Future Generation Computer Systems 179 (2026) 108337

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

A Portable Compiler-Runtime Approach for Scalability Prediction

Nicolai Stawinoga [a,*], Sohan Lal [b], Biagio Cosenza [c,**], Philip Salzmann [d], Peter Thoman [d], Thomas Fahringer [d]

[a] Technische Universität Berlin, Embedded Systems Architecture Group, EN-12, Einsteinufer 17, Berlin, 10587, Germany
[b] Technische Universität Hamburg, Massively Parallel Systems Group (E-EXK5), Am Schwarzenberg-Campus 3 (E), 4.046, Hamburg, 21073, Germany
[c] University of Salerno, Dipartimento di Informatica, Via Giovanni Paolo II, 132, Fisciano (Salerno), 84084, Italy
[d] University of Innsbruck, Research Centre High Performance Computing (FZ HPC), Distributed and Parallel Systems Group, Institute for Computer Science, Technikerstr. 21 a, Innsbruck, A-6020, Austria

ARTICLE INFO

Keywords:
High-Performance computing
Scalability
Feature engineering
Machine learning.

ABSTRACT

Highly scalable parallel applications can efficiently solve expensive computational problems when run on a large number of compute nodes. However, selecting the optimal number of nodes for a compute job of a given size is non-trivial, and allocating too few or too many nodes may not yield the expected performance. Knowing the scaling behavior of an application in advance enables us, for example, to make optimal use of the available hardware resources. We introduce a novel, portable approach to predict the scalability of parallel applications written in modern high-level programming models. We propose a predictive compiler-runtime framework based on Celerity, a task-based distributed runtime system that enables executing SYCL codes on clusters. The framework targets a broad range of computing systems, from CPU to GPU clusters, and proposes a model that combines machine learning, communication modeling and DAG heuristics. Experimental results on two large-scale clusters, JUWELS and Marconi-100, show accurate scalability prediction of unseen single and multi-task applications.

1. Introduction

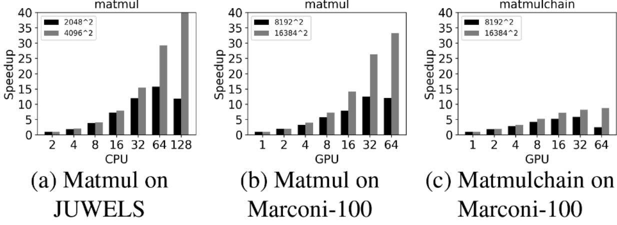
The scalability of software is crucially important for large-scale HPC systems. It is desirable not only to have scalable parallel systems [1], on which the speedup increases proportionally with the number of processors, but also to understand when such systems do not scale anymore. Scalability predictions are important for various reasons, e.g.: to find scalability bugs in complex codes [2]; to understand when applications should avoid slow secondary storage and networks, leaving the CPU as the main performance bottleneck [4,5]; to avoid wasting resources beyond the point at which they achieve good speedup [6]. Unfortunately, predicting scalability is a difficult task as it depends on different factors including application and algorithm characteristics, problem size, communication patterns, network topology, targeted hardware, as well as intricacies of the software stack.

Consider the examples given in Fig. 1. Fig. 1(a) shows a matrix multiply on a CPU cluster with different input sizes. With a smaller input size (2048^2) the performance initially increases as the number of CPUs is increased, but will eventually drop when the number of CPUs becomes too large. Larger input sizes (e.g., 4096^2) instead show a graceful performance degradation when increasing the number of CPUs, with an overall good scalability. The same code, however, may exhibit a different scaling behaviour on different computing systems: a CPU cluster Fig. 1(a) scales gracefully with very different combination of input sizes and number of devices with respect to a GPU cluster (Fig. 1(b)); for system configurations refer to Section 5.1). Additionally, large applications comprising of multiple tasks have a more complex scalability behaviour: while a standalone matmul shows good scaling (Fig. 1(a)), it scales poorly within a matmulchain application, because of the larger data movement (Fig. 1(c)).

Fig. 2 presents the overall approach of this work. We assume that, within a restricted execution model, it is possible to achieve more accurate and automatic scalability prediction in both single and multi-task scenarios. Although efforts have been spent to automate the task of performance modeling [7], of which scalability prediction is a special case, to the best of our knowledge there is no related work that proposes scalability predictions inherently integrated within a high-level programming model which can target distributed memory GPU clusters. We implement our research in Celerity [8], a programming model that provides a modern, restrictive and simplified execution model based on

* Corresponding authors.
** Principal corresponding author.
E-mail addresses: nicolai@aes.tu-berlin.de (N. Stawinoga), sohan.lal@tuhh.de (S. Lal), bcosenza@unisa.it (B. Cosenza), philip.salzmann@uibk.ac.at (P. Salzmann), peter.thoman@uibk.ac.at (P. Thoman), tf@uibk.ac.at (T. Fahringer).

https://doi.org/10.1016/j.future.2025.108337
Received 9 May 2025; Received in revised form 11 November 2025; Accepted 21 December 2025
Available online 25 December 2025
0167-739X/© 2026 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).



(a) Matmul on JUWELS (b) Matmul on Marconi-100 (c) Matmulchain on Marconi-100

Fig. 1. Scalability of two SYCL/Celerity applications on JUWELS and Marconi-100.

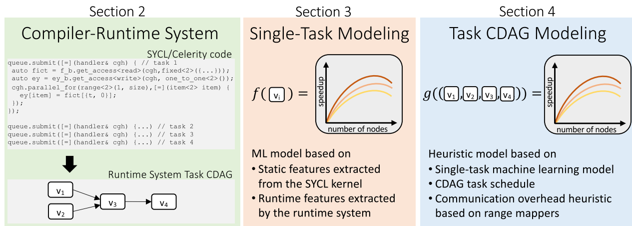


Fig. 2. Proposed prediction framework.

SYCL [9]. Celerity provides high-level semantics for data requirement specification (i.e., *range mapper*) and is supported by a distributed runtime system [10] that organizes the computation in a task *computation directed acyclic graph* (CDAG). We choose Celerity as the basis for our research as it provides a basis for executing a variety of algorithms with distinct load characteristics and data transfer requirements on CPU and GPU clusters within a single framework while featuring a high-level representation of data flow.

The prediction methodology comprises two modeling steps. The first step is a single-task model based on machine learning, which requires model training only once per system rather than per application. The training of the regression model uses static and run-time features on a small set of benchmarks, and is able to predict the scalability of *unseen* kernels. This means, using cross-validation we exclude all data related to one kernel from the model training set so as to predict the entire scalability curve, without relying on known data points and without using extrapolation or interpolation. Secondly, a multi-task application model extends the single-task model based on CDAG heuristics, while taking data movement into account. Our approach here is to micro-benchmark inter-node data transfers assuming a black-box approach to network topology. Altogether, the two-stage approach is able to predict the full scalability curves of unseen programs, which to the best of our knowledge has not been attempted.

Overall, the contributions of this work are:

- A portable compiler-runtime framework based on the SYCL/Celerity programming model, which is capable of extracting static information from kernels as well as dynamic information from the distributed runtime system (Section 2).
- A machine learning-based model operating on static and runtime feature representations, which predicts the scalability of single-task applications (Section 3). The proposed methodology is abstract to the hardware and is able to predict the scalability for both CPU and GPU clusters.
- An extended scalability prediction model for multi-task applications, which uses the computational DAG provided by Celerity's runtime system and the provided range mappers to carefully model the data movement (Section 4).
- An experimental evaluation demonstrating portability on two large-scale systems, JUWELS and Marconi-100 (equipped respectively with multi-core CPUs and GPUs), and including error model analyses, static and dynamic feature evaluation, and code correlation study explaining how the model automatically learns from the training data to predict the scalability of unseen applications (Section 5).

2. A Compiler-Runtime Approach

Large-scale clusters of accelerators are difficult to program. Therefore, scientists often prefer high-level programming models supported by runtime systems to low-level approaches such as MPI+X. We propose a prediction framework that is fully integrated into a high-level programming model and leverages both the underlying compiler and runtime system to derive accurate scalability predictions. This section describes the programming model (2.1), the compilation infrastructure (2.2), and the distributed runtime system (2.3).

2.1. Programming model

Our scalability prediction framework takes as input an application written in SYCL with Celerity extension and runtime system for cluster programming.

SYCL [9,11] is a programming model for heterogeneous computing that builds on modern C++. While SYCL follows the execution and memory model of OpenCL [12,13], implementations can have a non-OpenCL mapping, e.g., the Adaptive C++ CUDA backend [14]. SYCL supports single-source programming where both kernel and host codes are stored in the same source file, enabling e.g., C++ templates to work seamlessly and in a type-safe manner across boundaries of host and device code. When accelerators are targeted, a dedicated compiler identifies, extracts and compiles kernels into an intermediate representation or machine code. The default memory model for handling memory is the buffer-accessor model, where the application running on the host uses SYCL buffer objects to allocate memory in the global address space. To access a memory object, the user must create an *accessor* object that parameterizes the type of access to the memory object that a kernel or the host requires. An accessor object specifies whether the access is via global memory, constant memory or image samplers, and their associated access functions. The accessor also explicitly specifies whether the access is read-only, write-only, or read-write. The execution of data-parallel kernels is organized by a task graph, which is implicitly constructed by the runtime based on data access specifications that the programmer associates with a kernel using accessor objects.

Celerity [8] is built on top of SYCL by extending its API for distributed memory clusters, which substantially simplifies migration of existing single-node single-device SYCL applications to cluster-based multi-CPU and multi-GPU systems. Celerity's extensions comprise two features: While a SYCL program consists of a queue used to submit commands to a compute device, Celerity introduces a *distributed queue*, which represents all compute devices in the cluster as a single virtual device. Each buffer's accessor is associated to a functor indicating its data requirements called *range mapper*. In particular, range mappers are a way to express data dependencies and enable the runtime to efficiently implement data distribution among all cluster nodes.

Several built-in range mappers are provided for common data access patterns, including `one-to-one`, `neighborhood`, `slice`, `all`, and `fixed`. The `one-to-one` mapper specifies that a kernel, for every individual work item, will access that buffer only at that same global index. The `slice` mapper extends the range along one dimension indefinitely, thus selecting an entire slice of a buffer in that dimension; e.g., the matrix multiplication in Listing 1 uses two slice mappers for rows and columns. `neighborhood` is used for stencil-like codes, while `all` selects the entire buffer.

In this paper, we extend the Celerity programming model and distributed runtime system with an LLVM-based kernel compilation infrastructure, including a static feature extractor, a runtime extension to collect runtime features, and a scalability modeling framework.

```
queue.submit([=](handler& cgh) {
  accessor a{mA, cgh, slice<2>(1), read_only};
  accessor b{mB, cgh, slice<2>(0), read_only};
  accessor c{mC, cgh, one_to_one{}, write_only, no_init};
  cgh.parallel_for<class matmul>(range<2>(SIZE, SIZE),
  [=](item<2> itm) {
    float sum = 0.f;
    for(size_t k = 0; k<SIZE; k++)} {
     const float a_ik = a[{itm[0], k}];
     const float b_kj = b[{k, itm[1]}];
     sum += a_ik * b_kj;
    }
    c[item] = sum;
  });
});
```

Listing 1. A matrix multiplication in SYCL/Celerity (see queue and range mappers), including both host and kernel code.

2.2. Compilation infrastructure

The input codes are compiled with a SYCL compiler. As SYCL codes include both host and kernel code, existing implementations such as Adaptive C++[1] [14], Intel's oneAPI Data Parallel C++ compiler [15] or formerly CodePlay's ComputeCPP [16] provide a way to return the extracted kernel in an intermediate representation such as SPIR-V or LLVM bitcode. Our approach includes a set of LLVM passes, which analyze all kernel functions in the IR and provide a feature vector for each kernel. The feature representation aims at characterizing the kernel, e.g., to understand if it is dominated by floating point computation or memory accesses. This will act as the basis for our single task scalability prediction which we describe in Section 3.

2.3. Distributed runtime system

Based on the data access specifications and range mappers provided in the programming model, the runtime system can efficiently distribute data and computation among the available devices.

First, the runtime executes a *prepass*, which is similar to the normal program execution but, instead of actually executing the kernels collects dependency and scheduling meta-information in a *task graph*. This strategy also allows the runtime to infer the input size of each task before the actual execution. The graph used by the runtime is a representation of the planned execution of the kernels in the application: a vertex is a task, i.e., a direct mapping of a SYCL/Celerity kernel execution on the whole cluster with its entire execution space; a direct edge represents a data dependency between two tasks. As such, there are no loops (e.g., loops iterating a kernel execution are unrolled in the execution plan). A Celerity task graph is formally modeled as a *computation directed acyclic graph* (CDAG). We will describe the CDAG in more detail in Section 4 in the context of using it as the basis for our multi-kernel scalability prediction.

The prepass is executed on each node in an MPI-like fashion and generates the task graph; it is the same on all nodes. The number of nodes affects the generation of the command graph, but that happens asynchronously based on the task graph data generated by the prepass.

After DAG creation, a scheduler constructs a more detailed *command graph* from existing CDAG nodes, which includes individual instructions for each device in the system and encodes all necessary data transfers to maintain a consistent view of data. During this step, the scheduler asynchronously splits individual tasks into multiple commands, which represent work ranges of the sub-task to be executed on a single remote device.

The Celerity (distributed) execution model extends SYCL so that each kernel is executed on all devices in the cluster, in a data-parallel fashion: for example, in a GPU cluster, each GPU executes the same kernel on

[1] Formerly known as hipSYCL.

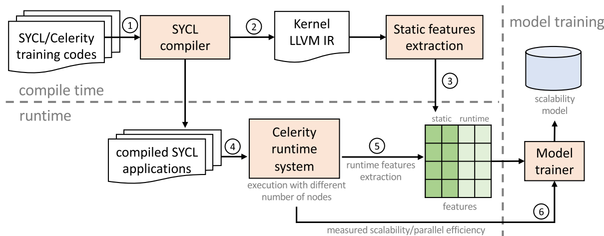


Fig. 3. Scalability model training phase.

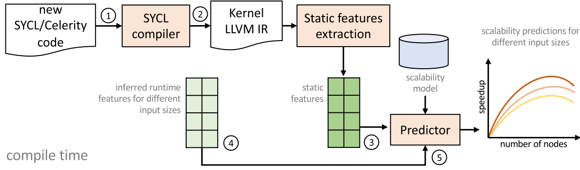


Fig. 4. Model inference for different problem sizes.

different execution ranges (and therefore on different data). Details of the Celerity execution model and scheduler are available in Celerity's reference papers [8,17].

3. Single-Task Prediction

The goal of this work is the prediction of the scalability of a program as a function of the number of devices. The prediction framework comprises two components: a scalability prediction model for single-task applications based on machine learning, and a multi-task model that reuses the single-task model on top of a computational DAG for heuristic application scalability predictions. This section provides an overview of the single-task prediction (3.1), and details regarding the feature extraction (3.2) and model training (3.3).

3.1. Overview

In this section, we assume that we have an application containing a single kernel that is executed only once. A scalability prediction for a similar single-task application is calculated with a two-phase machine learning methodology comprising of a training and inference phase. The proposed methodology is abstract to the hardware and can be used to predict the scalability of a program on a GPU cluster as well as a CPU cluster.

The **training phase**, which builds the model, is shown in Fig. 3. ① The input SYCL/Celerity codes used for training are given as input to the SYCL compiler. ② The compiler generates the intermediate representation for each kernel , ③ which is analyzed by a LLVM compilation pass that extracts the code features and saves them into a feature set. ④ The compiled training applications will be executed with different problem sizes and increasing number of nodes. For each parallel execution, ⑤ the runtime system will extract a set of runtime features and ⑥ measure the scalability. Finally, once all the training data has been collected, the model trainer will train a regression model, which will be saved for later use. Model generation is automatic and can be easily ported to any SYCL-supported compute systems.

Once the scalability model is available, it can be used in different scenarios. Fig. 4 shows how the model can be used within a compiler tool to predict the scalability of a kernel with different sizes. For the **scalability prediction** of a new input SYCL/Celerity code with a single kernel, ① the code is provided as input to the SYCL compiler, ② its kernel is translated into LLVM IR and ③ statically analyzed to extract static features. To have a scalability prediction before running the application, ④ we evaluate the runtime features for different problem sizes and number of nodes. Once all the runtime features are combined with

the static ones, ⑤ the predictor generates a scalability plot that predicts the scaling behaviour of the input application with different problem sizes.

3.2. Feature extraction

We use a feature engineering framework to characterize a parallel task execution, including the problem size and the number of compute nodes/devices, which summarizes all this information into a single feature vector. The feature vector conveniently encodes static code features, extracted by an LLVM pass on the intermediate representation generated for each kernel, and dynamic runtime features, which are extracted by the runtime system during the *prepass*.

The **static code features** have been designed to express the computation and memory behaviour of a kernel; the initial design was inspired by architectural consideration on GPU component architecture [18–20] and extended to CPU. The static feature set comprises 15 values: bitwise instructions; integer add/sub and multiplication; `f32` and `f64` add/sub, division and multiplication; load and store; other instructions; a feature representing the ratio of coalesced memory access over the total number of accesses; the total number of instructions. Formally, static features are represented by a feature vector $\vec{k}$, where every single component is normalized in $[0, 1]$. We use different normalization techniques according to the feature value distribution. The static feature extractor needs to know the execution count of each feature. The extraction of static features is computed by an LLVM pass that implements Kofler at al. [21] heuristics with the following steps. Initially, each instruction (in a basic block) has an assigned contribution of 1. After running the loop simplify analysis and imposing the Loop Closed SSA form on each loop, the contribution of each instruction in the loop is multiplied according to Kofler et al. heuristic: If the loop count is static, each instruction in that loop is multiplied by the loop count. If unknown, the heuristic assumes 100 iterations are performed. This heuristic is very simple but works surprisingly well with loops and loop nests of unknown iteration counts. The final calculation is performed using a dataflow analysis which takes the maximum contribution when different paths are taken (join operator). The static feature representation is based on high-level LLVM IR and does not contain information that prevents the portability of the approach on different target architectures.

The **runtime features** are dynamically extracted by a runtime system. They comprise the number of global work items, the size of input and output buffers normalized over the global size.

Fig. 1 shows a summary of both static and runtime features used in the predictive modeling approach.

Overall, a *parallel task execution* q is represented by a feature vector, which is a concatenation of the vectors with static and dynamic features, and the number of nodes n (or devices) used for that execution, i.e., $\vec{q} = (\vec{k}, \vec{d}, n)$.

Table 1
List of static and runtime features. The normalization allows for feature scaling so that feature values are in [0,1]. The value range represents the observed values in the training data before normalization.

Feature	Normalization	Value range
number of input buffers	min-max scaling	0.6
number of output buffers	min-max scaling	1.4
bitwise instructions	ratio (total insts)	0.903
integer add & sub	ratio (total insts)	3.30608
integer multiplication	ratio (total insts)	1.10203
32-bit floating point add & sub	ratio (total insts)	0.10000
32-bit floating point multiplication	ratio (total insts)	0.10000
32-bit floating point division	ratio (total insts)	0.100
64-bit floating point add & sub	ratio (total insts)	0.1
64-bit floating point multiplication	ratio (total insts)	0.1
64-bit floating point division	ratio (total insts)	0.1
load	ratio (total insts)	6.20012
store	ratio (total insts)	1.501
other	ratio (total insts)	13.61125
total number of instructions	min-max scaling	
problem size	log scaling	
number of devices	log scaling	

3.3. Model training

The task of predicting the scalability of a single-task data-parallel execution is formulated as a regression problem. The scalability behaviour of a code on a given computing system can be typically expressed as a unimodal function with a peak representing the maximum number of nodes before performance degradation. Formally, given a set of M parallel program executions in the training set T, we define a training sample of input-output pairs $(q_1, s_1), \dots, (q_M, s_M)$, where $q_i \in T$, and each kernel execution of q_i is associated to its measured speedup s_i. Therefore, the model is represented by the following function $f(\vec{q}) = s_i$. In practice, to demonstrate model generalization we dynamically separate the data into distinct test and training sets using leave-p-out cross-validation, which we explain in more detail in Section 5.2.

To model such a function, we experimentally evaluate several regression approaches: *Support Vector Regression* (SVR), *K-Nearest Neighbors* (KNN) and *Random Forest* (RF). We used machine learning methods that typically applies to small training set and have been also successfully used in similar other tuning problems [21–23]. As each approach exposes several parameters, a grid search has been performed to select those parameter configurations that yield higher accuracy. SVR [24] is a support vector machine designed for regression tasks. We tested several values for the free parameters (C and ϵ) and evaluated different kernel functions (the most accurate kernel was the radial basis function). KNN is a regression approach based on the k-nearest neighbors algorithm; after experimentally testing several values, the most accurate model used three neighbors. RF is a meta-estimator that fits a number of classifying decision trees on sub-samples of the dataset using averaging to improve the accuracy and to control over-fitting. Several parametrizations have been evaluated, and our final most accurate model uses 100 estimators and *Mean Squared Error* (MSE) to measure the quality of a split. RF is the approach resulting in the most accurate predictions, as our evaluation will show in Section 5.2. We further improve the accuracy by lowering the feature dimensionality using *Principal Component Analysis* (PCA) [25].

4. Multi-Task Prediction

Modern parallel applications are composed of many tasks organized in a complex control flow. To predict the scalability of such applications, we designed a framework that extends the single-task modeling outlined in Section 3 to a multi-task scenario.

Fig. 5 shows the overall approach for multi-task prediction. The first three steps are the same as for a single-task application: ① an input SYCL/Celerity code has ② its static features are extracted statically, for each kernel, by the compiler; ③ this set of feature is further extended with dynamic features collected by the distributed runtime system. ④ A Celerity application is modeled by a computational task graph where each vertex represents a task, i.e., a kernel execution, and edges correspond to data dependencies between tasks; this information, together with the range mapper associated to each data dependency and ⑤ the pre-trained per-task scalability model, will later be used for multi-task predictions ⑥.

We derive an application scalability model by using the single-task prediction function of each task. In a single-task scenario, there is no data movement between nodes and devices, as data is allocated in a distributed way before kernel execution. However, in a multi-task application, data communication may occur in between subsequent kernel executions in order to satisfy data dependencies. In a SYCL/Celerity

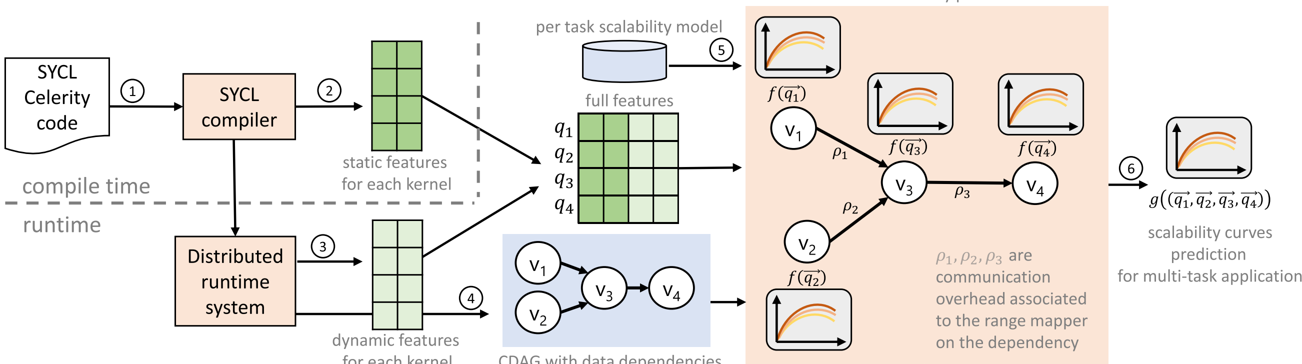


Fig. 5. Multi-task prediction, based on the single-task scalability model, the collected features, the CDAG and the communication overhead associated to each data dependency.

application, data dependencies and placement are expressed by the range mapper associated with a buffer's accessor. The key insight is that, given the restricted execution model, the volume and the type of message passing are entirely dependent on the combination of the range mappers of the previous write and the current read operation on a given buffer. Therefore, multi-task modeling should consider both task DAG and the range mappers.

This section introduces the task model used to represent the computation (4.1), a simple prediction model that ignores communication (4.2), and an enhanced communication-aware model (4.3).

4.1. Task model

We model the execution of a parallel program by using a *computation directed acyclic graph* (CDAG) as defined by Bilardi and Peserico [26]. In our case: tasks are kernels executed on all allocated devices of the cluster and represented by graph vertices; inter-task dependencies are captured by graph edges. Formally, a CDAG is a 4-tuple $C = (V, I, O, E)$ of finite sets such that: V is the set of vertices; $I \subseteq V$ is the input set and its vertices have no incoming edges; $O \subseteq V$ is the output set and its vertices have no outgoing edges; $E \subseteq V \times V$ is the set of edges; $G = (V, E)$ is a directed acyclic graph.

A CDAG does not specify an order of execution of the task, but only partial ordering constraints as edges. Considering that each task is actually executed on the whole cluster, e.g., on all devices available on the system, it is important to consider what schedule is applied to understand the scalability behaviour of an application. Formally, we define a schedule as a topological sorting $\tau = (v_1, \dots, v_N)$ of the elements of V so that whenever $(v_i, v_j) \in E$, then $i < j$.

For the Celerity version we used for this work, the order of execution of the task is the same order as they are placed on the distributed queue; therefore, the order must be predicted in advance for modeling purposes.

Given a task $v_i \in V$, we implicitly refer to the vector $\vec{q_i}$ as the feature representation of the task v_i. We also assume that we already have a function f that, given a single task t_i, correctly predicts the scalability of this task by using it's associated feature vector, i.e, $f(\vec{q_i})$. Our goal is to define a function $g(\tau)$ that predicts the scalability for the sequence of tasks in the schedule τ.

Recall that Celerity's runtime uses a prepass to capture data dependencies in the CDAG before the actual program execution. Similarly, once a prediction function is available for each kernel, no application execution is required for application prediction. In fact, the scalability tool works as a standalone tool (on a single node) that returns the prediction with different input sizes and number of nodes.

```
cgh.parallel_for(range<2>(N,N),
  [=](item<2> item) {
    auto sum = 0.f;
    for(size_t k = 0; k<N; k++)} {
       sum += a[{item[0],k}]*b[{k,item[1]}];
    }
    c[item] = sum;
  });
});
...
cgh.parallel_for(range<2>(N,N),
  [=](item<2> item) {
    auto sum = 0.f;
    for(size_t k = 0; k<N; k++)} {
        sum += a[{item[0],k}]*b[{k,item[1]}];
    }
    c[item] = sum;
  });
```

Listing 2. Code exerpt from 2mm.

4.2. Communication-less prediction

We first introduce a simpler model for the scalability prediction of a multi-task application by ignoring the data communication. The resulting prediction will be an over-approximation of the scalability function.

In the case of a single task application with $\tau = (v_1)$, the scalability of the application is trivially calculated by using the single-task scalability function f, i.e., $g((v_1)) = f(\vec{k_1})$. By ignoring data communication between task executions, the application model becomes a simple aggregation of the single-task scalability functions in the schedule. Eq. 1 introduces a simple heuristic, which defines the speedup as an average of all predicted speedups in the schedule, i.e.

$$g(\tau) = \frac{1}{N} \sum_{i=0}^{N} f(\vec{k_i}) \tag{1}$$

This heuristic makes the assumption that all kernels contribute equally to the overall scalability. To overcome this limitation, we propose in Eq. 2 a heuristic based on a weighted average:

$$g(\tau) = \sum_{i=0}^{N} f(\vec{k_i}) w_i \tag{2}$$

After the prepass and before the kernel execution, the runtime system is aware of the number of global work items of each task; weights w are initialized to the global sizes to handle unbalance due to different per-task input sizes. However, we found applications such as `syrk`, where unbalance comes from the inner structure of the kernel, e.g., the number of loops. For this reason, we refine the weight calculation by considering the loop information extracted statically for each kernel at compile time, and by normalizing the weights so that $\sum_{i=0}^{N} w_i = 1$.

```
cgh.parallel_for(range<2>(N,N),
  [=](item<2> item) {
    res[item] *= beta;
  });
});


...

cgh.parallel_for(range<2>(N,N),
  [=](item<2> item) {
    const auto i = item[0];
    const auto j = item[1];
    for(size_t k = 0; k<N; k++){
       res[item] += alpha*A[{i, k}]*A[{j, k}];
    }
  });
});
```

Listing 3. Code exerpt from syrk.

For clarity, we show an example of kernel weight calculation for `2mm` and `syrk`. For both kernels, we assume an execution with an input size of 8194^2. For `2mm` (Listing 2), we have two matrix multiplications with the same number of instructions and the same size; in this case, the weights are 0.5 and 0.5. In `syrk` (Listing 3), we have two different kernels. Although they both have the same input size, they contribute differently to the overall program execution. The first kernel has three instructions. For the second, we adopt the loop heuristic used in Section 3.2, which yields a total number of instructions of 903. By considering both input size and instruction count, we get normalized kernel weights of 0.003 and 0.997 for the two kernels, respectively, which reflects the fact that the second kernel is largely dominant.

4.3. Data movement overhead

Eq. 2 over-approximates the scalability because it does not consider the data communication, which incurs a severe overhead on modern large-scale compute clusters, at the point to severely limit the scalability. In a Celerity application, the data distribution policies are expressed by the programmer through the range mapper semantic, which is used by the runtime system to move (part of the) data buffer on those nodes that actually need it. Therefore, different range mappers will have a different impact on the scalability as they will transfer different amounts of data: given the restricted execution model, data movement mostly depends on the range mapper combination on read-after-write (RAW) dependencies.

To better understand the relation between range mapper and data communication, we show the behaviour of three multi-task applications with different communication behaviour (Fig. 6). In `syrk` (Fig. 6(a)), the first kernel writes to a buffer marked by a `one-to-one` range mapper, which is later read by the second task using again a `one-to-one` range mapper. As both range mappers indicate the same partitioning, no extra data movement occurs. In terms of modeling, the application scalability is largely dependent on the scaling behaviors of its tasks, as modeled by Eq. 2. In `matmulchain` (Fig. 6(b)), however, two dependencies connect buffer accesses (first written, then read) with different range mappers: `one-to-one` → `slice_d1` and `one-to-one` → `slice_d0`. We experimentally observed that the latter involves a larger communication overhead, affecting the scalability, which would otherwise be similar as `matmul` if communication was ignored. In `fdtd2d` (Fig. 6(c)), two out of three dependencies have a range mappers combination that identifies the halo zone data exchange of a stencil-like computation: `one-to-one` → `neighborhood<2>`. The communication is relatively small compared to `matmulchain`, but the overall scalability is low because of the poor scaling of the individual kernels.

To measure the data movement generated by the range mappers, we designed a benchmark that evaluates read-after-write (RAW) buffer dependencies with all possible combinations. The benchmark is based on a matrix addition kernel. For each combination, the communication overhead is calculated and normalized in [0, 1], where 0 means no data movement and 1 represents the maximum communication overhead.

Table 2
Communication overhead for selected range mappers on read-after-write dependencies.

Range Mapper Combinations	Communication Overhead ρ	
Write → Read	Marconi-100	JUWELS
all → all	0.986	0.874
one-to-one → all	0.922	0.983
one-to-one → slice x	0.863	0.999
…		
one-to-one → slice y	0.117	0.018
all → neighborhood 3x3	0.113	0.024
neighborhood 3x3 → neighbor. 3x3	0.012	0.017
one-to-one → neighborhood 3x3	0.012	0.018
…		
all → one-to-one	0.001	0.025
one-to-one → one-to-one	0.001	0.014

Table 2 shows the factors of communication overhead ρ for relevant combinations of range mappers on both target systems. In the case of `one-to-one` → `one-to-one` dependencies, the overhead is negligible in both systems. Typical stencil transitions from `one-to-one` to `neighborhood` have a slightly larger overhead. Column- and row-wise combinations require more communication, while those including the `all` range mapper are the worst.

The final heuristic considers the communication overhead ρ as a penalty. Given $\rho_1, \ldots, \rho_M$ overheads for M data dependencies, the final prediction function is defined as

$$g(\tau) = \max\left(\rho_1, \ldots, \rho_M\right) \sum_{i=0}^{N} f(\vec{k}_i) w_i \tag{3}$$

In case of multiple dependencies, Eq. 3 only takes into account of the most communication-intensive range pattern combination. The heuristic already accounts for communication/computation overlapping, which also occurs in the range mapper benchmark used for the calculation of ρ. Intuitively, our heuristic states that scalability is largely dependent on the communication bottleneck and applies a penalty ρ accordingly. We are aware that this may not be accurate in all cases, and that some cases may require jointly normalising the communication overhead and compute overhead. However, we did not encounter a case requiring this in the two benchmark suites used throughout our evaluation, and must therefore leave this current limitation for future work.

In summary, a simple computation scenario involving a single task requires only single-task prediction. For a multi-task scenario where all tasks share the same range mapper, there is no data movement overhead; in this case, the contribution from Eq. 3 is negligible, and Eq. 2 can be used. However, for all other, more complex scenarios, the more accurate Eq. 3 is required, as it accounts for data movement overhead.

5. Evaluation

We experimentally evaluated the single-task and multi-task application models using different methodologies. This section provides an overview of the experimental setup (Section 5.1), an accuracy evaluation of the single-task scalability model (Section 5.2), an analysis of the static code features (Section 5.3) and feature importance (5.4), and the accuracy evaluation of the multi-task model (5.5).

5.1. Experimental setup

We performed our experiments on two large-scale compute clusters: the JUWELS CPU cluster at Juelich Supercomputing Center and the Marconi-100 GPU cluster at CINECA. Where not explicitly stated, results refer to Marconi-100.

Marconi-100 nodes host 2x16 cores IBM POWER9 AC922 each at 3.1 GHz with 256 GB/node of RAM memory and 4 x NVIDIA Volta V100

Table 3
Range mappers for read/write accessors and kernel information for 12 single-task and 8 multi-task applications.

Benchmark	Source	# Tasks	Range Mappers R/W
median	SYCL-Bench	1	neighborhood / one-to-one
vecadd	SYCL-Bench	1	one-to-one / one-to-one
2Dconv	PolyBench	1	neighborhood / one-to-one
gemm	PolyBench	1	one-to-one / one-to-one
gesummv	PolyBench	1	one-to-one, slice y / one-to-one
matmul	SYCL-Bench	1	slice x, slice y / one-to-one
sobel3	SYCL-Bench	1	neighborhood / one-to-one
sobel5	SYCL-Bench	1	neighborhood / one-to-one
sobel7	SYCL-Bench	1	neighborhood / one-to-one
seidel	SYCL-Bench	1	neighborhood / neighborhood
jacobi1d	SYCL-Bench	1	neighborhood / neighborhood
jacobi2d	SYCL-Bench	1	neighborhood / neighborhood
syrk	PolyBench	2	one-to-one, one-to-one / one-to-one one-to-one, slice y / one-to-one
2mm	PolyBench	2	slice x, slice y / one-to-one slice x, slice y / one-to-one
matmulchain	SYCL-Bench	3	slice x, slice y / one-to-one slice x, slice y / one-to-one slice x, slice y / one-to-one
atax	PolyBench	3	– / one-to-one slice y, all / one-to-one slice x, all / one-to-one
fdtd2d	PolyBench	4	one-to-one, neighborhood, fixed / one-to-one neighborhood, one-to-one / one-to-one neighborhood, one-to-one / one-to-one neighborhood, neighborhood, one-to-one / one-to-one
bicg	PolyBench	2	slice y / one-to-one slice x, all / one-to-one
covariance	PolyBench	3	slice x / one-to-one slice y / one-to-one all / slice y
correlation	Polybench	4	slice x / one-to-one slice x, one-to-one / one-to-one one-to-one, all, all / one-to-one all / slice y

Mappers: one-to-one, slice x, slice y, neighborhood, fixed, all

GPUs, Nvlink 2.0. The network is a Mellanox Infiniband EDR DragonFly+. We used the Adaptive C++ implementation of SYCL 1.2.1 [9] with a CUDA backend [14] and Celerity v0.2.1 running with IBM Spectrum MPI 10.3.1. We base our data on the median run time of 5 repeated runs. We repeat each experiment for two different input sizes for $1, 2, 4, \dots 64$ GPUs (the maximum number of nodes allocatable on the default user production queue is 16). The baseline represents the parallel SYCL/Celerity code running on a single GPU.

JUWELS' CPU batch partition consists of 2271 compute nodes with Dual Intel Xeon Platinum 8168 (2x 24 cores at 2.7 GHz, 12x 8 GB at 2666 MHz) interconnected by an EDR-Infiniband (Connect-X4) network. We used Intel's DPCPP implementation of SYCL 2020 [11] with a SPIR-V target backend [15] and the newest Celerity v0.3.1 running with Intel MPI 2021.4.0.

Input sizes have been carefully chosen to conduct meaningful scalability experiments for strong scaling across all benchmarks so that they are neither too large to be reasonably executed on a single node, but at the same time sufficiently large to obtain meaningful results on a higher number of nodes.

Table 4
Different error metrics for SVR, KNN, and RF.

Error metric	SVR	KNN	RF
Mean squared error	25.30	12.40	11.40
Mean squared log error	1.11	0.21	0.17
Mean absolute percentage error	36.34	4.01	4.76

Table 3 shows twelve single-task and eight multi-task applications used throughout our evaluation, as well as their input/output range mappers. They have been taken from two distinct sets of existing application benchmarks: SYCL-Bench [27], which has been extended with Celerity annotations, and PolyBench [28], which has been ported to SYCL and extended with Celerity annotations[2] , for a total of 30 unique kernels. Kernels from the multi-task benchmarks are also extracted and used as single-task benchmarks. Any kernel used in several multi-task benchmarks is only counted once towards that total, e.g., `matmul`.

5.2. Single-task scalability predictions

We build the single-task model on a training set comprising of 30 kernels (see Table 3) and trained the model on the features extracted from the static codes, extended with the runtime features including the size and the number of devices. Additionally, we reduced the feature dimensionality by using PCA with six components.

To check model generalization to an independent data set, we apply a leave-p-out cross-validation, omitting p samples from training and using these for testing. Specifically, we define p to refer to *all* samples for one benchmark, with different input sizes and number of nodes such that, effectively, $p = 14$. Omitting all samples for one benchmark from the training process as well as its associated feature set ensures that testing effectively happens on previously unseen code. We repeat the process for all benchmarks in order to cross-validate the results. The training data generation takes 1h while the training time is relatively small (e.g., <1 sec for SVR).

Regression approaches. We evaluate the three modeling approaches by calculating aggregate error metrics on all tested combinations of cross-validated codes, and with different input sizes and number of GPUs. The accuracy is reported with three different error metrics. Table 4 shows the mean squared error, mean squared log error, and mean absolute percentage error for SVR, KNN, and RF. The RF regression has the lowest error among the three models. The mean squared log error and mean absolute percentage error is only 0.17 and 4.76 for RF. KNN has similar accuracy as RF. SVR has the highest error.

Scalability details on selected kernels. Fig. 7 shows in detail the scalability behaviour of four applications with two different input sizes on Marconi-100. They exhibit different scalability behaviour, from a very good (`matmul` and `gemm`), to moderate (`sobel5` and `sobel3`). The model understands the difference between computationally intensive kernels (i.e., with better scalability), as compared to kernels with very little computation (i.e., modest or weak scalability). Thanks to the runtime features, the model correctly evaluates the impact of the input size, thus understanding which sizes lead to a drop in scalability. We observed a few prediction mismatches in the transition from 4 to 8 GPUs (contained in 1 and 2 nodes, respectively), for instance in `sobel3` and `sobel5` with size 8192^2. In this range, the scalability function exhibits a sudden drop because each node is equipped with 4 GPUs. While the effects of scaling to more than one node is usually captured by the model, it fails

[2] Available at: https://github.com/bcosenza/celerity-bench

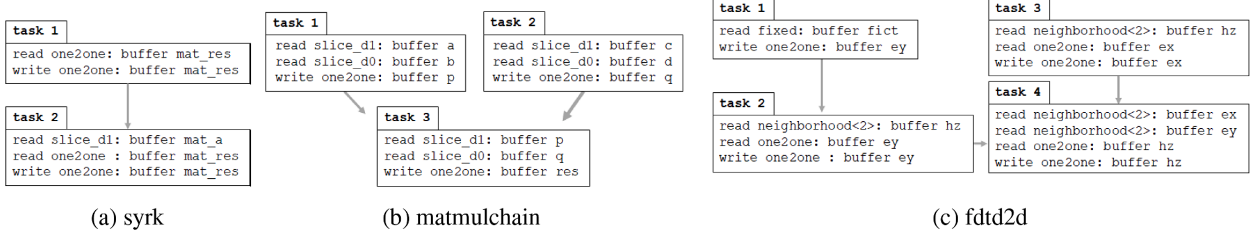


Fig. 6. Task DAG with buffer's range mapper information for three selected applications.

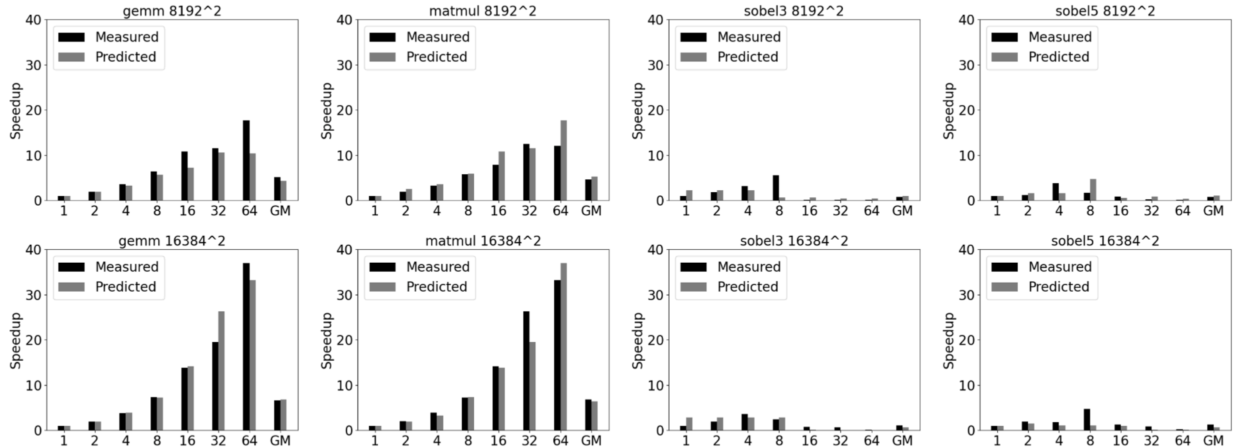


Fig. 7. Scalability of four selected single-task applications on Marconi-100 (speedup/number of GPUs).

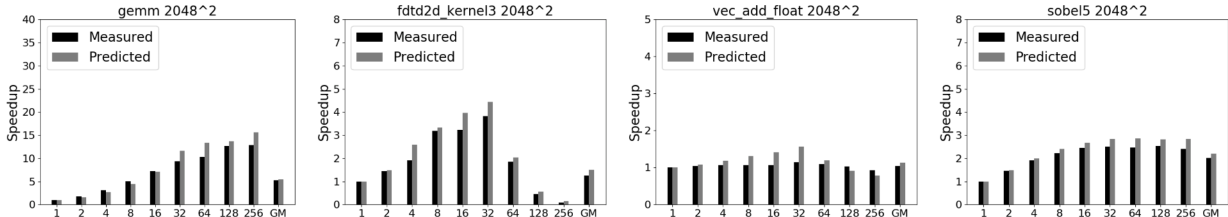


Fig. 8. Scalability of four selected single-task applications on JUWELS (speedup/number of CPUs).

in these few cases. Distributing the computation across several nodes, which may be allocated in different groups or potentially different islands in the DragonFly + network, means that network traffic becomes an issue. As we ran our experiments on a live production system under load, we experienced noticeable variability in the measurements obtained. A recent study on the same cluster highlights the effects of such background traffic [29]. As such, any discrepancy between measured and predicted speedup may also be due to noise in the measurements, especially where the number of GPUs is larger than 4. Fig. 7 also shows the geometric mean (GM) of the measured and predicted speedup as an aggregate metric. Overall, we see that there is not a big difference between the GM of predicted and measured speedups. For example, `gemm` has an overall measured speedup of 6.58 and a predicted speedup of 6.82 for the 16384^2 size. We selected GM as it is the preferred metric for calculating the mean of ratios such as speedup. Moreover, it is always lower than or equal to the arithmetic mean and always higher than or equal to the harmonic mean and is thus often interpreted as the more honest mean [30].

Fig. 8 shows three selected codes on JUWELS, demonstrating model portability to a different target system.

5.3. Static code feature analysis

Static code features distance analysis. To understand how the model learns from the 30 kernels in the training data, we analyzed how the static features extracted by those kernels are distributed and whether there are similarities between codes. Fig. 9 shows the distances in the feature space between different kernels. The distance has been computed using the cosine similarity metric. Distance values are in $[0, 1]$, darker means distance close to zero.

The analysis shows several kernel clusters, which means that for each kernel there is at least one other similar kernel in feature space.

Loop heuristics. The benchmarks exhibit a variety of loop structures, which affects static feature extraction. Of the 30 kernels, 4 kernels have statically known loop bounds (`median` and the three `sobel` variants). Our analysis uses this static trip count to correctly weigh features inside the loop body. 13 kernels have dynamically-sized loops with a dependency on the problem size. For the problem sizes used in our experiments, we found that the majority of loops typically had 8K-64M iterations in single or double-loop nests.

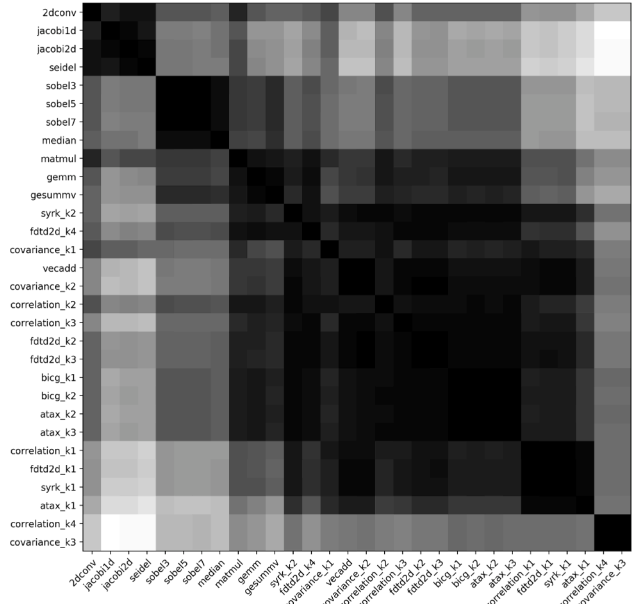


Fig. 9. Pairwise static feature cosine similarity among the 30 kernels used for the single-kernel model. Distance values are in $[0, 1]$. Darker means distance close to zero.

Table 5
Top five features using greedy selection.

Rank	Feature	Score loss
1	number of processors (runtime)	0.5504
2	input size (runtime)	0.6020
3	load instructions (static)	0.6023
4	f32.addsub (static)	0.6026
5	input buffers (static)	0.6026
Full model		0.9734

5.4. Feature importance analysis

Given our scalability prediction model, we generate insight into what features are the most important for an accurate prediction. For this purpose, we used a greedy feature selection algorithm that ranks the feature set based on the accuracy of the prediction model. The algorithm starts with the full feature set and iteratively selects the feature that produces the largest model error when the model is trained without that feature. This selects the single best feature of the model. In the next iteration, the algorithm selects a second feature, which, combined with those already selected, produces the largest model error for a model trained with two features. The algorithm then continues selecting features until a given number. We chose as error metric the model's score, which is the R^2 coefficient. Table 5 shows the top-ranked features using greedy forward selection.

In related work [6], only input features such as data set size and processor grid dimensions are considered (i.e., our runtime features). To investigate whether our extended feature set made of both static and dynamic features is a better application characterization for modeling, we tested our model with static-only and runtime-only features. With static-only features, we get a per-application constant model that has 8.5 MAPE for RF, which means a 1.8× decrease in accuracy. With runtime-only features, we get an input-dependent but application-unaware model that has 39.1 MAPE for RF, which means a 8.2× decrease in accuracy. Our model uses static and runtime features resulting in better accuracy.

5.5. Multi-Task scalability predictions

We employ our previous findings to develop and evaluate a modeling approach for predicting multi-task scalability. Fig. 10 compares the measured speedup with the predicted speedup using both the simpler model in Eq. 2 and the communication-aware formulation in Eq. 3. The figure also shows the GM of both the measured and predicted speedups. We show results for five multi-task applications, sorted by scalability, including applications that both have good and bad scaling behaviour. For the measured speedup, each experiment was repeated five times, and we reported the median.

Results show that, for applications with both high (e.g., `syrk`) and low scalability (e.g., `fdtd2d`) which are mostly limited by the single-kernel scaling rather than communication, the simple model is already good. However, applications like `matmulchain` require a more accurate communication-aware model. Overall, our communication-aware model accurately predicts the scalability of all five tested applications for the two input sizes. We observe a decrease in the MAPE from 0.91 to 0.82 while using the communication-aware model. Additionally, the model correctly understands when an application exhibits poor scalability due to smaller problem sizes.

6. Related Work

Scalability Prediction. Scalability modeling is a special case of performance modeling. Table 6 summarizes the most relevant state-of-the-art work on scalability prediction. Barnes et al. [6] extrapolate scalability for a known program on a multi-core cluster using a regression-based prediction mechanism based entirely on run-time features. PEMOGEN [7] applies a profiling-based approach to automatically generate performance models during run-time, and can effectively interpolate and extrapolate scalability for a known program. The LLVM-based compilation and modeling framework uses LASSO regression on a set of features extracted from automatically identified kernels. The authors extend their work by supporting combinations of static and dynamic analyses and investigating reducing the model search space [31]. ScaAnalyzer [5] is a tool that pinpoints scaling losses due to poor memory access ESTIMA [5] extrapolates the scalability of known programs on a multi-core system. The presented profiling-based approach is interestingly based on stall cycles. In contrast, we aim to predict scalability for an unseen application where training is required once per system rather than per application, and where execution data of the application is not required.

Seminal theoretical work by Grama et al. [1] introduces the isoefficiency model to analytically reason about extrapolating scalability by modeling e.g., the degree of concurrency. Coarfa et al.[32] presented a top-down scalability analysis tool combining path profiles and expectations to discover poor scaling behaviors. Hoefler and Kwasniewski [33] presented an automatic method to symbolically count loop iterations and derive parallel work and depth from the loop count models. Schudler et al. [34] introduced an automated testing procedure to validate the asymptotic scaling trends of parallel libraries. Grain graphs [35] is an OpenMP performance analysis method that highlights problems such as low parallelism, work inflation, and poor parallelization benefit. Performance-detective [36] analyses parameter interactions to deduce an optimized, minimal experiment design and is orthogonal to the applied performance modeling approach. ScalAna [37] introduces compiler and runtime lightweight techniques to generate a performance graph and perform a graph analysis algorithm to detect the root cause of scaling issues. Arndt et al. [38] introduced a statistical model that, given the scalability of a given benchmark/platform, predicts the performance on different benchmarks/platforms by using distance metrics and interpolation. The sample-based strategy has been used by Wei and Mellor-Crummey [39] for the diagnosis of scalability losses; their framework analyzes sample-based time series data to diagnose scalability losses in parallel executions. In contrast to profiling-based related work,

Table 6
Comparison to state-of-the-art scalability prediction approaches.

	Model input	Model re-use	Target	Comm. modeling	Prediction for
Barnes et al. [6]	runtime features	per application	multi-core cluster	critical path on MPI tasks	scaling extrapolation
PEMOGEN [7]	profiling + instrumentation	per application	multi-core cluster	implicit	analysed application
ESTIMA [5]	stall cycles	per application	multicore	limited to NUMA	scaling extrapolation
ScaAnalyzer [3]	lightweight profiling	per application	multicore	limited to NUMA	analysed application
Choi et al. [41]	profiling roofline + MPI traces	per application	cluster of GPUs	MPI traces	analysed application
This work	static + runtime features, CDAG	per system	cluster of CPUs, cluster of GPUs	range mappers CDAG	unknown application

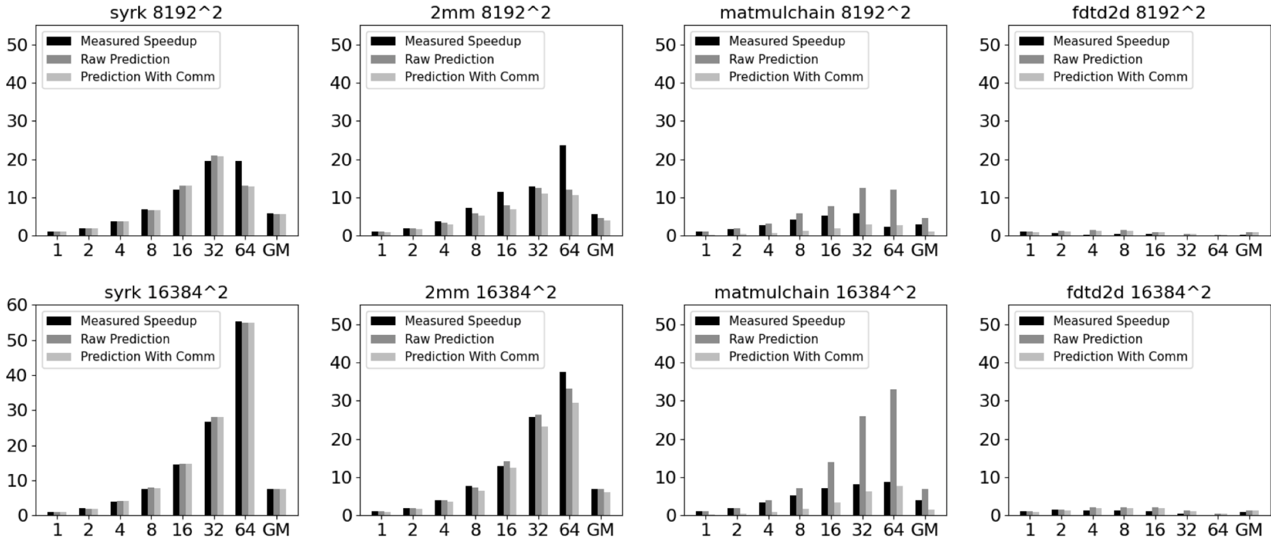


Fig. 10. Predicted versus measured speedup of multi-task applications with two sizes (speedup/GPUs).

our per-kernel prediction model is based on a combination of static features extracted by a compiler and dynamic ones, extracted by a runtime system.

The machine learning techniques used in this article are commonly applied in cases where the sample data set is of a containable size. As an example, Braun et al. [23] use random forest trained on code features to predict execution times on a single GPU, whereas our goal is to predict a normalised speedup on a cluster, as the number of GPUs changes. The authors also provide a careful review of a large body of work similar to theirs. Cummins et al. [40] apply gated-graph neural networks to a graphical program representation (ProGraML) which can model data dependencies; however, in our case, the feature representation focuses on identifying what major factors affect the scalability such as compute vs. memory boundness and data movement; as such, we do not require this more sophisticated representation.

Communications Modeling. Barnes et al. [6] analyse several methods for communication modeling, of which critical-path MPI communication modeling is the most complex method investigated. It poses an interesting contrast to the black-box approach they propose for modeling computation (see discussion above). PEMOGEN [7] has implicit communication modeling. This is also the case for ESTIMA [5], where it is implied by stall cycles, but limited to one NUMA domain. More recently, Choi et al. [41] presented a prediction framework for clusters of GPUs, which combines a profiling-based roofline model for GPU kernels and a trace-driven simulation for MPI communications. In contrast, our work relies on an execution model that is more restrictive than generic MPI applications (i.e., data transfers are precisely described by *range mappers*) and simplifies communications modeling. Alternatively, our approach potentially applies to generic MPI applications if there is a way to identify communication patterns, e.g., using static dataflow analysis [42], collective identifications [43] and frameworks such as ParFuse [44].

7. Conclusions

We presented an approach for predicting the scalability of parallel applications written in SYCL/Celerity for compute clusters. The predictive framework integrates a task-based distributed runtime system and compiler to collect accurate information regarding the applications to be predicted. The modeling approach is based on machine learning for single-task applications, which is extended using CDAG heuristics for more complex multi-task applications. After a deployment phase in which setup and model training are performed, the resulting per-system model can be reused to predict the scalability of any new, previously unseen application. The approach has been validated on two large-scale high-performance computing clusters, JUWELS and Marconi-100. Results show that the approach is portable and accurately predicts the scalability of single- and multi-task applications on both target computing systems.

Limitations. For single-task prediction, the approach can be applied to any heterogeneous programming model, provided the compiler toolchain can be instrumented to extract both static code features and dynamic features, which are then used for machine learning-based prediction. In contrast, for multi-task prediction, the method requires an explicit representation of the computation DAG (e.g., tracking data dependencies), as well as mechanisms to automatically partition and transfer data across devices. In our work, we leverage Celerity for this purpose; in principle, the same technique can be applied to other programming models with a distributed runtime and knowledge of dependency structures, such as Legion or OmPSs.

Future work. In future work, we will deploy our scalability tool as a shared module across all clusters at CINECA and JSC, thereby providing users with a tool to make the most efficient use of computational resources so as to avoid over-allocation. Alternatively, the runtime could

also decide to use allocated resources differently and introduce task parallelism guided by scalability predictions, for instance where it is expected that a given task does not scale efficiently across all allocated nodes. This would require deeper integration of the runtime and the prediction model and must be left for future work. While our approach allows us to model systems without the use of hardware or network features, it would be interesting to explore the addition of hardware features in combination with transfer learning, making not just the approach but the trained model itself portable. We plan to extend our experimental evaluation to include two larger benchmarks, CloverLeaf and MiniWeather.

CRediT authorship contribution statement

Nicolai Stawinoga: Writing – original draft, Software, Methodology, Investigation, Funding acquisition, Conceptualization; **Sohan Lal:** Writing – original draft, Software, Methodology, Investigation, Conceptualization; **Biagio Cosenza:** Writing – original draft, Software, Methodology, Investigation, Funding acquisition, Conceptualization; **Philip Salzmann:** Software; **Peter Thoman:** Writing – review & editing, Software, Funding acquisition; **Thomas Fahringer:** Supervision.

Data availability

No data was used for the research described in the article.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This document is the results of the research project partially funded by FWF (I 3388) and DFG (CO 1544/1-1, project number 360291326) as part of the DACH project CELERITY.

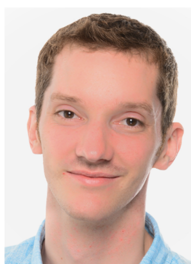
The authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time through the John von Neumann Institute for Computing (NIC) on the GCS Supercomputer JUWELS at Jülich Supercomputing Centre (JSC).

The authors gratefully acknowledge the CINECA award under the ISCRA initiative for the availability of high-performance computing resources and support.

References

[1] A. Grama, A. Gupta, V. Kumar, Isoefficiency: measuring the scalability of parallel algorithms and architectures, IEEE Parallel Distributed Technol. Syst. Appl. 1 (3) (1993) 12–21. https://doi.org/10.1109/88.242438
[2] A. Calotoiu, T. Hoefler, M. Poke, F. Wolf, Using automated performance modeling to find scalability bugs in complex codes, in: International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17–21, 2013, 2013, pp. 45:1–45:12.
[3] X. Liu, B. Wu, Scaanalyzer: a tool to identify memory scalability bottlenecks in parallel programs, in: Kern, J., Vetter, J.S. (Eds.), Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15–20, 2015, 2015 ACM. pp. 47:1–47:12. https://doi.org/10.1145/2807591.2807648.
[4] H. Lim, D. Han, D.G. Andersen, M. Kaminsky, MICA: A holistic approach to fast in-Memory key-Value storage, in: Proceedings of the 11Th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2–4, 2014, 2014, pp. 429–444.
[5] G. Chatzopoulos, A. Dragojević, R. Guerraoui, ESTIMA: Extrapolating scalability of in-Memory applications, ACM Trans. Parallel Comput. 4 (2) (2017) 10:1–10:28.
[6] B.J. Barnes, B. Rountree, D.K. Lowenthal, J. Reeves, B. de Supinski, M. Schulz, A regression-based approach to scalability prediction, in: Proceedings of the 22Nd Annual International Conference on Supercomputing, ICS '08, ACM, New York, NY, USA, 2008, pp. 368–377.
[7] A. Bhattacharyya, T. Hoefler, PEMOGEN: Automatic adaptive performance modeling during program runtime, in: International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24–27, 2014, 2014, pp. 393–404.
[8] P. Thoman, P. Salzmann, B. Cosenza, T. Fahringer, Celerity: high-Level c + + for accelerator clusters, in: Euro-Par 2019: Parallel Processing - 25Th International Conference on Parallel and Distributed Computing, 2019, pp. 291–303.
[9] K.S.W. Group, SYCL Specification, Version 1.2.1 Revision 6, Technical Report, Khronos Group, 2019.
[10] P. Salzmann, F. Knorr, P. Thoman, P. Gschwandtner, B. Cosenza, T. Fahringer, An asynchronous dataflow-Driven execution model for distributed accelerator computing, in: Proceedings of the 23Rd IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGrid23, 2023.
[11] K.S.W. Group, SYCL 2020 Specification, revision 3, Technical Report, Khronos Group, 2021.
[12] K.O.W. Group, OpenCL 3.0 API Specification, Technical Report, Khronos Group, 2021.
[13] K.O.W. Group, OpenCL 3.0 C Language Specification, Technical Report, Khronos Group, 2021.
[14] A. Alpay, V. Heuveline, SYCL Beyond opencl: the architecture, current state and future direction of hipsycl, in: IWOCL '20: International Workshop on OpenCL, 2020, p. 8:1. https://doi.org/10.1145/3388333.3388658
[15] Intel, oneAPI Data Parallel C + + compiler, 2022, Online; accessed 1 Feb 2022, https://github.com/intel/llvm.
[16] Codeplay Software, ComputeCpp, 2022, (????). Online; accessed 1 Feb 2022, https://codeplay.com/solutions/ecosystem/#our-sycl-based-solutions.
[17] P. Salzmann, F. Knorr, P. Thoman, P. Gschwandtner, B. Cosenza, T. Fahringer, An asynchronous dataflow-Driven execution model for distributed accelerator computing, in: Y. Simmhan, I. Altintas, A.L. Varbanescu, P. Balaji, A.S. Prasad, L. Carnevale (Eds.), 23Rd IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGrid 2023, Bangalore, India, May 1–4, 2023, IEEE, 2023, pp. 82–93. https://doi.org/10.1109/CCGrid57682.2023.00018
[18] C. Isci, M. Martonosi, Runtime power monitoring in high-End processors: methodology and empirical data, in: Proceedings of the 36Th Annual International Symposium on Microarchitecture, San Diego, CA, USA, December 3–5, 2003, 2003, pp. 93–104.
[19] J. Guerreiro, A. Ilic, N. Roma, P. Tomás, GPGPU Power modeling for multi-domain voltage-Frequency scaling, in: IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24–28, 2018, 2018, pp. 789–800.
[20] K. Fan, B. Cosenza, B.H.H. Juurlink, Predictable GPUs frequency scaling for energy and performance, in: Proceedings of the 48Th International Conference on Parallel Processing, ICPP 2019, Kyoto, Japan, August 05–08, 2019, 2019, pp. 52:1–52:10.
[21] K. Kofler, I. Grasso, B. Cosenza, T. Fahringer, An automatic input-sensitive approach for heterogeneous task partitioning, in: International Conference on Supercomputing, ICS, 2013, pp. 149–160.
[22] D. Grewe, Z. Wang, M.F.P. O'Boyle, Portable mapping of data parallel programs to openCL for heterogeneous systems, in: Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23–27, 2013, IEEE Computer Society, 2013, pp. 22:1–22:10. https://doi.org/10.1109/CGO.2013.6494993
[23] L. Braun, S. Nikas, C. Song, V. Heuveline, H. Fröning, A simple model for portable and fast prediction of execution time and power consumption of GPU kernels, ACM Transactions on Architecture and Code Optimization (TACO) 18 (1) (2020) 1–25.
[24] H. Drucker, C.J.C. Burges, L. Kaufman, A.J. Smola, V. Vapnik, Support vector regression machines, in: Advances in Neural Information Processing Systems 9, NIPS, 1996, pp. 155–161. http://papers.nips.cc/paper/1238-support-vector-regression-machines.
[25] C.M. Bishop, Pattern recognition and machine learning, 5th Edition, Information science and statistics, Springer, 2007. https://www.worldcat.org/oclc/71008143.
[26] G. Bilardi, E. Peserico, A characterization of temporal locality and its portability across memory hierarchies, in: F. Orejas, P.G. Spirakis, J. van Leeuwen (Eds.), Automata, Languages and Programming, Springer Berlin Heidelberg, Berlin, Heidelberg, 2001, pp. 128–139.
[27] S.L. Heuveline, A. Alpay, P. Salzmann, B. Cosenza, A. Hirsch, N. Stawinoga, P. Thoman, T. Fahringer, Vincent, SYCL-Bench: A versatile cross-Platform benchmark suite for heterogeneous computing, in: Euro-Par 2020: Parallel Processing, 2020.
[28] L.-N. Pouchet, PolyBench/C 3.2, 2012, https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/.
[29] M. Salimi Beni, B. Cosenza, An analysis of long-Tailed network latency distribution and background traffic on dragonfly +, in: International Symposium on Benchmarking, Measuring and Optimization, Springer, 2022, pp. 123–142.
[30] T. Hoefler, R. Belli, Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '15, Association for Computing Machinery, New York, NY, USA, 2015. https://doi.org/10.1145/2807591.2807644
[31] A. Bhattacharyya, G. Kwasniewski, T. Hoefler, Using compiler techniques to improve automatic performance modeling, in: 2015 International Conference on Parallel Architectures and Compilation, PACT 2015, San Francisco, CA, USA, October 18–21, 2015, 2015, pp. 468–479.
[32] C. Coarfa, J.M. Mellor-Crummey, N. Froyd, Y. Dotsenko, Scalability analysis of SPMD codes using expectations, in: B.J. Smith (Ed.), Proceedings of the 21Th Annual International Conference on Supercomputing, ICS 2007, Seattle, Washington, USA, June 17–21, 2007, ACM, 2007, pp. 13–22. https://doi.org/10.1145/1274971.1274976

[33] T. Hoefler, G. Kwasniewski, Automatic complexity analysis of explicitly parallel programs, in: Proceedings of the 26Th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '14, ACM, New York, NY, USA, 2014, pp. 226–235.
[34] S. Shudler, A. Calotoiu, T. Hoefler, A. Strube, F. Wolf, Exascaling your library: will your implementation meet your expectations?, in: Proceedings of the 29Th ACM on International Conference on Supercomputing, ICS'15, Newport Beach/Irvine, CA, USA, June 08 - 11, 2015, 2015, pp. 165–175.
[35] A. Muddukrishna, P.A. Jonsson, A. Podobas, M. Brorsson, Grain graphs: openMP performance analysis made easy, in: Proceedings of the 21St ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '16, ACM, New York, NY, USA, 2016, pp. 28:1–28:13.
[36] L. Schmid, M. Copik, A. Calotoiu, D. Werle, A. Reiter, M. Selzer, A. Koziolek, T. Hoefler, Performance-detective: automatic deduction of cheap and accurate performance models, in: Proceedings of the 36Th ACM International Conference on Supercomputing, 2022, pp. 1–13.
[37] Y. Jin, H. Wang, X. Tang, T. Hoefler, X. Liu, J. Zhai, Identifying scalability bottlenecks for large-Scale parallel programs with graph analysis, in: Proceedings of the 25Th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'20, Association for Computing Machinery, New York, NY, USA, 2020, p. 409–410.
[38] O.J. Arndt, M. Lüders, H. Blume, Statistical performance prediction for multicore applications based on scalability characteristics, in: 2019 International Conference on Application-specific Systems, Architectures and Processors (ASAP), 2160-052X, 2019, pp. 255–262.
[39] L. Wei, J.M. Mellor-Crummey, Using sample-based time series data for automated diagnosis of scalability losses in parallel programs, in: PPoPP '20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2020, pp. 144–159.
[40] C. Cummins, Z.V. Fisches, T. Ben-Nun, T. Hoefler, M.F.P. O'Boyle, H. Leather, Programl: a graph-based program representation for data flow analysis and compiler optimizations, in: International Conference on Machine Learning, PMLR, 2021, pp. 2244–2253.
[41] J. Choi, D.F. Richards, L.V. Kalé, A. Bhatele, End-to-end performance modeling of distributed GPU applications, in: E. Ayguadé, W.W. Hwu, R.M. Badia, H.P. Hofstee (Eds.), ICS '20: 2020 International Conference on Supercomputing, Barcelona Spain, June, 2020, ACM, 2020, pp. 30:1–30:12. https://doi.org/10.1145/3392717.3392737
[42] G. Bronevetsky, Communication-Sensitive static dataflow for parallel message passing applications, in: Proceedings of the CGO 2009, the Seventh International Symposium on Code Generation and Optimization, Seattle, Washington, USA, March 22–25, 2009, IEEE Computer Society, 2009, pp. 1–12. https://doi.org/10.1109/CGO.2009.32
[43] T. Hoefler, T. Schneider, Runtime detection and optimization of collective communication patterns, in: P. Yew, S. Cho, L. DeRose, D.J. Lilja (Eds.), International Conference on Parallel Architectures and Compilation Techniques, PACT '12, Minneapolis, MN, USA - September 19, - 23, 2012, ACM, 2012, pp. 263–272. https://doi.org/10.1145/2370816.2370856
[44] S. Aananthakrishnan, G. Bronevetsky, M. Baranowski, G. Gopalakrishnan, Parfuse: parallel and compositional analysis of message passing programs, in: C. Ding, J. Criswell, P. Wu (Eds.), Languages and Compilers for Parallel Computing - 29Th International Workshop, LCPC, 10136 of *Lecture Notes in Computer Science*, Springer, 2016, pp. 24–39. https://doi.org/10.1007/978-3-319-52709-3_3



Nicolai Stawinoga received his Ph.D. degree at the Department of Computing at Imperial College London. He now works as an independent PostDoc at the Technical University of Berlin. His research interests include accelerator programming, compiler frameworks, software performance, and high-performance computing.

Sohan Lal is a junior professor and head of the Massively Parallel Systems Group at TU Hamburg, a position he has held since September 2021. Prior to this, he was a postdoctoral researcher at TU Berlin, where he also completed his Ph.D. in Computer Science under the supervision of Prof. Ben Juurlink. His research specializes in GPU architecture, with broader interests in power and performance modeling, parallel systems, memory systems, heterogeneous computing, and applied machine learning. He has presented his work at leading conferences such as IPDPS, DATE, ICCD, and ISPASS, earning a Best Paper Award at HPEC 2024.

Biagio Cosenza is an Associate Professor in the Department of Computer Science at the University of Salerno (Italy). From 2015 to 2019, he was Senior Research at the TU Berlin (Germany), where he was Principal Investigator for the DFG project Celerity and received his Habilitation from the Faculty IV. From 2011 to 2015, he was Postdoctoral researcher at the University of Innsbruck (Austria), where he contributed to the Insieme Compiler project and the DK-Plus scientific platform. Cosenza's research interests are in the area of programming models and high-performance computing, and he is a member of the Khronos SYCL Working Group.

Philip Salzmann is a PhD student at the Distributed and Parallel Systems Group at the University of Innsbruck, Austria, from where he has also received his BSc and MSc degrees. His research interests include high performance accelerator computing, system design and software architectures as well as high-level API design with a focus on productivity.

Peter Thoman is an Assistant Professor at the University of Innsbruck, Austria, where he also obtained his PhD. He is one of the original developers and designers of the Insieme C++ research compiler and the Celerity runtime system for accelerator clusters, and was involved in multiple international research projects in this capacity. In particular, he led the core technical work package in the Horizon 2020 AllScale project, the Ligate FET HPC project, and the D-A-CH Celerity project. Peter's research interests include accelerator computing, fine-grained task parallelism, compiler-supported optimizations, and API and runtime system design for parallelism.

Thomas Fahringer received his Ph.D. degree from the Vienna University of Technology in 1993. Since 2003, he has been a full professor of computer science at the Institute of Computer Science, University of Innsbruck, Austria. His main research interests include software architectures, programming paradigms, compiler technology, performance analysis, and prediction for parallel and distributed systems. He is a member of the IEEE.