit classifications
(4) Final WCET analysis using the new hit classifications

Step 1. In order to perform the shared cache analysis, we require both worst-case and best-case timing information on a basic block level. Thus, as the first step an isolated timing analysis is conducted for each task. To compute WCET estimates, accesses to the shared cache are considered to be cache misses. Whereas the best-case execution time (BCET) is computed using a classical age-based abstract domain [4] without considering the impact of inter-core interference.

Step 2. Following the timing analysis, the event-arrival curves of cache access events originating from each task $\tau \in \mathrm{T}$ are derived. More precisely, it is determined how many distinct cache blocks may be accessed during a particular time frame by solving an ILP model. During this step, the BCET information is utilized to arrive at a safe upper bound. The total inter-task interference experienced by a specific task τ is then given as the sum of interference caused by all tasks running in parallel to τ.

Step 3. Based on this information, a backward data-flow analysis (DFA) is performed. The DFA investigates all accesses which potentially result in a cache hit. An access is considered as a *potential-hit*, if it would result in a cache hit without any inter-core interference. For each *potential-hit* access, there exist corresponding cache accesses which initially caused the relevant cache block to be loaded into the shared cache. The DFA determines the maximal duration between loading the block into the shared cache and subsequently accessing it. It also keeps track of any intra-task interference.

By evaluating the event-arrival curves of co-running tasks for the maximal load-access path duration, the inter-task interference can be safely bounded. Finally, potential-hits are classified as either definite cache hits or potential cache misses.

Step 4. The access classifications computed in the previous step can now be used in a WCET analysis to determine the final WCET estimate for each task.

4 EVENT-ARRIVAL CURVES FOR SHARED CACHE INTERFERENCE

In this section the cache access behavior of a task is examined and quantified. To this end, we introduce the concept of event-arrival curves for shared cache interference. The notation used in the following is shown in Table 1.

Under the LRU replacement policy, cache blocks are ordered based on which block was accessed last. Multiple accesses to the same cache block do not cause further aging of other data contained in the cache. Consequently, we define the notion of multiple events as accesses targeting a set of distinct cache blocks.

We denote the number of accesses to pair-wise different cache blocks issued from task $\tau \in \mathrm{T}$ during a time frame of Δt cycles by $\eta_\tau : \mathbb{N} \rightarrow \mathbb{N}$. A mapping $\eta_\tau(\Delta t) = n$ thus means that there exists a scenario in which τ will access n different cache blocks during a time frame of Δt cycles.

Table 1: Variables and Notation

Symbol	Meaning
T	Set of tasks
$\mathcal{A}$	Associativity of the shared cache
Acc	Set of all cache accesses
$\mathcal{B}$	Set of all cache blocks
$cb : Acc \rightarrow \mathcal{B}$	Mapping of accesses to the target block
$q_v \in \mathbb{N}$	Execution count of node v
$z_v \in \mathbb{N}$	Execution time contribution from v
$t_b \in \{0, 1\}$	Indicator whether $b \in \mathcal{B}$ is accessed

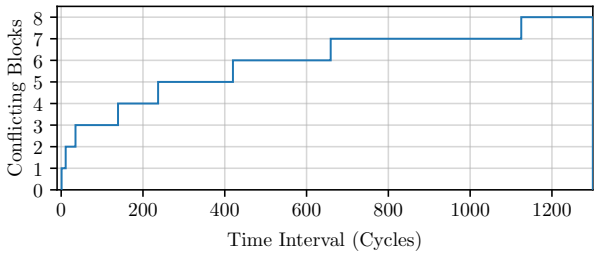


Figure 1: Example for an event-arrival curve expressing shared cache interference. Derived from the `canrdr01` benchmark of the EEMBC AutoBench suite [15].

In Figure 1 an example for such an event-arrival curve is shown. The time interval is shown on the x-axis, while the y-axis represents how many different cache blocks may be accessed. In this example, the interference grows quickly in the beginning. After a time frame of around 140 cycles, the task may have issued accesses to 4 different cache blocks. However, the time interval required to observe a larger amount of interference is significantly higher. To increase the observed interference from 6 to 8 conflicting blocks takes around 700 cycles. This clearly demonstrates the pessimism in the currently available analysis techniques [8] [11] [17]. All these techniques assume that every conflicting block may be accessed instantaneously by an interfering task. However, we can see here that it takes a substantial amount of time for the task to generate the traffic on the shared cache.

We will now show how such an event-arrival curve can be derived from a task's control-flow graph. To derive an event arrival curve from the CFG of a task τ, we have to associate cache accesses to the nodes in the CFG (V, E), where V contains the nodes in the graph and E are the edges connecting the nodes. For data caches, this relation arises from the memory-accessing instructions contained in the program and their respective target addresses. For

instruction caches, cache accesses originate from fetching operations in the processor pipeline. Thus, we have to consider the pipeline behavior.

We denote the set of all cache blocks by $\mathcal{B}$. Let $\rightsquigarrow \subset V \times \mathcal{B}$ be a relation that contains the pair (v, b) iff executing the node $v \in V$ may cause an access to the block $b \in \mathcal{B}$. A path π is a sequence of nodes $\pi = (v_1, \ldots, v_p)$ with $(v_i, v_{i+1}) \in E,\ i \in \{1, \ldots, p-1\}$. The execution of a path π in the CFG of τ causes other tasks to experience n interfering cache accesses, as given in (1).

$$n = \left| \bigcup_{v \in \pi} \{b \in \mathcal{B} \mid v \rightsquigarrow b\} \right| \tag{1}$$

Note that η_τ is a step function and we are only interested in the arrival of the first $\mathcal{A}$ events (as after $\mathcal{A}$ events all blocks are evicted from an LRU cache). Thus, we can capture a task's cache access behavior by deriving the minimal time required to access 1, 2, ..., $\mathcal{A}$ distinct cache blocks. These values correspond to the location of the steps in Figure 1.

To compute the time required for a particular number of accesses, we utilize an IPET model [7] [14]. As the basic construction of such a model is out of the scope of this paper, we focus only on the additional constraints required to model cache interference.

For each cache block, we introduce a binary decision variable to indicate whether this cache block is accessed on the considered path. The variables are denoted by t_b for $b \in \mathcal{B}$. The indicator variables t_b are constrained by (2) and (3), where q_v is the variable containing the execution count of v.

$$0 \leq t_b \leq 1 \tag{2}$$

$$t_b \leq \sum_{v \rightsquigarrow b} q_v \tag{3}$$

Thus, if any node is executed which may access the block b, the indicator variable t_b may take the value 1, otherwise it is set to 0.

To determine the minimal number of cycles during which n cache blocks may be accessed, only paths containing accesses to at least n cache blocks are considered (4). In the model, n is a parameter which may be set to values 1, ..., $\mathcal{A}$ to determine the time span required for different levels of interference.

$$n \leq \sum_{b \in \mathcal{B}} t_b \tag{4}$$

Consequently, the objective of the ILP is to minimize the number of cycles required to cause accesses to n distinct cache blocks (5). The path duration is computed as the sum of the execution time contributions z_v of each node $v \in V$.

$$\min \sum_{v \in V} z_v \tag{5}$$

The execution time contribution z_v depends on the best-case execution time of the node v and the execution count q_v. However, as we do not make assumptions about the distribution of events inside the nodes, the time contributions of the first and last nodes of the (partial) path is reduced to a single cycle. Solving this ILP for all parameter values $1 \leq n \leq \mathcal{A}$ then allows us to construct the event-arrival curve for each task. As noted before, cache sets

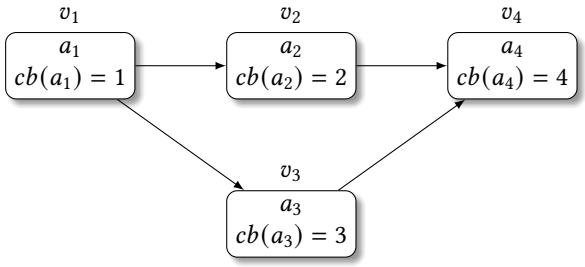


Figure 2: Example CFG containing four accesses targeting different cache blocks. The accesses a_2 and a_3 are mutually exclusive.

are analyzed independently of each other, thus event-arrival curves need to be derived for each cache set.

Note that, by determining interference on the basis of paths in the CFG, we implicitly eliminate any mutually exclusive accesses from the interference computation. For example, consider the control-flow graph shown in Figure 2. The nodes $v_1, \ldots, v_4$ each contain an access targeting a different cache block. The IPET model will consider the two paths (v_1, v_2, v_4) and (v_1, v_3, v_4). Hence, the maximal interference caused by this CFG is 3 blocks, either the set $\{1, 2, 4\}$ or $\{1, 3, 4\}$ is accessed. The blocks 2 and 3 are never accessed together on the same path. Such mutually exclusive access behavior is therefore safely excluded from the event-arrival curves, leading to a tight estimation of the possible interference.

The cumulative interference experienced by the task τ can be safely estimated by the addition of the curves η_ϕ from co-running tasks $\phi \in \mathrm{T} \setminus \{\tau\}$. These considerations yield (6) as a safe approximation of the inter-task interference depending on the duration l between subsequent accesses to a particular cache block.

$$\gamma_\tau(l) = \sum_{\phi \neq \tau} \eta_\phi(l) \tag{6}$$

Using the cumulative interference function γ_τ, the time-to-live of a cache block $b \in \mathcal{B}$ can then be expressed as a function of its age. Here, we use the word *age* to refer to the number of conflicting blocks which have been accessed by τ since the last access to b, without considering the inter-task interference. We denote this function as $\mathrm{TTL}_\tau : \{0, \ldots, \mathcal{A}-1\} \to \mathbb{N}_0 \cup \{\infty\}$.

$$\mathrm{TTL}_\tau(age) = \sup_{0 \leq l} \{l \mid \gamma_\tau(l) < \mathcal{A} - age\} \tag{7}$$

Note that the TTL of a block may be $+\infty$ cycles in case the interfering tasks never issue enough interfering accesses to trigger the eviction. We utilize the notion of cumulative interference in the next section to derive cache hit classifications.

5 CACHE HIT CLASSIFICATION

In this section, we construct a cache hit classification criterion based on the duration between accesses to the same cache block. We call the set of cache accesses contained in the programs *Acc*. The *cache-access-classification* (CAC) is denoted by the mapping $\mathrm{CAC} : Acc \to \{A, N, U\}$. The CAC signifies whether the access

will reach the shared cache always (A), never (N), or whether the behavior is uncertain (U).

We will now examine the intra-task interference between two accesses to the same cache block. For this purpose, we can abstract a path in the CFG to a sequence of accesses $(a_i)_{i=1}^m$.

Definition 1. *The intra-task interference on a path between two accesses a_1 and a_m to the same cache block $cb(a_1) = cb(a_m)$ is given by:*

$$int((a_i)_{i=1}^m) = \left\{ cb(a_s) \;\middle|\; \begin{array}{l} 1 \le s \le m : \\ cb(a_s) \neq cb(a_m) \wedge \\ CAC(a_s) \neq N \end{array} \right\} \tag{8}$$

The set consists of all cache blocks which may be accessed in the sequence $(a_i)_{i=1}^m$ that conflict with the target cache block $cb(a_m)$. Given the associativity $\mathcal{A}$ of the shared cache, we can define the eviction distance along a path.

Definition 2. *The eviction distance ξ of a cache block $cb(a_m)$ along a path with access sequence $(a_i)_{i=1}^m$ is the minimal number of additional interfering cache accesses which could lead to a cache miss for a_m as given in (9).*

$$\xi((a_i)_{i=1}^m) = \mathcal{A} - \left| int((a_i)_{i=1}^m) \right| \tag{9}$$

As mentioned in Section 3, we want to analyze paths which may lead to a cache hit on the shared cache. These paths are characterized by a positive eviction distance. To denote such paths we introduce the notion of a potential-hit path.

Definition 3. *A path with associated cache access sequence $(a_i)_{i=1}^m$ is called a potential-hit path for the access a_m iff:*

$$CAC(a_m) \neq N \tag{10a}$$

$$cb(a_1) = cb(a_m) \wedge CAC(a_1) = A \tag{10b}$$

$$0 < \xi((a_i)_{i=1}^m) \tag{10c}$$

Equation (10) contains the requirements for the access a_m to potentially result in a hit on a particular path: *(a)* the access a_m may reach the shared cache, *(b)* the targeted cache block is loaded into the cache by a_1, *(c)* intra-task interference will not cause the eviction of $cb(a_m)$.

The only missing piece to classify the access a_m as a cache hit is to check whether the inter-task interference is less than the eviction distance. As seen in (6), using event-arrival curves, the inter-task interference may be quantified as a function of time. By evaluating γ_τ for the WCET of the considered path, a safe estimate of the inter-task interference can be made. We can thus construct a cache hit classification which depends on the temporal reuse distance of the cache-block and the interference function derived from the event-arrival curves discussed in Section 4.

Theorem 1. *An access $a \in Acc$ will always result in a cache hit if it may only be reached by traversing potential-hit paths π with access sequence $(a_i)_{i=1}^m$, $a_m = a$ satisfying:*

$$\gamma_\tau(WCET(\pi)) < \xi((a_i)_{i=1}^m) \tag{11}$$

Proof. Consider the scenario that the access a could result in a cache miss. This may happen for three different reasons: The targeted cache block was *(a)* not loaded into the shared cache previously, *(b)* evicted due to intra-task interference, or *(c)* evicted due to inter-task interference.

However, all executions containing a must load the targeted cache block into the cache and this block will not be evicted due to intra-task interference as π is a potential-hit path. Furthermore, $\gamma_\tau(\mathrm{WCET}(\pi))$ is a safe upper bound on the number of aging events due to interfering accesses from competing tasks. As (11) requires that the eviction distance is strictly greater than the interference, the cache block $cb(a)$ will not be evicted due to inter-task interference. Thus, the access will always result in a cache hit. □

Note that the condition given in (11) can be written equivalently using the time-to-live function:

$$\mathrm{WCET}(\pi) \le \mathrm{TTL}_\tau(|int((a_i)_{i=1}^m)|) \tag{12}$$

At this point, we are able to quantify cache interference using event-arrival curves and have formulated a sufficient condition to classify an access as a hit. What is missing now is an algorithmic description on how we can efficiently use these two components to derive a classification for every access.

6 PATH ANALYSIS

In this section, we utilize Theorem 1 to determine whether accesses to the shared cache definitively result in a cache hit. To this end, we perform a data-flow analysis. In the DFA, we examine accesses which would result in a hit provided no inter-core interference is present. Checking the condition given in (11) for a particular access $a \in Acc$ requires knowledge of the WCET of potential hit-paths leading to the access and their respective eviction distance. Consequently, we may abstract a path from a concrete sequence of nodes to a safe upper bound on its execution time and a set of potentially accessed conflicting blocks.

Thus, path information for access classification can be represented in an abstract domain using a semi-lattice $D = (\mathbb{N} \times 2^{\mathcal{B}}) \cup \{\bot, \top\}$. Elements $(l, C) \in \mathbb{N} \times 2^{\mathcal{B}}$ represent a maximal path duration of l cycles and intra-task interference $C \subseteq \mathcal{B}$. $\top$ corresponds to a potential cache miss, while $\bot$ is used to signal a finished load-access path. I.e. another access will load $cb(a)$ into the shared cache en route to the access a, resulting in a cache-hit. The tuples are intuitively ordered $(l_1, C_1) \le (l_2, C_2) \iff l_1 \le l_2 \wedge C_1 \subseteq C_2$, while $\top$ is the greatest element and $\bot$ is the least element. Joining of two elements is performed by the $\sqcup$ operator as defined in (13).

$$(l_1, C_1) \sqcup (l_2, C_2) = (\max(l_1, l_2), C_1 \cup C_2) \tag{13a}$$

$$d \sqcup \top = \top,\ d \sqcup \bot = d,\ d \in D \tag{13b}$$

To compute the data-flow information for all nodes in the control-flow graph, we conduct a data-flow analysis in the backward direction. To formalize this DFA, we specify the data-flow information as the mappings $in[v] : Acc \to D$ and $out[v] : Acc \to D$ for $v \in V$. Although the analysis is conducted in the backward direction, we use the word *in* (*out*) to denote the data-flow information at the beginning (end) of a node in the regular sense.

Every node $v \in V$ possesses an associated sequence of cache accesses, which is denoted using the superscript v, $(a_i^v)_{i=1}^m$. The

impact of a single cache access a_i^v issued during the execution of v on the analyzed access a is determined by the function f_i^v (14).

$$f_i^v(a,(l,C)) = \begin{cases} (a,\top) & \text{if } \gamma_\tau(l) \geq \mathcal{A} - |C'| \\ (a,\bot) & \text{else if } \mathrm{CAC}(a_i^v) = A \wedge \\ & \quad cb(a) = cb(a_i^v) \\ (a,(l,C')) & \text{else} \end{cases} \tag{14a}$$

$$f_i^v(a,\top) = (a,\top),\ f_i^v(a,\bot) = (a,\bot) \tag{14b}$$

where

$$C' := C \cup \left\{ cb(a_i^v) \;\middle|\; cb(a) \neq cb(a_i^v) \wedge \mathrm{CAC}(a_i^v) \neq N \right\} \tag{15}$$

The first case in (14a) covers the situation in which the interference on the shared cache is too high to guarantee a cache hit. Thus, the value is updated to $\top$, showing that this access is a potential miss. In the second case, the path duration is acceptable and the access a_i^v actually causes the target block $cb(a)$ to be loaded into the cache. The value is updated to $\bot$ to show that the load-access path is terminated at this point. In the final case, the data-flow information is propagated further with the updated set of conflicting cache blocks C' given in (15). The functions f_i^v can be composed to operate on sequences of accesses $f_{\alpha,\ldots,\beta}^v = f_\alpha^v \circ \ldots \circ f_\beta^v$.

Data-flow information for accesses contained in v is initially generated by $G[v]$ as defined in (16).

$$G[v] = \left\{ f_{1,\ldots,(t-1)}^v(a_t^v,(\mathrm{WCET}(v),\emptyset)) \mid 1 \leq t \leq m \right\} \tag{16}$$

The initial path duration is approximated by $\mathrm{WCET}(v)$. Starting from the empty set, the blocks conflicting with a_t^v are derived from the event sequence $(a_i^v)_{i=1}^{t-1}$ by using the propagation function $f_{1,\ldots,(t-1)}^v$.

Data-flow information arriving at v from successor nodes is contained in $out[v]$. This information is propagated backwards along v by $P[v]$ as shown in (17).

$$P[v] = \left\{ f_{1,\ldots,m}^v(a,(l+\mathrm{WCET}(v),C)) \mid (a,(l,C)) \in out[v] \right\} \tag{17}$$

The propagation function $f_{1,\ldots,m}^v$ is applied to the accesses contained in $out[v]$ which are not mapped to $\top$ or $\bot$. The path duration l is increased to $l + \mathrm{WCET}(v)$ to reflect the fact that it may increase by up to $\mathrm{WCET}(v)$ cycles by extending the path over v.

The incoming data-flow information for node v is the combination of $G[v]$ and $P[v]$ (18), whereas the outgoing data-flow information consists of the merged information from all successors w (19).

$$in[v] = G[v] \sqcup P[v] \tag{18}$$

$$out[v] = \bigsqcup_{(v,w)\in E} in[w] \tag{19}$$

Here, the join operation $\sqcup$ is extended to data-flow mappings by pair-wise joining of elements associated to the same access event. The values of $in[v]$ and $out[v]$ are computed iteratively until they stabilize. Then, an access a can be safely classified as a cache hit if no node v exists with $(a,\top) \in in[v]$.

6.1 Ensuring Termination

In its current formulation, the semi-lattice D used in the DFA has infinite height, as the path duration is not limited. Therefore, D does not satisfy the ascending chain condition. It is not guaranteed that the data-flow information converges after a finite number of iterations.

We know that all feasible paths contained in the analyzed CFG are of finite duration as we assume that all tasks terminate and thus have a finite WCET. In practice, however, non-termination of the analysis can occur when information is propagated along a path that is infeasible in reality.

To correct this behavior, it is possible to limit the maximal duration value in the abstract domain. Instead of allowing the path duration to take an arbitrary duration $l \in \mathbb{N}$, only a limited interval $[1,\ L_\tau] \subset \mathbb{N}$ can be permitted. A natural choice for L_τ is the smallest duration to experience the maximal interference as given in (20).

$$L_\tau = \min_{l\in\mathbb{N}} \{ l \mid \gamma_\tau(l) = \gamma_\tau(\mathrm{WCET}(\tau)) \} \tag{20}$$

In case a path duration exceeds this value while being propagated along a node, the duration is instead capped at the upper limit L_τ. This limit on the path duration is safe, as it never underestimates the potential inter-task interference due to the choice of L_τ. Using this upper limit on the path duration, only a finite number of updates may be applied to an abstracted path until the value necessarily stabilizes.

While termination is now ensured, in practice, the number of iterations required for termination may still be prohibitively large. To solve this problem, we widen an abstracted path (l,C) to $\top$ after the number of performed propagations over nodes exceeds a certain threshold value. This procedure is safe as we do not introduce faulty cache hit classifications here. However, some precision is sacrificed. We evaluate the impact of different threshold values in Section 7.3.

6.2 Relative Precision

In this section, we make a few notes about the relative precision of the presented analysis compared to the baseline analysis [8]. We use the abbreviation CCN to refer to the baseline method which uses the cache block conflict number to derive cache hit classifications. For brevity, we denote the method presented in this paper as the EAC method.

It is clear that the quantification of interference over time using event-arrival curves can yield more precise results than assuming that every conflicting block causes interference at all points in time. However, the EAC classification approach presented in this paper is not strictly more precise than the CCN method.

Consider the control-flow graph in Figure 3 as an example. The graph contains three nodes x, y, z. They contain accesses a and b in the nodes x and z respectively.

In this scenario, the access a causes the block $cb(a)$ to be loaded into the shared cache. The node y does not contain any accesses to the shared cache. Thus, when performing the access b to $cb(b) = cb(a)$, no further blocks were loaded into the cache by the analyzed task. Thus, the block age from the isolated perspective is 0. The CCN analysis can now determine whether a hit classification is appropriate using the maximal number of interfering cache blocks.

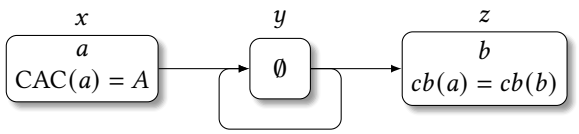


Figure 3: Example CFG containing two accesses targeting the same cache block.

The DFA utilized in the EAC approach, however, has to process the loop which allows the node y to be executed multiple times between x and z. The path information originating from b can potentially be propagated many times along the loop y. Depending on the propagation threshold, the DFA may stop and widen the result of b to a potential miss. Thus, the EAC technique is not strictly more precise compared to CCN.

However, it is possible to combine the two techniques to create a more precise method. To consider a cache access as a hit, it is sufficient that either the CCN analysis or the EAC analysis produces a hit classification. To combine the two classifications approaches, first, the EAC classification is determined. If the result is not conclusive the information of the CCN analysis can be utilized. We refer to the combination of the EAC and CCN classifications as EAC+. We evaluate the relative precision of the EAC and CCN analyses in practice in Section 7.4.

6.3 Iterative Application

The DFA described in this section contains an implicit dependence between the classification of different accesses. To determine a safe upper bound for the duration of a path, the WCET of the contained nodes is accumulated. Initially, these WCET values are derived under the assumption that no access to the shared cache will result in a cache hit. However, after classifying some accesses as definitive cache hits, these WCET values may be reduced. The DFA can then be performed a second time using the tighter WCET values. Accesses which were previously classified as potential misses may now be classified as cache hits due to the shorter duration of the potential-hit paths. Hence, it is possible to apply the DFA iteratively to gain more and more precise cache hit information.

7 EVALUATION

To evaluate the performance of the novel classification approach, we implemented it as a module in the WCC compiler [3]. The target architecture consisted of multiple ARM7TDMI cores with private L1 caches connected to a shared L2 cache using a round-robin arbitrated bus. We considered systems containing 2, 4, and 8 cores. For all core counts, we analyzed 10 systems. For each system, we randomly assigned a task to each core. The tasks were taken from the EEMBC AutoBench 1.1 benchmark suite [15] to create a realistic workload. Virtual inlining and unrolling was limited at a depth of 3, respectively.

For the private L1 caches, we set the cache size to 256 bytes, direct-mapping and a cache block size of 16 bytes. Shared cache sizes of 4 KB to 32 KB were evaluated, with 8-way associativity, and cache block size of 64 bytes. The caches used the LRU replacement policy. The instruction access timings were set to 1 cycle for an L1 hit, 10 cycles for an L2 hit and 40 cycles in case of an L2 miss, excluding potential bus access delays. As we focus on instruction caches in this evaluation, we assume that each data access takes 3 cycles.

Table 2: Worst-case access timing including the bus access delay using round-robin arbitration.

Cores	L2 Hit	L2 Miss	Hit-Miss Ratio
2	50	80	0.63
4	130	160	0.81
8	290	320	0.91

An instruction access may be stalled for up to $(|T| - 1) \cdot 40$ cycles at the shared bus due to accesses from other cores. The worst-case access timing thus consists of the sum of the bus access delay and the L2 access time. Table 2 shows the different timing values including the worst-case bus stall time. It can be seen, that the difference between a cache hit and cache miss shrinks for higher core counts, as the worst-case access delay is predominantly determined by the delay to access the shared bus. Thus, improvements in the cache hit classifications may have a smaller than expected impact on the WCET of a task, as the WCET is also impacted by other factors such as the bus access delay.

7.1 Evaluation Results

In this subsection, we evaluate the performance of the presented EAC+ analysis. As the reference point, we also analyze the systems using the method described in [8]. This baseline is abbreviated as the CCN analysis.

To evaluate the performance, we utilize two different metrics. The first metric is the percentage of accesses contained in the CFG that could be classified as a cache hit. The second metric used in this evaluation is the relative WCET achieved by the EAC+ analysis compared to the CCN approach as the reference value.

We use the hit ratio in addition to the relative WCET, as the WCET value does not capture improved classifications outside the critical path. Additionally, changes in WCET might be small, even though a significant number of accesses were newly classified as cache hits as the WCET is also influenced by other factors such as the bus access delay.

We use box plots to visualize the results. The line in the middle of each box represents the median value, while the lower and upper bounds of a box show the 25th and 75th percentile. The whiskers are at most 1.5 times as large as the central box. Data points outside this range are considered outliers and are marked by a small dash. The metrics are evaluated for each task and grouped for each system configuration. The configuration groups are named using the number of cores (2, 4, or 8) and the size of the shared cache (4, 8, 16, or 32 KB).

7.1.1 Dual-Core Systems. Figure 4 shows the hit ratio for dual-core systems. For the 4 KB cache, the median hit ratio is 0% for the baseline CCN analysis. Here, the EAC+ analysis achieves significant improvements with a median hit ratio of 76%. For the 8 KB cache, the

CCN analysis is able to achieve a hit ratio of 74%. Again, the EAC+ analysis is able to outperform the CCN method and yields a median hit ratio of 88%. In the largest cache configuration for dual-core systems, the EAC+ approach achieves only a small improvement over the CCN approach. Both classification methods achieve a hit ratio of around 90%. This result indicates that a 16 KB cache is large enough for the considered dual-cores systems relative to the contained code size so that inter-core interference does not lead to a substantial degradation of the worst-case timing behavior.

In Figure 5, the WCET of the EAC+ analysis is compared to the CCN analysis. The first three box-plots show the results for dual-core systems. For the smallest cache size of 4 KB, the EAC+ analysis yields an average WCET improvement of 14.4% (11.2% median). Using a larger cache, this gap shrinks as the CCN analysis is able to classify more accesses as cache hits. For an 8 KB shared cache, the average WCET reduction is 9.8% (2.8% median). In the largest cache configuration of 16 KB, no median WCET reduction is achieved. Again, this is due to the fact that the 16 KB cache is large enough to completely contain the code for most dual-core systems.

However, there is still an outlier which experienced a WCET reduction of 60%. This outlier occurred in a system containing the tasks `aiifft01` and `bitmnp01`. The code size of `bitmnp01` is around 27 KB. Thus, the CCN approach is unable to classify any accesses belonging to `aiifft01` as a cache hit. However, the EAC analysis was able to classify 95% of accesses contained in `aiifft01` as cache hits. This discrepancy leads to the large reduction in WCET.

7.1.2 Quad-Core Systems. A different landscape unfolds for systems containing four cores. As can be seen in Figure 6, the cumulative code size of the tasks contained in the systems is so large that it prohibits the CCN analysis from making any useful conclusions about cache hits. The median hit ratio is 0% even for the largest 16 KB cache. In contrast to this, the EAC+ analysis presented in this paper is still able to classify many accesses as cache hits. The median hit ratio values for 4 KB caches is 41%; for 8 KB it is 71%; and for 16 KB it is 78%. This substantial improvement in access classification is also reflected in the relative WCET values, shown in Figure 5. The three box plots on the right show the WCET reduction for quad-core systems. The EAC+ analysis reduces the WCET by 10.2% on average (4.0% median) for 4 KB caches. The performance gap increases to an average 17.5% (11.5% median) for 8 KB caches. For 16 KB caches, the average relative WCET reduction is 16.7% (12.2% median).

These results demonstrate the necessity of a precise analysis of the inter-task cache interference. Without precise information on the potential cache interference, the cache becomes useless in the worst-case as almost no access can be classified as a cache hit using the standard classification method.

7.1.3 Scalability to Octa-Core Systems. To explore the scalability of the analysis, we also performed an evaluation of octa-core systems. The size of the shared cache was increased to 32 KB for this evaluation. All other parameters remained unchanged. We evaluated 10 task sets. The results are shown in Figure 7.

As was the case for quad-core systems, the CCN analysis was not able to classify any significant amount of accesses as cache hits. The hit ratio for almost all tasks is 0%. Again, the EAC+ analysis is able to perform substantially better with a median hit ratio of 49.6%.

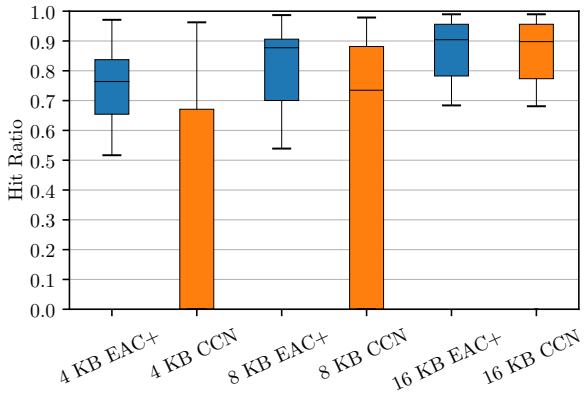


Figure 4: Dual-core cache hit ratio of the EAC+ (blue) and CCN (orange) analyses for different cache configurations.

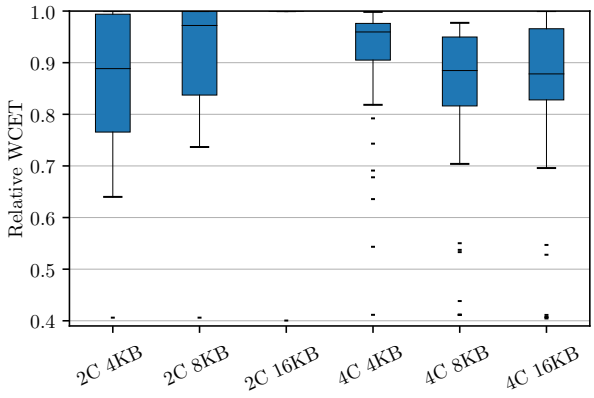


Figure 5: Relative WCET values derived using the EAC+ analysis in relation to the CCN analysis.

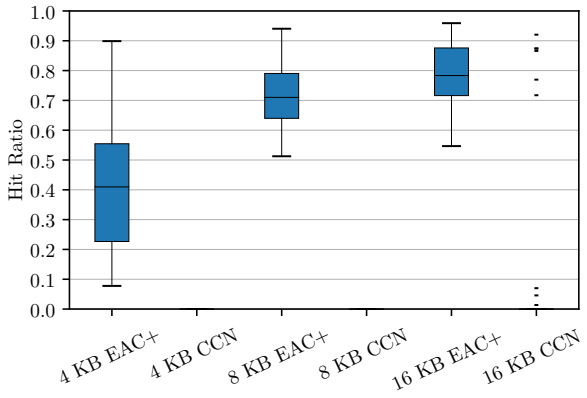
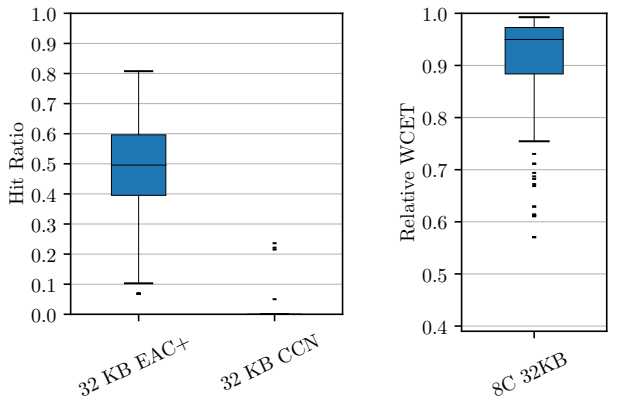


Figure 6: Quad-core cache hit ratio of the EAC+ (blue) and CCN (orange) analyses for different cache configurations.



(a) Hit ratio **(b) Relative WCET**

Figure 7: Hit ratio of the EAC+ (blue) and CCN (orange) analysis, as well as the relative WCET for octa-core systems.

This increase in hit classifications resulted in an average relative WCET reduction of 11.3% (5.1% median), while the maximal WCET reduction for a task was 43%.

Hence, we conclude that the proposed analysis is also applicable to multi-core systems with a high core count and yields notable improvements over the baseline analysis.

7.2 Runtime Evaluation

In this subsection, we take a look at the runtime overhead that is required by the proposed analysis. The evaluations were conducted on an Intel Xeon Server containing 46 cores running at 3.2GHz. All analyses were configured to use only a single processor core.

The runtime of the EAC+ analysis consists of *(a)* the BCET/WCET analysis used to derive the event-arrival curves and required to perform the DFA, *(b)* the derivation of the event-arrival curves, *(c)* the DFA to classify accesses, *(d)* the final WCET analysis. The runtime of the CCN method includes determining the cache block conflict number for every cache set and a WCET analysis. A table containing the measured runtimes is shown in Table 3.

As the analysis is based upon an isolated per-core WCET analysis it is expected that the runtime scales linearly in the number of cores. Additionally, the size of the shared cache also has a linear impact on the number of ILPs that potentially need to be solved to determine the event-arrival curves. These expectations are matched in the measured runtimes. The time required on average for an analysis of a dual-core system ranges from 3.5 to 4.9 minutes. We observed a small decrease in the runtime of the 16 KB configuration compared to the 8 KB configuration. This runtime decrease occurred during the derivation of the event-arrival curves. As the cache size increased, the traffic on each individual cache set decreased. This resulted in a more efficient event-arrival curve derivation process and caused a small drop in the required analysis time. For quad-core systems the runtime increased to 7.9 to 11.9 minutes. The average runtime of an 8-core system was 24.7 minutes. Note that the 8-core systems were also equipped with a larger 32 KB shared cache.

Table 3: Average analysis runtime in minutes.

Cores	Cache Size	EAC+	CCN
2	4 KB	3.5	0.5
	8 KB	4.9	0.8
	16 KB	4.4	0.9
4	4 KB	7.9	1.2
	8 KB	11.8	1.9
	16 KB	11.9	2.2
8	32 KB	24.7	6.1

In the EAC+ analysis, a large proportion of the time was spent to solve the ILPs formulated in Section 4 to derive the event-arrival curves. As the ILPs are independent of each other, this step could be parallelized to reduce the time requirement. The data-flow analysis presented in Section 6 was very quick, with an average runtime of less than one second per system.

The average EAC+ analysis runtime is larger than the time required by the CCN method by a factor of 4× to 7×. This increase in runtime is however justifiable, as the CCN method did not produce any useful cache hit classifications for 5 out of the 7 system configurations. Using the CCN Classifications, the median hit ratio was higher than 0% only for dual-core systems with a cache size of 8 KB and 16 KB. The presented analysis technique, however, achieved significant improvements in the number of hit classifications and the WCET estimate.

7.3 Propagation Limit Sensitivity

To ensure that the data-flow analysis presented in Section 6 converges quickly, the propagation of abstracted path information over nodes in the CFG is limited by a threshold value as described in Section 6.1. When the number of nodes a path is propagated over exceeds the threshold value, the path is considered to potentially cause a cache miss. In this subsection, we explore whether this cut off threshold is necessary and how it affects the analysis precision.

To achieve the results shown in the previous sections, we utilized a threshold value of 30. This means, that potential-hit paths containing up to 30 nodes may be recognized by the DFA as resulting in a cache-hit. Paths were not investigated beyond this threshold and were considered a potential miss instead. To explore the sensitivity of the analysis to this parameter, we also performed the analysis of dual-core and quad-core systems with different threshold values.

We first decreased the propagation limit to 5 nodes. As only short hit paths may be recognized with this setting, a decrease in the number of successful cache hit classifications is expected. For quad-core systems with a 16 KB shared cache, the largest average hit ratio reduction occurred. The average hit ratio reduced by 1.5% from 78.3% to 76.8%. Thus, a lower threshold only has a minor impact on the precision.

To check whether a higher propagation limit is useful, we tried increasing the limit to 150. The biggest increase in the average hit ratio was observed for dual-core systems with a 4 KB cache.

The hit ratio increased by 0.5% from 74.9% to 75.4%. These results suggest that accesses to the shared cache exhibited highly localized behavior.

The average runtime of the data-flow analysis for quad-core systems was 1.2 seconds with the propagation limit set to 30. Decreasing the limit to 5 reduced the runtime to 0.24 seconds. While increasing the limit to 150 caused the analysis to require 9.6 seconds on average. Removing the propagation limit altogether caused 8 of the 30 analyzed quad-core systems to not terminate after 2 hours. Thus, we conclude that the propagation limit is necessary, but the analysis precision itself is not very sensitive to the choice of the parameter value.

7.4 Relative Precision Evaluation

In Section 6.2, we note that the precision of the EAC and the CCN analysis techniques are not comparable as there are situations in which either of the two analyses is more precise than the other one. To evaluate whether this theoretical consideration manifests itself in practice, we also performed the evaluation of dual-core and quad-core system using only the classifications derived by the DFA without using the CCN classification as the fallback.

For dual-core systems, we observed a difference in the analysis precision between the different classification strategies. The median hit ratio of the EAC classification for 4 KB caches was 73%. This is similar to the 76% which were achieved by EAC+. The median hit ratio of CCN for this configuration was 0%. For 8 KB caches, the median hit ratio using EAC was 77%. Recall that the CCN approach yielded a 74% median hit ratio, while the combination of the two classification methods yielded a median hit ratio of 88%. The combination of the two hit classification methods thus performed better than each method did when applied separately.

For the largest cache configuration of 16 KB, the CCN analysis classified more accesses as cache hits than EAC. The median hit ratio for EAC was 82%, while both EAC+ and CCN achieved around 90%. Thus, EAC performed notably better than CCN for the smallest cache size, while CCN had higher performance than EAC for the 16 KB cache.

For quad-core systems, however, we observed that there was no significant difference between the performance of the EAC and EAC+ classifications. The reason for this is that while the CCN analysis may be more precise in some situations in theory, for the evaluation setup shown in this paper, the CCN analysis was unable to generate hit classifications in almost all situations (see Figure 6). Thus, for quad-core systems, solely using the EAC classifications instead of the EAC+ combination, which also makes use of the CCN classifications, did not result in decreased analysis precision.

These results demonstrate that there are situations in which either the EAC or CCN technique performs better, while the combination of both classification approaches EAC+ performs the best.

8 CONCLUSION

In this paper, we presented a novel analysis perspective for shared caches accessed via a round-robin arbitrated bus. In the analysis, the inter-task interference between tasks running on different cores is expressed using event-arrival curves. These curves quantify how much time will pass between accesses to multiple conflicting cache blocks. This perspective enables us to view inter-task cache interference as a function of time. Furthermore, we presented a data-flow analysis with which the temporal reuse distance of a cache block can be determined. These two components, the event-arrival curves and the temporal reuse distance, allow the analysis to derive safe cache hit classifications for individual accesses to the shared cache.

We evaluated the performance of the analysis using realistic workloads from the EEMBC AutoBench 1.1 benchmark suite. We evaluated systems containing 2, 4, and 8 cores and shared caches ranging from 4 KB to 32 KB. The presented analysis significantly outperforms the baseline analysis [8] in most situations. The baseline analysis collects all potentially accessed cache blocks and assumes that these blocks may be accessed at any time. This pessimistic assumption caused the analysis to not produce any substantial amount of hit classifications in 5 out of the 7 considered system configurations. Compared to this standard analysis, the presented analysis performed particularly well in systems with a small cache size relative to the total program size.

Further research could expand the work presented in this paper by analyzing systems containing more than one task per core. Another avenue for future research is the applicability of event-arrival curves to replacement policies other than LRU. To reduce the required runtime of the analysis, it is conceivable to compute safe approximations of the event-arrival curves instead of precise curves. This trade off between runtime and precision could be explored in future work.

REFERENCES

[1] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 1977)*. 238–252. https://doi.org/10.1145/512950.512973

[2] P. Padma Priya Dharishini and P. V. R. Murthy. 2021. Precise Shared Instruction Cache Analysis to Estimate WCET of Multi-threaded Programs. In *2021 IEEE 18th India Council International Conference (INDICON)*. 1–7. https://doi.org/10.1109/INDICON52576.2021.9691620

[3] Heiko Falk and Paul Lokuciejewski. 2010. A Compiler Framework for the Reduction of Worst-Case Execution Times. *Real-Time Systems* 46, 2 (2010), 251–300. https://doi.org/10.1007/s11241-010-9101-x

[4] Christian Ferdinand and Reinhard Wilhelm. 1999. Efficient and Precise Cache Behavior Prediction for Real-Time Systems. *Real-Time Systems* 17, 2 (1999), 131–181. https://doi.org/10.1023/A:1008186323068

[5] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. 2010. The Mälardalen WCET Benchmarks – Past, Present and Future. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, Björn Lisper (Ed.). OCG, Brussels, Belgium, 136–146. https://doi.org/10.4230/OASIcs.WCET.2010.136

[6] Damien Hardy, Thomas Piquet, and Isabelle Puaut. 2009. Using Bypass to Tighten WCET Estimates for Multi-Core Processors with Shared Instruction Caches. In *30th IEEE Real-Time Systems Symposium (RTSS)*. 68–77. https://doi.org/10.1109/RTSS.2009.34

[7] Yau-Tsun Steven Li and Sharad Malik. 1997. Performance Analysis of Embedded Software Using Implicit Path Enumeration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 16, 12 (1997), 1477–1487. https://doi.org/10.1109/43.664229

[8] Yun Liang, Huping Ding, Tulika Mitra, Abhik Roychoudhury, Yan Li, and Vivy Suhendra. 2012. Timing Analysis of Concurrent Programs Running on Shared Cache Multi-Cores. *Real-Time Systems* 48, 6 (2012), 638–680. https://doi.org/10.1007/s11241-012-9160-2

[9] Mingsong Lv, Nan Guan, Jan Reineke, Reinhard Wilhelm, and Wang Yi. 2016. A Survey on Static Cache Analysis for Real-Time Systems. *Leibniz Transactions on Embedded Systems (LITES)* 3, 1 (2016), 05:1–05:48. https://doi.org/10.4230/LITES-v003-i001-a005

[10] Claire Maiza, Hamza Rihani, Juan M. Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I. Davis. 2019. A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems. *ACM Computing Surveys (CSUR)* 52, 3 (2019), 1–38.

https://doi.org/10.1145/3323212

[11] Kartik Nagar. 2016. *Precise analysis of Private and Shared Caches for tight WCET Estimates.* Ph. D. Dissertation. Indian Institute of Science Bangalore.

[12] Kartik Nagar and Y. N. Srikant. 2014. Precise Shared Cache Analysis using Optimal Interference Placement. In *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 125–134. https://doi.org/10.1109/RTAS.2014.6925996

[13] Dominic Oehlert. 2021. *Worst Case Execution Time Oriented Code Optimization of Hard Real-Time Multicore Systems.* Ph. D. Dissertation. Technische Universität Hamburg.

[14] Dominic Oehlert, Selma Saidi, and Heiko Falk. 2018. Compiler-Based Extraction of Event Arrival Functions for Real-Time Systems Analysis. In *30th Euromicro Conference on Real-Time Systems (ECRTS)*. 4:1–4:22. https://doi.org/10.4230/LIPIcs.ECRTS.2018.4

[15] The Embedded Microprocessor Benchmark Consortium. 2023. *About the EEMBC AutoBench™ Performance Benchmark Suite.* EEMBC. Retrieved 2023-01-20 from https://www.eembc.org/autobench/

[16] Valentin Touzeau, Claire Maïza, David Monniaux, and Jan Reineke. 2019. Fast and Exact Analysis for LRU Caches. *Proceedings of the ACM on Programming Languages (POPL)* 3 (2019), 54:1–54:29. https://doi.org/10.1145/3290367

[17] Wei Zhang, Mingsong Lv, Wanli Chang, and Lei Ju. 2022. Precise and Scalable Shared Cache Contention Analysis for WCET Estimation. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC)*. 1267–1272. https://doi.org/10.1145/3489517.3530613