Code-Inherent Traffic Shaping for Hard Real-Time Systems

DOMINIC OEHLERT, SELMA SAIDI, and HEIKO FALK, Hamburg University of Technology, Germany

Modern hard real-time systems evolved from isolated single-core architectures to complex multi-core architectures which are often connected in a distributed manner. With the increasing influence of interconnections in hard real-time systems, the access behavior to shared resources of single tasks or cores becomes a crucial factor for the system's overall worst-case timing properties. Traffic shaping is a powerful technique to decrease contention in a network and deliver guarantees on network streams. In this paper we present a novel approach to automatically integrate a traffic shaping behavior into the code of a program for different traffic shaping profiles while being as least invasive as possible. As this approach is solely depending on modifying programs on a code-level, it does not rely on any additional hardware or operating system-based functions.

We show how different traffic shaping profiles can be implemented into programs using a greedy heuristic and an evolutionary algorithm, as well as their influences on the modified programs. It is demonstrated that the presented approaches can be used to decrease worst-case execution times in multi-core systems and lower buffer requirements in distributed systems.

CCS Concepts: • **Computer systems organization** → **Real-time systems**; • **Mathematics of computing** → *Integer programming*; *Bio-inspired optimization*;

Additional Key Words and Phrases: Real-time, multi-core, traffic shaping, event arrival functions

ACM Reference format:
Dominic Oehlert, Selma Saidi, and Heiko Falk. 2019. Code-Inherent Traffic Shaping for Hard Real-Time Systems. *ACM Trans. Embed. Comput. Syst.* 18, 5s, Article 108 (October 2019), 21 pages.
https://doi.org/10.1145/3358215

1 INTRODUCTION AND MOTIVATION

Traffic shaping is a well-known technique to guarantee or improve end-to-end delays and avoid buffer overflow, thereby increasing the Quality of Service in networks [10]. This is done by *shaping* the traffic of network nodes such that they adhere to a defined *traffic profile*. Traffic shapers are components which receive a stream of packets and potentially delay them, such that their outgoing stream contains less bursts and a guaranteed number of packets per time interval. This can decrease network contention and limit the buffer size in network nodes required to temporarily store incoming data. By abstracting traffic behavior to so-called event arrival functions [22], tight guarantees on, e.g., latencies can be estimated.

This article appears as part of the ESWEEK-TECS special issue and was presented at the International Conference on Embedded Software (EMSOFT) 2019.
Authors' addresses: D. Oehlert, S. Saidi, and H. Falk, Hamburg University of Technology, Hamburg, Germany; emails: {dominic.oehlert, selma.saidi, heiko.falk}@tuhh.de.

1539-9087/2019/10-ART108
https://doi.org/10.1145/3358215

Traffic shaping can also be applied in the domain of (hard) real-time systems to improve important characteristics of real-time systems. It has been shown that the techniques of shaping can be applied successfully to, e.g., the area of scheduling [13, 19, 29] or shared bus systems [7, 17, 31, 38]. Yet, traffic shaping can typically not be applied to shared on-chip buses, as this would require the introduction of additional hardware into the system. While this is potentially possible with open source multiprocessor architectures, it is not feasible for the vast majority of commercial off-the-shelf (COTS) products. Regarding hard real-time distributed systems, the introduction of new components into the system is generally possible. At the same time it may not be desirable, as it could increase the costs or energy consumption.

We introduce the notion of *code-inherent* traffic shaping to bypass these obstacles. By transforming the behavior of a given program, its corresponding event arrival function is modified such that its traffic will match a required profile. This way, we import the behavior of a traffic shaper *into* the program, hence the notion of code-inherent traffic shaping. This enables us to leverage the assets of traffic shaping for on-chip buses in COTS multi-core architectures or in a distributed system without additional hardware.

The presented approach transforms the traffic profile of a program by inserting additional machine instructions, therefore it does not rely on any operating system, other kind of existing higher level supervising software components or specific hardware. By delaying the execution of certain instructions, we enforce a given rate control at the compiler level and thus decrease end-to-end delays and buffer requirements.

We demonstrate an evolutionary algorithm-based approach, as well as a greedy heuristic to apply the traffic shaping behavior to the program. As the introduction of additional instructions can potentially harm real-time characteristics of a system, both approaches aim at carefully selecting program paths to shape which will not increase the worst-case timing. Additionally, we show how a program's traffic profile can be efficiently examined whether it meets a required profile or not.

2 RELATED WORK

Most of the work regarding traffic shaping considers comparably complex networks. Shaping is applied there to, e.g., improve latencies or guarantee specific properties of network streams. One of the most commonly used shaping principles is the so-called "leaky bucket" algorithm [35] which was also adopted by the IETF into their *Specification of Guaranteed Quality of Service* [33]. This shaping algorithm allows an adjustable level of burstiness and recovery period between bursts. The algorithm is discussed in a greater detail later in Section 3.2. As our focus lies on applying traffic shaping in the domain of hard real-time systems, we are discussing existing works in this specific domain in the following.

Wandeler et al. analyzed the performance of greedy shapers in real-time systems [38]. A greedy shaper is placed inside each processor of a multi-core architecture and shapes the outgoing communication towards a shared bus. By inserting shapers, the maximum end-to-end delay can be decreased by up to 40%. While these results are very promising, it is left open, how, e.g., a greedy shaper can be integrated into a single processor.

Zhou and Wentzlaff [41] presented a HW-based approach to integrate traffic shapers into single cores of a NoC-based multi-core setup. The additional hardware components enforce a defined traffic profile outgoing from each core into the network. They concluded that they can achieve a significant performance gain using the presented approach compared against static bandwidth allocation. Yet, this approach can not be applied to current multi-core COTS, as they do not feature such HW-based traffic shapers.

Davis and Navet presented an approach to reduce jitter in a controller area network (CAN) by introducing a traffic shaping policy into the gateways of the network [7]. When a CAN message

traverses a gateway, it has to be queued inside the gateway until it is finally transferred from one network to another. This can lead to a significant increase of jitter if queued messages are forwarded in an unfavorable manner. An increased jitter in turn can increase the estimated worst-case response time of messages. In order to decrease this *queuing jitter*, the authors propose a *non-blocking jitter reduction* policy inside the gateways to improve the message queue handling. It is shown that when applied, the traffic shaping can significantly reduce the so-called queuing jitter.

Kumar and Thiele proposed an approach to improve thermal characteristics by shaping the execution of tasks [19]. Whereas the authors' approach is to implement the shaping feature into the scheduler, our approach could be used to directly implement the shaping into the behavior of the actual programs. The basic idea is to selectively pause the execution of a task and put the processor into an idle mode for a certain amount of time to reduce the peak temperature of a chip. This is done by shaping the desired computational time of tasks by either letting them run or briefly halting them. The shaping curves are designed such that all real-time requirements are still guaranteed to be met.

A way to improve battery performance of mobile systems was presented by Chiasserini and Rao [6]. It is shown that the capacity of a battery can be used more efficiently if it is not constantly discharged, but rather discharged in pulses followed by idle periods. As wireless interfaces are one of the major power consumers of mobile systems, the overall idea is to shape the service requests to the wireless interface. The authors conclude that the performance of battery cells can be improved using the presented technique by permitting a trade off against a potential delay of service requests. The authors do not describe how the actual shaping can be implemented. As our presented approach does not rely on any hardware requirements, a mobile application could be shaped in order to increase battery performance.

Rahmani et al. [31] analyzed the effect of traffic shaping inside an Ethernet-based automotive network. The authors analyze different shaping algorithms and their effects on the QoS of the multimedia streams. It is shown that the required buffers in the system as well as the delay of specific packages can be potentially reduced by integrating traffic shapers. While inserting traffic shapers is possible in the automotive domain, our approach can be used to pre-shape the outgoing traffic, hence minimizing the requirements for the following shapers.

Our presented approach could also be used to counter side-channel attacks on a compiler-level similar to the works of Wu et al. [40] or Wang et al. [39]. Wu et al. present an approach to remove instruction- and cache-timing side channels by modifying a given program automatically on a compiler-level. By inserting countermeasures, such as reading all elements of an array instead of only one or pre-loading cache lines, the authors make sure that the execution time of every path is independent from a given set of "secret data". Similarly, Wang et al. [39] presented a way to reduce power side channels by modifying the register allocation and memory locations of potentially leaky variables during compilation. By shaping, e.g., the bus access profile of a program in a more evenly manner, our presented approach could be exploited to reduce timing or power side channels as previously characteristic paths (e.g., a very bursty one) get smoothed.

3 EVENT ARRIVAL FUNCTIONS AND TRAFFIC SHAPERS

In this section, we give a brief overview of event arrival functions, the extraction of these functions on the code-level and an introduction of traffic shapers. Subsequently, we discuss how to verify that an event arrival function adheres to a traffic shaping profile.

3.1 Event Arrival Functions

Event arrival functions can be used to describe complex dynamics of a system. Depending on the definition of an event, multiple characteristics of a task (or more generally of a system) can be

described using event arrival functions like, e.g., its activation pattern or its access behavior to a shared resource.

Definition 3.1. Event. A single event of an event stream is an abstract resource demand request, e.g., a task activation inside a multithreaded system or a bus request inside a multi-core system.

A given cumulative request function $R(t)$ describes the sum of events seen in the time interval $[0, t]$ with $t \in \mathbb{N}_0$. In order to give certain guarantees, the upper event arrival function $\eta^+(\Delta t)$ and lower event arrival function $\eta^-(\Delta t)$ are introduced [22, 37]:

Definition 3.2. Event Arrival Functions. With $R(t)$ describing the number of events seen in the interval $[0, t]$, $t \in \mathbb{N}_0$, the upper event arrival function $\eta^+(\Delta t)$ and lower event arrival function $\eta^-(\Delta t)$ are described by the following inequation:

$$\eta^-(t-s) \leq R(t) - R(s) \leq \eta^+(t-s), \forall t \geq s \geq 0 \tag{1}$$

where $\eta^-(0) = \eta^+(0) = 0$.

The upper event arrival function $\eta^+(\Delta t)$ is a cumulative function describing the *maximum* number of events that can be issued in a time interval of length Δt. The lower event arrival function $\eta^-(\Delta t)$ is the corresponding counterpart and describing the *minimum* number of events that can be seen in a time interval of length Δt. In the domain of hard real-time systems, system-level analysis tools (e.g., Real-Time Calculus [37]) can use the behavior-describing functions to estimate guaranteed worst- and best-case timings for a system. As we are only interested in the upper event arrival functions (describing the maximum accesses to a shared resource), we will refer to the upper event arrival function as simply $\eta(\Delta t)$ in the following.

If the event arrival function of a task is depending on specific actions during the execution of the task (e.g., bus accesses), a safe, yet tight extraction of event arrival functions is complex. An event arrival function can be constructed by measurements and recording traces. Yet similar to measurement-based WCET analysis compared to a formal WCET analysis, the resulting event arrival function is not guaranteed to be safe. Jacobs et al. [15] presented how to safely extract an upper event arrival function from the code-level of a task. This was then formalized and extended to lower event arrival functions by Oehlert et al. [27].

The approach's basic idea of extracting a task's event arrival function lies in the description of all possible paths through the program using an integer linear program (ILP). The so-called *implicit path enumeration technique* by Li and Malik [24] is extended to support a path starting and ending at arbitrary basic blocks. The ILP can be solved for one particular interval length Δt and returns the maximum (or minimum) number of events that can be issued in this interval. Regarding an upper event arrival function, this is done by setting the objective to maximize the number of events a_{Total} triggered along an arbitrarily taken control-flow path, whereas the total required amount of time w_{Total} is not allowed to exceed a given Δt. This ILP model will be used in the upcoming sections in order to determine the event arrival function of a given task.

The calculation of a single value of $\eta(\Delta t)$ for a given Δt can be seen as a selective traveling salesman problem (STSP) [21]: The graph is represented by the control flow graph of the program on a basic block level with a virtual starting point connected to every block. An edge's cost represents the execution time of the corresponding basic block, whereas the profit of a node represents the number of events of the corresponding basic block. The aim is to maximize the profit (number of events) while the costs (execution time) are not allowed to exceed the given Δt. As the STSP is NP-hard it can be concluded that the calculation of a single value of $\eta(\Delta t)$ on a basic block level is NP-hard as well.

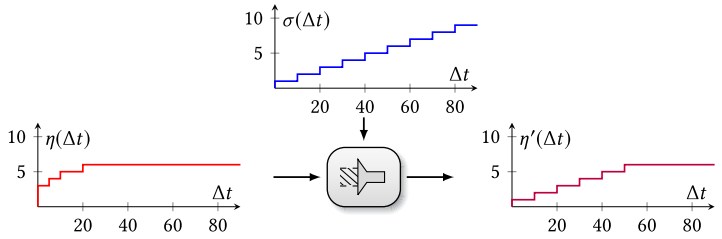


Fig. 1. Greedy traffic shaper.

3.2 Traffic Shaper

Given an input stream with an upper event arrival function of $\eta(\Delta t)$, a so-called *shaper* processes this input stream and generates an output stream which is bound by a profile function $\sigma(\Delta t)$. A so-called *greedy shaper* buffers the input stream in case it would violate the profile function $\sigma(\Delta t)$, but emits it as soon as possible [22]. Figure 1 depicts an example of a greedy shaper. The curve $\eta(\Delta t)$ is an incoming event arrival curve which is not conform to any profile. The profile function $\sigma(\Delta t)$ describes the upper curve allowed to appear at the output of the traffic shaper. In this particular example, $\sigma(\Delta t)$ limits the output event stream to contain at maximum one event every 10 time units. $\eta'(\Delta t)$ depicts the outgoing event arrival curve of the shaper which is conform to the given profile function $\sigma(\Delta t)$.

One commonly known example of a greedy shaper is the so-called *leaky bucket* algorithm [35]. We use this shaping algorithm as an example as it is still commonly used in many areas [18, 30, 36] and adopted into the *Specification of Guaranteed Quality of Service* [33] by the IETF. A so-called token is added to a collecting bucket with a definable rate of r. Such a token represents the right to send a specific amount of data. As the bucket only has a fixed capacity of b, it can only hold b tokens at maximum. Arriving data is queued in a FIFO queue. If there are enough tokens inside the bucket to transmit the data (every transmission "costs" a certain amount of tokens), the data is directly passed on and the tokens are removed from the bucket. In the other case, the data is stalled until the bucket is filled up with the required amount of tokens. Therefore, the maximum burstiness of the output stream is defined by the token bucket size b, whereas r can be seen as defining the rate of the outgoing stream.

3.3 Profile Adherence Checking

In order to implement a traffic shaping behavior, a crucial requirement is to validate whether a given program adheres to a desired profile function or not. In the following, we will present methods to validate this requirement for an arbitrary traffic shaping profile or more efficiently for a selected number of specific shapers. If not noted otherwise, we assume a discrete time model with $t \in \mathbb{N}_0$.

3.3.1 Arbitrary Shaping Profiles. As discussed in Section 3.2, traffic shaping profiles can be represented by a corresponding profile function $\sigma(\Delta t)$. If it can be validated that the event arrival function $\eta(\Delta t)$ of a given program is below or equal to $\sigma(\Delta t)$ for any Δt, it is guaranteed that the program's behavior adheres to the desired profile. As $\eta(\Delta t)$ and $\sigma(\Delta t)$ are both monotonically increasing and piecewise constant, it is sufficient to check at every discontinuity of $\eta(\Delta t)$. We combine all values of Δt for which $\sigma(\Delta t)$ has to be tested to a set $\mathcal{D}$. The required test described in Algorithm 1 is similar to the so-called processor demand test proposed by Baruah [2]. It returns true in case the event arrival function $\eta(\Delta t)$ adheres to the desired shaping profile and otherwise false. As $\sigma(\Delta t)$ is known by definition and $\eta(\Delta t)$ can be determined using the described ILP model,

ALGORITHM 1: Basic test for arbitrary profiles.

```
for Δt in 𝒟 do
    if η(Δt) > σ(Δt) then
        return false
    end if
end for
return true
```

all inputs are present. Yet, the magnitude of number of points to be evaluated can quickly render this approach practically infeasible. In case of a periodic system, $\mathcal{D}$ would contain all points of discontinuity up to the hyperperiod of the system.

Alternatively, the profile function $\sigma(\Delta t)$ can be described using linear constraints and directly be integrated into the ILP used to model the event arrival function as described before. Instead of maximizing the number of events in a given time interval length Δt, the maximum difference between the number of events a_{Total} and $\sigma(\Delta t)$ for any arbitrary Δt is searched:

$$\max : a_{\text{Total}} - \sigma(w_{\text{Total}}) \tag{2}$$

The ILP variable w_{Total} describes the accumulated time over the chosen control-flow path. This is simply done by replacing the ILP objective with the given term and removing any restrictions on the time interval Δt. The description of $\sigma(w_{\text{Total}})$ inside the ILP model is depending on the required profile function. The complete ILP to model an event arrival function [27] is not discussed here as it would it exceed the scope of this paper and does not offer any novelty. In case the ILP solver returns a negative value or zero for the objective, it is ensured that the event arrival function $\eta(\Delta t)$ is lower than or equal to $\sigma(\Delta t)$ for all possible Δt. Otherwise, the objective value represents the maximum violation of $\eta(\Delta t)$ in regard to the desired profile.

3.3.2 Periodic Step Function Profiles. The effort for a profile adherence check can be drastically reduced for shapers which have a simple periodic step function as a profile function in the form of $\sigma(\Delta t) = Y \cdot \lceil \frac{\Delta t}{P} \rceil$, where Y is the height of a step and P the period.

Proposition 3.3. *If $\eta(P) \leq Y$ holds, $\eta(\Delta t) \leq \sigma(\Delta t)$ holds for any $\Delta t \in \mathbb{N}_0$, given a profile function in the form of $\sigma(\Delta t) = Y \cdot \lceil \frac{\Delta t}{P} \rceil$.*

Proof. The value of $\eta(\Delta t)$ at every multiple period value is lower or equal to the profile function $\sigma(\Delta t)$:

$$n \in \mathbb{N}_0 \tag{3}$$

$$\eta(n \cdot P) \leq n \cdot \eta(P) \leq n \cdot Y \tag{4}$$

$$\sigma(n \cdot P) = Y \cdot \left\lceil \frac{n \cdot P}{P} \right\rceil = n \cdot Y \tag{5}$$

$$\Rightarrow \eta(n \cdot P) \leq \sigma(n \cdot P) \tag{6}$$

Equation (4) is exploiting the subadditivity of $\eta(\Delta t)$ and the required precondition given in Proposition 3.3. As $\sigma(\Delta t)$ only increases at points $\Delta t = n \cdot P + 1$ ($n \in \mathbb{N}_0$) and given the fact that $\eta(\Delta t)$ is monotonically increasing, the maximum deviation between the two curves can be found at points $\Delta t = n \cdot P$. As shown in Equation (6), the event arrival function is always below or equal to the shaper function for these points if the precondition is met. Hence, it is guaranteed that $\eta(\Delta t)$ is lower or equal to $\sigma(\Delta t)$ for any given Δt. □

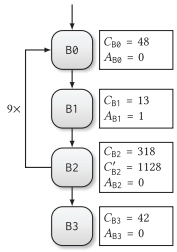


Fig. 2. Exemplary CFG I.

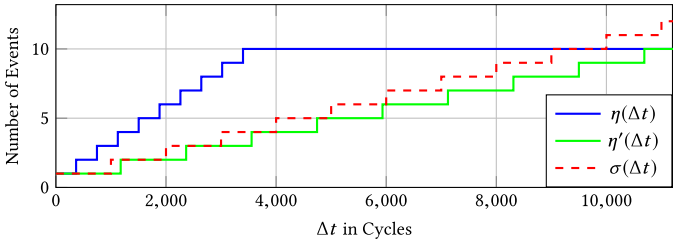


Fig. 3. Event arrival curves for the exemplary CFG I, with modified timings and the desired maximum access profile.

This reduces the profile adherence check to a single calculation of $\eta(\Delta t)$ at point $\Delta t = P$. In case the precondition of Proposition 3.3 is not met, the difference $\eta(P) - Y$ can be used as a figure of merit of how much the shaping profile is violated.

4 CODE-INHERENT TRAFFIC SHAPING

In this section, we introduce the idea of *code-inherent* traffic shaping. In contrast to traditional hardware-based traffic shapers in a system, we present a novel approach to achieve a desired temporal access profile by modifying the program itself. We assume an architecture with private memories for code, such that the insertion of additional delay code does not increase the number of accesses.

4.1 Basic Principle

Accesses to a shared medium are triggered by specific actions inside a program. After specifying the event triggering instructions, the approach shown in [15] and [27] is able to generate an upper event arrival curve. The specification which instructions of a program may trigger a shared memory access can be done using, e.g., instruction classification and value analysis, as accesses to a shared medium are typically done by accessing a certain memory area. This can either be an actual shared memory or, e.g., memory mapped registers for sending messages.

By extracting the upper event arrival function, it can be examined whether the program adheres to a desired access profile or not. To illustrate this, an exemplary control-flow graph is given in Figure 2. The blocks depict sequences of instructions, whereas the edges indicate the control-flow. Beside each block B, its execution time C_B and the maximum number of events it may trigger A_B is shown. The loop has a given upper bound of 10 iterations (note that the loop is tail-controlled, hence for 10 iterations, the back-edge can only be taken 9 times). The corresponding event arrival function $\eta(\Delta t)$ is depicted in Figure 3. As expected, the event arrival function shows a periodic increase and converges with a total number of 10 events.

Figure 3 additionally depicts an arbitrarily chosen desired profile function $\sigma(\Delta t)$. In this specific example, it is desired to have at most 1 event triggered every 1000 cycles. As it can be clearly seen, the actual event arrival function $\eta(\Delta t)$ does not meet this desired characteristic as it surpasses $\sigma(\Delta t)$ already at $\Delta t \approx 370$.

In order to shape our event arrival function such that it will meet its desired profile function $\sigma(\Delta t)$, we can modify our program. As an example, additional instructions are added into the block B2, such that its execution time is increased to $C'_{B2} = 1128$ cycles. The basic block B2 is chosen arbitrarily here to illustrate the general principle. Delaying instructions could also be added to the blocks B0 or B1. The updated event arrival function $\eta'(\Delta t)$ is depicted in Figure 2 as well. It can be seen that due to the increased timing of block B2, the program now provably adheres to the

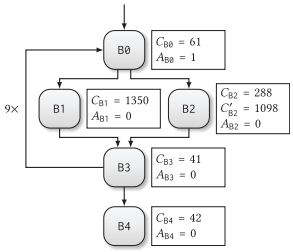


Fig. 4. Exemplary CFG II.

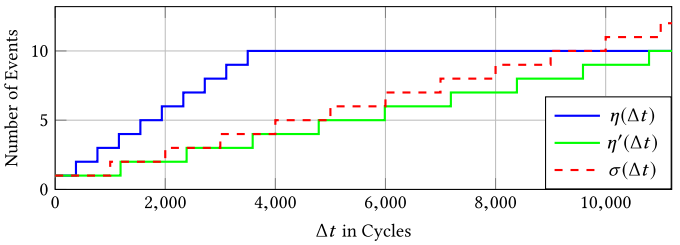


Fig. 5. Event arrival curves for the exemplary CFG II, with modified timings and the desired maximum access profile.

given profile at any time. This principle can be seen as a *code-inherent* traffic shaping, as the access behavior of the modified program is now bounded by the profile function $\sigma(\Delta t)$.

While the modified event arrival curve $\eta'(\Delta t)$ conforms with the profile function $\sigma(\Delta t)$, the shaping is not *greedy*, as this would require that $\eta'(\Delta t)$ is equal to $\sigma(\Delta t)$ up to $\Delta t = 10\,000$ cycles. This is due to two reasons: 1) The granularity of adding a delay using instructions is limited by several factors of an architecture like, e.g., memory latency or fetching width. Therefore, an event may be delayed longer than minimally required. 2) Depending on the structure of the program, the insertion of additional delays may inevitably shift the whole event arrival curve and not only a local point. Therefore, it may be impossible (without, e.g., the removal or rescheduling of code) to achieve a greedy shaping behavior when specific events need to be delayed.

In case a new or different profile function $\sigma(\Delta t)$ is needed due to changed traffic shaping requirements later, the code-inherent shaping has to be re-run with the original, unmodified program as an input in order to receive an updated program which adheres to the given profile function.

Inserting additional delaying instructions into the program that do not seem to be required, might rise certification concerns for hard real-time environments. A widely used collection of development guidelines for critical systems are the MISRA C guidelines [1]. According to the guidelines concerning dead code, dead code is defined as "*any operation that is executed but whose removal would not affect program behavior*", yet clarifying that "*the behavior of an embedded system is often determined not just by the nature of its actions, but also by the time at which they occur*". While the additional delay instructions are not changing the system state by their computational results, but change the system behavior due the point in time they are executed, they hence are no dead code and thus comply with the MISRA C rules. Generally speaking, any insertion of code into a safety-critical system by a compiler must comply with a corresponding functional requirement. And if the adherence to some timing profile and schedulability of tasks are part of the requirements, code-inherent traffic shaping is feasible for critial systems.

4.2 WCET-aware Code-inherent Traffic Shaping

The approach shown in Section 4.1 is able to shape the event arrival function, yet obviously increases the worst-case execution time in this case. Whereas the WCET was originally 3832 cycles, the modification increases the WCET to 11 932 cycles. To avoid this, we can exploit the structure of a program while still maintain a traffic shaped behavior.

Figure 4 shows another exemplary control-flow graph. Figure 5 shows the corresponding event arrival curve $\eta(\Delta t)$ and desired profile function $\sigma(\Delta t)$. Yet again, the original program behavior does not comply to $\sigma(\Delta t)$. In contrast to the control-flow graph shown in Figure 2, the graph in Figure 4 has a conditional branch inside the loop. In this case, additional code is introduced to block B2 and increases its timing C'_{B2} to 1098 cycles. This leads to the updated event arrival function $\eta'(\Delta t)$ shown in Figure 5 which satisfies the required profile $\sigma(\Delta t)$. However,

the worst-case execution time remains constant at 14 562 cycles also with the additional code. This is due to the fact that the worst-case execution path (WCEP) stays untouched as only the best-case execution time is increased. As the upper event arrival function describes the *maximum* number of events in a certain time interval, it is defined by the *minimum* time between two or more events. This provides the opportunity to adjust the program's behavior by only modifying parts which are not part of the WCEP (and do not cause a switch of it).

5 INTEGRATION OF WCET-AWARE TRAFFIC SHAPING BEHAVIOR

In the upcoming section, we present two algorithms to solve the following problem: *Given a program* P *and profile function* $\sigma(\Delta t)$*, determine the locations (i.e., which basic blocks) and amount of delaying instructions to be inserted into the program* P*, such that the resulting event arrival function* $\eta(\Delta t)$ *will be less than or equal to* $\sigma(\Delta t)$ *for all* Δt *while increasing the WCET as little as possible.*

As the calculation of a single value $\eta(\Delta t)$ for a given Δt on a basic block level is an NP-hard problem (cf. Section 3.1), the optimization problem to be solved is NP-hard as well as it requires to calculate the value of $\eta(\Delta t)$ as a prerequisite.

5.1 Greedy Heuristic

The general idea of this heuristic is to analyze which path of a program is conflicting with the required profile and try to insert additional code on parts of this path which is not overlapping with the current WCEP. Algorithm 2 shows the complete heuristic.

The algorithm's outer loop starts with running an analysis whether the current program is violating the required shape or not in line 2. This returns the number of exceeding events V and a set $\mathcal{P}_\eta$ which includes all basic blocks of a path which violates the profile. An arbitrary violating path is chosen in case there are multiple paths violating $\sigma(\Delta t)$. If the program's behavior is conform with the required profile function, $V = 0$ and the set $\mathcal{P}_\eta$ is empty. The actual implementation of this analysis is discussed in Section 3.3. In case the shape requirement is not violated, the algorithm finishes in line 4. Otherwise, the current WCET C and the corresponding set of basic blocks $\mathcal{P}_W$ which are part of the WCEP are analyzed. In line 7, the difference between the set $\mathcal{P}_\eta$ and $\mathcal{P}_W$ is calculated, meaning all basic blocks which are part of the shape violating path, yet not of the WCEP. If this does not result in an empty set, any arbitrary basic block B from this set is chosen and the indicator variable Z is set to false. Z is used to indicate whether B was part of the WCEP or not. Otherwise, the basic block B of the shape violating path with the lowest worst-case execution count (WCEC) is chosen and Z is set to true. In case there are multiple basic blocks of the set $\mathcal{P}_\eta$ sharing the minimum WCEC, any of these blocks is chosen arbitrarily.

The subsequent do-while-loop between lines 15 and 23 is then stepwise increasing the number of additional delay code in the chosen basic block B. $\mathbf{N}[\mathrm{B}]$ describes the additional delay which is added to the basic block B. Initially, $\mathbf{N}$ is zero for all basic blocks. The previously added delay is stored to the variable L. If no delay was previously added to B, a minimum amount of delay μ is inserted. Otherwise, the currently inserted delay is doubled. The minimum delay value μ is determined by the minimal delay which can be achieved by inserting additional code, e.g., the number of cycles required for a single NOP operation. With the applied additional delay to the basic block B, the WCET and shape violation are re-calculated. In case we previously chose a basic block which was not part of the WCEP (Z was set to false) but the WCET increased ($C' > C$), the algorithm breaks the loop already here. This means that the introduced delay to basic block B caused a switch of the WCEP. Otherwise, it is checked whether there is still a shape violation or the chosen basic block B is still on the shape violating path. If not, the loop is exited.

At line 24, a delay of $\mathbf{N}[\mathrm{B}]$ is introduced to B which led to either no shape violation at all anymore, the exclusion of B from the shape violating path or a WCEP switch. In either case, the current

ALGORITHM 2: Greedy heuristic for WCET-aware traffic shaper integration.

```
1:  while true do
2:      (V,P_η) ← shapeViolation()
3:      if V = 0 then
4:          break
5:      end if
6:      (C,P_W) ← calcWCET()
7:      P'_η ← P_η \ P_W
8:      if P'_η ≠ ∅ then
9:          B ∈ P'_η
10:         Z ← false
11:     else
12:         B ∈ {x | x ∈ P_η, WCEC(x) ≤ min_{∀i∈P_η} (WCEC(i))}
13:         Z ← true
14:     end if
15:     do
16:         L ← N[B]
17:         N[B] ← (N[B] = 0 ? μ : N[B] · 2)
18:         (C', P_W) ← calcWCET()
19:         (V,P_η) ← shapeViolation()
20:         if ( Z = false ) and (C' > C) then
21:             break
22:         end if
23:     while (V > 0) and ({B} ∩ P_η ≠ ∅)
24:     U ← N[B]
25:     N[B] ← (L + U)/2
26:     while ((U − L)/L > ε) and (N[B] ≥ μ) do
27:         (C', P_W) ← calcWCET()
28:         (V, P_η) ← shapeViolation()
29:         if (V = 0) or ({B} ∩ P_η = ∅) or (Z̄ and (C' > C)) then
30:             U ← N[B]
31:         else
32:             L ← N[B]
33:         end if
34:         N[B] ← (L + U)/2
35:     end while
36:     N[B] ← U
37: end while
```

value of N[B] is easily larger than required to lead to this change, as it was found by simply keep multiplying the old value by 2. Therefore, we try to find a smaller value for N[B] inside the range (L, U). This is done in lines 26 to 35 using a binary search. The binary search is carried out as long as the difference between the upper and lower boundary (normalized on the lower boundary) is greater than a user-defined threshold ϵ and the added delay is greater than the absolute minimum delay μ possible. The threshold ϵ is introduced to speed up the algorithm while trading a potentially worse WCET against a lower runtime.

During the binary search, the worst-case timing behavior, as well as the shape violation are recalculated at each iteration. In case the updated value of N[B] causes the selected basic block B not to be part of the shape violating path anymore, a shape violation of zero (or an increase of

the WCET if B was not part of the WCEP previously), we select $\mathbf{N}[B]$ as the current upper value U. Otherwise, it is used as the current lower value L. Subsequently, $\mathbf{N}[B]$ is newly set between the upper and lower bound. After the termination criterion of the binary search was met, the last known upper bound U is used as the amount of delay to be added to the basic block B.

5.2 Evolutionary Algorithm-based Shaping

The following section proposes a genetic algorithm to implement a traffic shaping behavior into a program while trying to keep the increase in WCET as low as possible. The algorithm is bi-criteria, focusing on WCET and shape violation. We will discuss the genome composition of a single individual, the fitness function, the mutation, the crossover and the initial population.

Genome Composition. The genome of a single individual is represented as a list of unsigned integers. Each integer represents the number of delaying cycles to be inserted in a corresponding basic block. Therefore, each individual's genome consists of M_B integers, where M_B corresponds to the total number of basic blocks.

Initial Population. We place one individual with a completely unmodified genome and a heavily modified one in the initial population. In *each* basic block of the heavily modified individual, a constant amount of delay is inserted. This delay inserted in each basic block is chosen sufficiently large to ensure at least one individual which is not violating the profile function $\sigma(\Delta t)$. The rest of the initial population is created completely random for one half and in a directed manner for the other. For the complete random half, for each basic block the delay is randomly drawn (uniform distribution) from the range $(0, D)$, where D is a user-defined constant describing the maximum amount of delay initially added to a basic block. An initial individual of the directed half is also assigned randomly drawn delays from the range $(0, D)$, yet only for basic blocks which are part of the current shape violating path according to an initial analysis. All other basic blocks do not receive an additional delay. The directed half is generated to increase the chances of a good starting point of the algorithm, whereas the complete random half increases the diversity during recombination, especially in the early generations.

Fitness Function. The fitness function evaluates the two criteria most relevant in this case: shape violation and WCET. For each individual, a shape violation analysis is carried out which returns a figure of merit how much the current system is violating the profile function $\sigma(\Delta t)$. Besides, a WCET analysis is done as well. Both results are cached to avoid re-calculation of identical genomes. The implementation of the shape violation analysis is discussed in Section 3.3. The fitness of an individual is determined using the multiobjective fitness assignment of the SPEA2 algorithm [42], whereas shape violation and WCET are the competing objectives.

Mutation. The applied mutation is not completely random, yet guided to speed up the convergence of the algorithm. In case an individual is still violating the shape requirements, mutation is only allowed to *add* delay to basic blocks which are on the shape violating path according to the individual's last analysis. If the individual does adhere to the required profile, the mutation is only allowed to *remove* a certain amount of delay of any basic block.

During the start of the mutation phase of a single individual, the number of mutations to be performed is drawn randomly (uniform distribution) in the range of $(0, M_E - 1)$, where M_E is the total number of basic blocks eligible for mutation. This means if the individual is violating the profile function $\sigma(\Delta t)$, M_E equals the number of basic blocks on the violating path. Otherwise, M_E equals the number of total basic blocks, hence $M_E = M_B$. Then, every qualified basic block is mutated with a user-defined probability until the determined number of mutations to be performed is reached. If a basic block is mutated, it is checked whether there is already an additional delay

incorporated or not. In case a delay was already added to the basic block, the delay is increased by a factor randomly drawn (uniform distribution) from the range (1.0,2.0) if the individual is violating the profile function. If the individual is adhering to the profile function, the delay is decreased by a factor randomly drawn (uniform distribution) from the range (0.0,1.0)

In case the block to be mutated does not contain any added delay and the individual is violating the profile function, a randomly drawn delay from the range $(0, D)$ is added to the block. If the individual adheres to the profile function and the basic block does not contain any additional delay, the block is not altered by the mutation.

Crossover. A single point crossover with two individuals is used. The crossover point is determined by drawing a random integer from the range $(0, M_B - 1)$. The new individual is then simply created by adapting the genome of the first individual up to the crossover point and from the second individual for the rest.

6 EVALUATION

In the following section, the shaper integrations presented in Section 5 are evaluated for different shapers and varying settings. All experiments were performed on an Intel Xeon Server (20 cores at 2.3 GHz with 94 GB RAM) and ILPs were solved using Gurobi 8.1.0, whereas each ILP solving process was limited to a single thread. The PISA framework [3] was used for the evolutionary algorithm-based approach. All benchmarks were compiled with several ACET-oriented optimizations activated (`-O2` flag). Timing analyses were done using the internal WCET analyzer provided by the WCET-aware C compiler (WCC) [9]. Benchmarks with irreducible loops were excluded from the evaluation as they are currently not supported by the timing analyzer.

In Section 6.1, an overview of the implemented shapers is given which are used in the following. Here, we restrict the traffic shaping behaviors to (mostly) periodic ones as these are commonly used [33]. Section 6.2 shows how the implemented shapers transform the event arrival curve for one particular example in detail. In Section 6.3, it is shown for one exemplary system in detail, how the presented techniques can be used to improve worst-case timings in a multi-core hard real-time system. Finally, Section 6.4 covers a more thorough evaluation of the introduced shapers and implementation approaches for another exemplary application, namely the reduction of buffer requirements.

6.1 Implemented Shapers

Different shapers were implemented for evaluation purposes and presented in the following. The introduced technique of code-inherent traffic shaping and the implementation approaches are not restricted to presented shapers, but can be applied to any shaping algorithm which can be described by a corresponding profile function $\sigma(\Delta t)$.

Lazy Leaky Bucket. The *lazy leaky bucket* (LLB) shaper has the following shaping curve:

$$\sigma(\Delta t) = \begin{cases} 0 & \text{if } \Delta t \leq 0, \\ b + \lceil r \cdot \Delta t \rceil & \text{else.} \end{cases} \tag{7}$$

The shaping curve corresponds to the event arrival function of a regular leaky bucket shaper with a bucket size of b and rate of r. It is prefixed with *lazy* as a regular leaky bucket shaper is greedy, yet as explained in Section 4.1 the code-inherent traffic shaping may be non-greedy. The profile adherence test is implemented by describing Equation (7) inside the ILP formulation used to model the event arrival function.

Lazy Leaky Bucket Rate. The *lazy leaky bucket rate* (LLBR) shaper is a simple specialization of the LLB shaper with $b = 0$. Its curve is described using the following equation:

$$\sigma(\Delta t) = \begin{cases} 0 & \text{if } \Delta t \leq 0, \\ \lceil r \cdot \Delta t \rceil & \text{else.} \end{cases} \tag{8}$$

The profile adherence test is implemented using the simplified test described in Section 3.3.2.

Full Refill. The *full refill* (FR) shaper is similar in its behavior to a leaky bucket shaper, yet more simplified. Its shaping curve $\sigma(\Delta t)$ is described by the following equation:

$$\sigma(\Delta t) = \begin{cases} 0 & \text{if } \Delta t \leq 0, \\ Y \cdot \lceil r \cdot \Delta t \rceil & \text{else.} \end{cases} \tag{9}$$

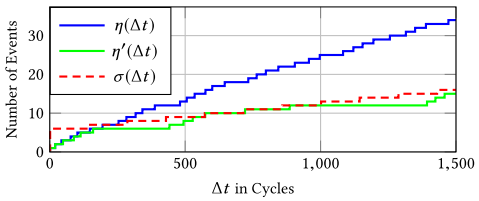
Whereas the token bucket of a leaky bucket shaper gets refilled token by token every few time units, it is simply completely refilled to its maximum Y for the *full refill* shaper. Since it fulfills the requirements of a simplified profile adherence check described in Section 3.3.2, it is implemented as such.

6.2 Case Study

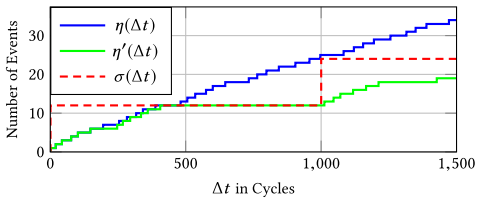
In the following, we present the effects of three different styles of shapers exemplary in a greater detail on the `select` benchmark from the MRTC benchmark suite [11] with annotated loop bounds from TACLeBench [8]. This benchmark was chosen randomly with no deeper reason. Note that this case study should only illustrate the effects of the different shapers on a given example program and its event arrival function. A more thorough evaluation of more benchmarks and precise criteria follows in Section 6.4. In this example, we choose a single-core ARM7TDMI processor as the evaluation architecture to showcase the modifications made to a single event arrival function. We assume that each access to the `.data` section will cause an event.

Figures 6(a) to 6(c) depict the original event arrival function $\eta(\Delta t)$, the corresponding shaping curve $\sigma(\Delta t)$ and the shaped event arrival function $\eta'(\Delta t)$ for different shapers. The shaping was applied using the greedy heuristic shown in Section 5.1. All shaping curves were set to ensure 50% reduction of the maximum number of events occurring in an interval of $\Delta t = 1000$ cycles. The parameters of the lazy leaky bucket shaper depicted in Figure 6(a) were chosen to be in between the *full refill* shaper (Figure 6(b), *FR*) and the lazy leaky bucket rate shaper (Figure 6(c), *LLBR*). It can be seen that all shaped event arrival functions fulfill the required profile as they are lower than or equal to the shaping curve for any value of Δt.

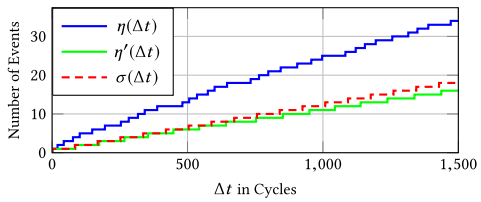
Figure 6(d) shows the effects when applying the different shapers. All values are normalized to the corresponding *LLB* shaper values. It can be seen that for this particular example, the application of the *LLB* shaper increases the WCET significantly more, whereas the *LLBR* shaper increases the WCET the least. This is also reflected by the amount of delay inserted into the program, yet with greater relative differences. It is shown that the application of the *FR* shaper adds 71% less delay than the *LLB* shaper, while the *LLBR* shaper adds 79% less. The significant increase in WCET when applying the *LLB* shaper in comparison to the other shapers most likely stems from the lower gradient (number of events per time unit). While the *LLBR* and *FR* shapers have a gradient of ≈ 0.012 events per time unit, the *LLB* shaper only has a gradient of ≈ 0.007. The lower WCET increase when comparing the *LLBR* shaper to *FR* originates from the relative ϵ threshold (set to 0.2 here) of the greedy heuristic. As the heuristic tests the event arrival function at different values for Δt (1 000 for *FR* and 84 for *LLBR*), the absolute tightness of the upper and lower bounds to be found differ. This is also reflected in the relative runtime required for applying the different shapers. Whereas the *LLB* and *FR* shapers require a similar runtime, the *LLBR* shaper needs ≈ 254 times longer than the *LLB* shaper to be applied using the greedy heuristic.



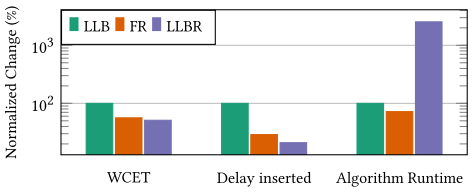
(a) Lazy Leaky Bucket ($b = 5$, $r = 1/143$), *LLB*.



(b) Full Refill ($Y = 12$, $r = 1/1000$), *FR*.



(c) Lazy Leaky Bucket Rate ($r = 1/84$), *LLBR*.



(d) Comparison of shaper-induced characteristics.

Fig. 6. Shapers applied to the `select` benchmark to reduce the max. number of events for $\Delta t = 1000$ cycles by 50% using the greedy heuristic.

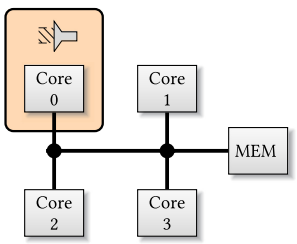


Fig. 7. Exemplary multi-core architecture.

6.3 Application Example

One possible application of the presented code-inherent traffic shaping is the improvement of worst-case timings inside a multi-core architecture. As previously mentioned, actual hardware-based traffic shapers can typically not be integrated into a COTS on-chip bus system. Figure 7 depicts an exemplary multi-core architecture with 4 cores and a shared memory. A single core consists of an ARM7TDMI architecture. Each core also has a private memory which is not shown here due to space reasons. We assume that all instructions are placed inside the private memory and all data objects reside in the shared one. The bus is assumed to follow a fixed priority non-preemptive arbitration policy, whereas core 0 has the highest priority and core 3 the lowest one. We consider all data accesses to be blocking, i.e., if an instruction reads or writes to the shared memory, the pipeline will be halted until the access is finished.

Table 1 shows the tasks mapped to the cores, their periods and their timing information. The benchmarks were taken from the MRTC benchmark suite [11]. $\text{WCET}_{\text{Orig}}$ denotes the maximum execution time (in cycles) of the benchmarks for the unmodified system. The periods were chosen such that the load per core is close to 0.3 when executed in isolation without interference of the other cores. This load was selected to have an equal load in the isolated case for all cores, yet still a load below 1.0 for all cores with interference. The actual WCET C_i of a task i (also considering

Table 1. Tasks and Their Corresponding Timing Information (All Timings are Given in Cycles)

Core	Benchmark	Period	$\text{WCET}_{\text{Orig}}$	WCET_{Mod}	Change (%)
0	`bsort100`	17 781 120	6 217 369	6 511 399	4.7
1	`matmult`	11 664 000	5 410 138	5 234 783	−3.2
2	`crc`	3 402 000	2 240 660	2 137 600	−4.6
3	`cnt`	252 000	208 084	196 824	−5.4

the interference from the other cores) is calculated using the following formula:

$$C_i = C_i^{\text{iso}} + \underbrace{\sum_{j \in \mathcal{H}_i} \eta_j(C_i) \cdot L}_{X} + \underbrace{\min\left(\sum_{j \in \mathcal{L}_i} \eta_j(C_i), \eta_i\left(C_i^{\text{iso}}\right)\right) \cdot (L-1)}_{Y} \tag{10}$$

Whereas C_i^{iso} is the WCET of task i executed in isolation, $\mathcal{H}_i$ is the set of all tasks executed on cores with a higher bus priority than i and $\mathcal{L}_i$ is the counterpart for all tasks on cores with a lower priority. L is the memory access latency for the shared memory. The formula is very similar to the analysis of worst-case response times for fixed priority tasks initially given by Joseph and Pandya [16] and later refined and extended by, e.g., Lehoczky [23] and more. In the worst case, the task i is blocked for X cycles by higher priority cores when trying to access the bus. Additionally, a single access may be blocked for up to $L-1$ cycles from a lower priority core if it received the bus grant before task i requested it. As the number of accesses initiated by task i in a single program run is limited by $\eta_i(C_i^{\text{iso}})$, not more accesses can be blocked by lower priority cores. If the sum of all accumulated accesses from lower priority cores is below $\eta_i(C_i^{\text{iso}})$, the number of blocked accesses for task i can not be greater than this, hence the min-term. Equation (10) is solved iteratively until C_i converges. As an initial value of C_i, the isolated WCET C_i^{iso} is used. We assume an architecture which is free of timing anomalies [32] (as typical system-level analyses do as well), as a purely compositional view is otherwise not safe [15].

In order to improve the timing behavior of the lower priority cores 1, 2 and 3, a traffic shaper is applied to the highest priority core 0. In this example, a *full refill* strategy is applied with a refill period of 2000 cycles and 28 tokens. This means that each data access costs 1 token for core 0, while its token budget is only stocked up to at maximum 28 tokens every 2000 cycles. Without shaping, the higher priority core 0 initiated up to 30 accesses in 2000 cycles.

We applied the heuristic ($\epsilon = 0.2$) described in Section 5.1 to the given system in order to enforce the traffic shaping behavior into the code. In this particular example, the algorithm inserted only 5 NOPs in total in order to achieve a shape conformity. Due to this additional code, the WCET of the benchmark `bsort1000` increased by ≈4.7%. Yet, the benchmarks on the lower priority cores 1, 2 and 3 can leverage the reduced bus interference. The `matmult` benchmark's WCET decreases by 3.2%, `crc` shows a WCET reduction of 4.6%, whereas `cnt` has a reduction of 5.4%.

In general, we can conclude that the proposed integration of traffic shaping behavior into the program can potentially increase the performance of a system. In this particular example, the access profile of a high bus priority core was shaped to decrease the worst-case timings of the lower priority cores.

6.4 Buffer Reduction

Another possible application of applying a traffic shaping behavior to a program is to ease the requirements of buffers inside a system, to prevent a buffer overflow or to be able to give certain guarantees on the outgoing traffic.

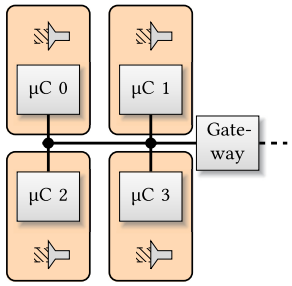


Fig. 8. Exemplary distributed system.

Evaluation Setup. We assume the architecture shown in Figure 8. The architecture consists of four microcontrollers connected over a field bus with a static priority bus arbitration. The network is connected to another network over a gateway. Such architectures are commonly found in, e.g., the automotive domain [26] where the CAN bus [14] (a field bus with static priority arbitration) is connecting several MCUs and is connected to other networks (either CAN as well or other automotive bus types like FlexRay or LIN) via gateways. The gateway transfers messages from the depicted network over to a second network and vice versa. As the second network may be operating at a different speed or is simply blocked for an amount of time by other bus devices, the gateway potentially needs to buffer a certain amount of messages before it can transfer them.

For an exemplary evaluation setup, we assume that the buffer size inside the gateway is only large enough to guarantee a flawless operation for up to F messages in a time interval of 1000 cycles, otherwise a buffer overflow may be possible. The value of F will be chosen individually per system in this evaluation. In order to evaluate the different shapers, implementation strategies and their influences on the worst-case timings, we assume that the four microcontrollers in combination exceed this threshold by a certain degree and need to be shaped. For this, benchmarks from different suites (MRTC [11], UTDSP [4], DSPStone [43], MediaBench [5], MiBench [12], NetBench [25], PolyBench [20] and StreamIT [34]) were sorted by their average-case execution time and bundled to sets of 4, allocating one program per core. A total of 112 different benchmark combinations were evaluated for each setting. We assume ARM7TDMI-based microcontrollers with no caches. Periods were set to achieve a load of 0.5 per core with implicit deadlines. Higher loads caused the LLB shaper in combination with the greedy heuristic to generate only unschedulable systems. For the sake of evaluation, we assume that each access to the `.data` section will cause a message.

The greedy heuristic's binary search margin ϵ was set to 0.2. The evolutionary algorithm was set to 30 generations with a population size of 20. A timeout was set to 2 h for both implementation strategies.

Figure 9 depicts the relative repair rate of the different shaping styles and implementation techniques plotted against the relative reduction of events. A reduction of 5% means that the total accumulated number of events over all 4 cores needs to be reduced by 5% to meet the maximum of F messages per 1000 cycles. A system is considered repaired if this maximum number of messages is reached after the algorithm terminated. It can be seen that for all shapers, the evolutionary algorithm-based implementation has a consistently higher repair rate than its corresponding greedy heuristic-based implementation. On average among all reduction values examined, the evolutionary algorithm-based approach is able to repair 22.6% more systems than the greedy heuristic in the given timespan. Overall, the lazy leaky bucket rate shaper implemented with the EA shows the highest repair rates with close to 95% for all reduction values. In general,

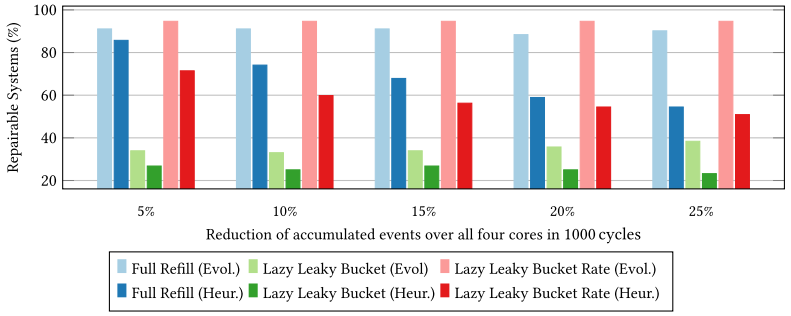


Fig. 9. Repair rates plotted against relative reduction of events in a time interval length of 1000 cycles.

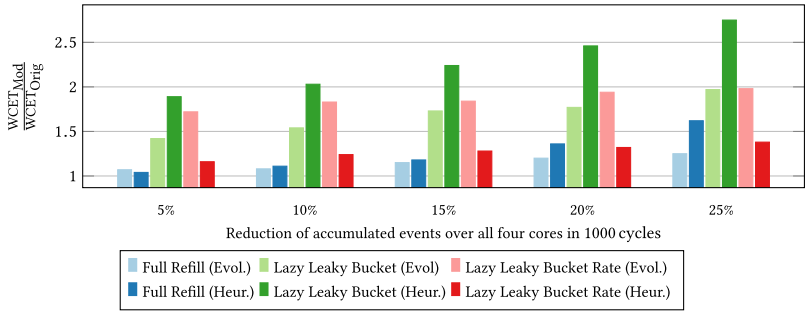


Fig. 10. Relative WCET change (median) plotted against relative reduction of events in a time interval length of 1000 cycles.

the EA-based shapers show a mostly stable repair despite an increasing number of events to be reduced. This most likely stems from the insertion of a heavily modified individual in the initial population which fulfills the requirements (yet with a very high WCET increase). The greedy heuristic shapers show a decline of repairable systems with an increasing number of events to be reduced, as the heuristic has to perform more iterations.

The impact on the WCET (median) is shown in Figure 10. Only systems which actually could attain the required reduction of events in the interval of 1000 cycles were included. As expected, the influence on the WCET increases for all shapers and both implementations with a growing number of events to be reduced. The lazy leaky bucket shaper implemented with the greedy heuristic shows the worst performance (median) with an increased WCET by a factor of 1.89 at 5% reduction and by a factor of 2.75 at 25%. The overall best performance is shown by the full refill shaper in combination with the EA, increasing the WCET by a factor of 1.07 at a reduction of 5% and by a factor of 1.25 at a reduction of 25%. On average among all reduction values and shapers examined, the evolutionary algorithm-based approach increases the WCET less by a factor of 0.038 in comparison to the greedy heuristic. Yet, as can be seen in the graph, this is heavily depending on the shaper.

Figure 11 shows the system schedulability after the shaper was applied. Only systems which actually could attain the required reduction of events are considered here. A system is considered schedulable if all cores can still meet their initially set deadline after the programs were modified. The best performance can be seen for the full refill shaper in combination with the EA and the lazy leaky bucket rate shaper in combination with the greedy heuristic. This resembles the results seen in the WCET evaluation. Similarly, the combination with the highest WCET increase (lazy leaky

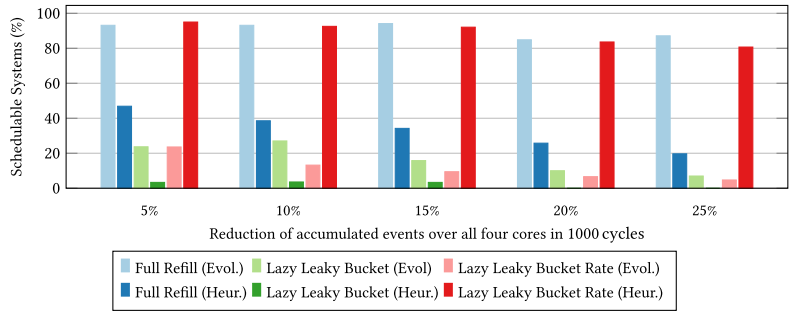


Fig. 11. Schedulable systems (%) plotted against relative reduction of events in a time interval length of 1000 cycles.

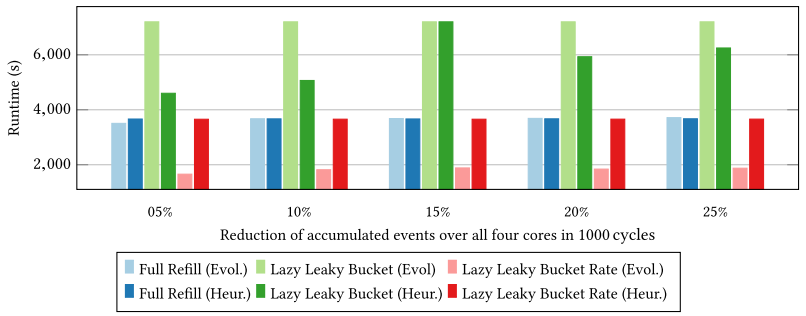


Fig. 12. Average (median) runtime plotted against relative reduction of events in a time interval length of 1000 cycles.

bucket & greedy heuristic) also results in the lowest number of schedulable systems. Comparing the EA and greedy heuristic among all reduction values and shapers, the greedy heuristic is able to generate 1.72% more schedulable systems on average.

The median runtimes of the different shapers and approaches is shown in Figure 12. As the EA was given a fixed amount of generations, the runtimes are mostly stable despite a growing number of events to reduce. Also the greedy heuristic shows a mostly constant runtime (except in combination with the LLB shaper) due to the relative ϵ threshold (as more delay has to be inserted, also the tolerated deviation between the upper and lower boundary during the binary search increases). The lazy leaky bucket shaper in combination with the EA shows the highest median runtime overall, followed by the combination with the greedy heuristic. This is most likely due to the extended ILP formulation. As described in Section 6.1, the profile adherence test was integrated by formulating the profile function $\sigma(\Delta t)$ into the ILP formulation. This increases the ILP's complexity and can lead to larger solving times. The lazy leaky bucket rate shaper in combination with the EA results in the lowest median runtimes. This most likely stems from the smaller combinatorial problem to be solved during the profile adherence check. As the Δt value to be checked is always lower for the lazy leaky bucket rate shaper in comparison to the full refill one (cf. step lengths of $\sigma(\Delta t)$ in Figure 6(b) and Figure 6(c)), the combinatorial complexity is expected to be lower. On average among all reduction values and shaper strategies evaluated, the EA requires approx. 7.9% less runtime compared against the greedy heuristic.

A shared bottleneck of both implementation strategies is the underlying extraction of event arrival functions $\eta(\Delta t)$, which is required to perform the profile adherence checking. As the ILP

model describing the event arrival function grows linearly with the number of basic blocks and event triggering instructions, the solving time grows exponentially in the worst case [28].

As a general conclusion when comparing the EA-based approach against the greedy heuristic, it can be seen that the evolutionary approach consistently outperforms the greedy heuristic in nearly all aspects, yet for some corner cases the greedy heuristic can deliver better results. The lazy leaky bucket rate shaper in combination with the greedy heuristic represents such a corner case. While in general the ratio of repairable systems is lower compared to the evolutionary approach here, the outcome in terms of WCET change and schedulability is in average better (*if* the system could be repaired successfully).

7 CONCLUSION AND FUTURE WORK

In this paper we presented a novel approach to integrate traffic shaping behavior into a program itself without relying on additional hardware or operating system features. The presented approach is able to transform a program such that its event arrival function will provably always adhere to a user-defined profile function while trying to be minimal invasive. We compared different profile functions and could show that adapting a traffic shaping behavior into programs can improve worst-case timings and limit overall traffic to a required amount.

As a part of future work we plan to combine this presented approach with additional WCET-aware optimizations to further improve worst-case timings in hard real-time multi-core architectures. Besides, the efficiency could be further improved by not only inserting delaying instructions, but also exploiting instruction rescheduling or pausing execution in a, e.g., co-operative multithreaded system. In case of such a co-operative multithreaded system, instead of inserting delaying instruction, the program could relinquish execution until enough cycles have passed while other tasks can be executed during this time. Furthermore, where possible, the interaction between our presented approach and HW-based traffic shapers or contention managing arbitration schemes, such as CSMA/CA, can be investigated. Here code-inherent traffic shaping can either work orthogonal to the existing approaches, e.g., to ease the requirements, or as an alternative.

ACKNOWLEDGMENTS

This work is part of a project that has received funding from the European Union Horizon 2020 research and innovation programme under grant agreement No. 779882.

REFERENCES

[1] The Motor Industry Software Reliability Association. 2013. MISRA C: 2012 Guidelines for the use of the C language in critical systems.

[2] S. K. Baruah. 2003. Dynamic- and static-priority scheduling of recurring real-time tasks. *Real-Time Systems* 24 (2003).

[3] S. Bleuler, M. Laumanns, L. Thiele, et al. 2003. PISAâa platform and programming language independent interface for search algorithms. In *Proceedings of EMO 2003*.

[4] C. G. Lee, P. Chow and M. G. Stoodley. [n.d.]. UTDSP Benchmark Suite. Retrieved 2019-04-05 from http://www.eecg.toronto.edu/ corinna/DSP/infrastructure/UTDSP.html.

[5] M. Potkonjak C. Lee and W. H. Mangione-Smith. 1997. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of MICRO 1997*.

[6] C. Chiasserini and R. R. Rao. 2001. Improving battery performance by using traffic shaping techniques. *IEEE Journal on Selected Areas in Communications* 19, 7 (2001).

[7] Robert I. Davis and Nicolas Navet. 2012. Traffic shaping to reduce jitter in controller area network (CAN). *SIGBED Rev.* 9, 4 (2012).

[8] H. Falk, S. Altmeyer, P. Hellinckx, et al. 2016. TACLeBench: A benchmark collection to support worst-case execution time research. In *Proceedings of the WCET 2016*.

[9] H. Falk and P. Lokuciejewski. 2010. A compiler framework for the reduction of worst-case execution times. *Real-Time Systems* 46, 2 (2010).

[10] L. Georgiadis, R. Guerin, V. Peris, and K. N. Sivarajan. 1996. Efficient network QoS provisioning based on per node traffic shaping. *IEEE/ACM Transactions on Networking* 4, 4 (1996).
[11] J. Gustafsson, A. Betts, A. Ermedahl, et al. 2010. The Mälardalen WCET Benchmarks – Past, Present and Future. In *Proceedings of WCET 2010*.
[12] M. R. Guthaus, J. S. Ringenberg, D. Ernst, et al. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of IISWC 2001*.
[13] B. Hu, K. Huang, G. Chen, et al. 2015. Adaptive runtime shaping for mixed-criticality systems. In *Proceedings of EMSOFT 2015*.
[14] ISO 11898-2:2016 2016. *Road Vehicles – Controller Area Network (CAN) – Part 2: High-speed Medium Access Unit*. Standard. International Organization for Standardization, Geneva, CH.
[15] M. Jacobs, S. Hahn, and S. Hack. 2015. WCET analysis for multi-core processors with shared buses and event-driven bus arbitration. In *Proceedings of RTNS 2015*.
[16] M. Joseph and P. Pandya. 1986. Finding response times in a real-time system. *Comput. J.* 29, 5 (1986).
[17] A. Kostrzewa, S. Saidi, L. Ecco, et al. 2016. Dynamic admission control for real-time networks-on-chips. In *Proceedings of ASP-DAC 2016*.
[18] M.-A. Kourtis, H. Koumaras, G. Xilouris, et al. 2017. An NFV-based video quality assessment method over 5G small cell networks. *IEEE MultiMedia* (2017).
[19] P. Kumar and L. Thiele. 2011. Cool shapers: Shaping real-time tasks for improved thermal guarantees. In *Proceedings of DAC 2011*.
[20] L. Pouchet. [n.d.]. PolyBench/C - The Polyhedral Benchmark Suite. Retrieved 2019-04-05 from http://www.cs.ucla.edu/~pouchet/software/polybench/.
[21] G. Laporte and S. Martello. 1990. The selective travelling salesman problem. *Discrete Applied Mathematics* (1990).
[22] J.-Y. Le Boudec and P. Thiran. 2001. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*.
[23] J. Lehoczky. 1990. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings of RTSS 1990*.
[24] Y.-T. S. Li and S. Malik. 1995. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of DAC 1995*.
[25] G. Memik, W. H. Mangione-Smith, and W. Hu. 2001. NetBench: A benchmarking suite for network processors. In *Proceedings of ICCAD 2001*.
[26] T. Nolte, H. Hansson, and L. L. Bello. 2005. Automotive communications-past, current and future. In *Proceedings of ETFA 2005*.
[27] D. Oehlert, S. Saidi, and H. Falk. 2018. Compiler-based extraction of event arrival functions for real-time systems analysis. In *Proceedings of ECRTS 2018*.
[28] C. H. Papadimitriou. 1981. On the complexity of integer programming. *J. ACM* (1981).
[29] L. T. X. Phan and I. Lee. 2013. Improving schedulability of fixed-priority real-time systems using shapers. In *Proceddings of RTAS 2013*.
[30] Y. Qian, X. Li, S. Ihara, et al. 2017. A configurable rule based classful token bucket filter network request scheduler for the lustre file system. In *Proceedings of the SC 2019*.
[31] M. Rahmani, K. Tappayuthpijarn, B. Krebs, et al. 2009. Traffic shaping for resource-efficient in-vehicle communication. *IEEE Transactions on Industrial Informatics* 5, 4 (2009).
[32] J. Reineke, B. Wachter, S. Thesing, et al. 2006. A definition and classification of timing anomalies. In *Proceedings of WCET 2006*.
[33] S. Shenker, C. Partridge, and R. Guerin. 1997. Specification of Guaranteed Quality of Service. IETF.
[34] StreamIt Community. 2018. The StreamIt Benchmark Suite. http://groups.csail.mit.edu/cag/streamit/shtml/benchmarks.shtml.
[35] J. Turner. 1986. New directions in communications (or which way to the information age?). *IEEE Communications Magazine* 24, 10 (1986).
[36] R. Underwood, J. Anderson, and A. Apon. 2018. Measuring network latency variation impacts to high performance computing application performance. In *Proceedings of the ICPE 2018*.
[37] E. Wandeler. 2006. *Modular Performance Analysis and Interface-Based Design for Embedded RealTime Systems*. Ph.D. Dissertation. ETH Zürich.
[38] E. Wandeler, A. Maxiaguine, and L. Thiele. 2006. Performance analysis of greedy shapers in real-time systems. In *Proceedings of DAC Europe 2006*.
[39] J. Wang, C. Sung, and C. Wang. 2019. Mitigating power side channels during compilation. *arXiv e-prints* (2019).
[40] M. Wu, Sh. Guo, P. Schaumont, et al. 2018. Eliminating timing side-channel leaks using program repair. *arXiv e-prints* (2018).

[41] Y. Zhou and D. Wentzlaff. 2016. MITTS: Memory inter-arrival time traffic shaping. In *Proceedings of ISCA 2016*.
[42] E. Zitzler, M. Laumanns, and L. Thiele. 2001. *SPEA2: Improving the Strength Pareto Evolutionary Algorithm*. Technical Report. ETH Zürich.
[43] V. Zivojnović, J. M. Velarde, C. Schläger, et al. 1994. DSPSTONE: A DSP-oriented benchmarking methodology. In *Proceedings of IC-SPAT 1994*.

Received April 2019; revised June 2019; accepted July 2019