MUSA: A Multi-Level Simulation Approach for Next-Generation HPC Machines[1]

Thomas Grass*† César Allande† Adrià Armejach† Alejandro Rico‡ Eduard Ayguadé*†
Jesus Labarta*† Mateo Valero*† Marc Casas† Miquel Moreto*†
*Universitat Politècnica de Catalunya †Barcelona Supercomputing Center ‡ARM Inc.

***Abstract*—The complexity of High Performance Computing (HPC) systems is increasing in the number of components and their heterogeneity. Interactions between software and hardware involve many different aspects which are typically not transparent to scientific p rogrammers a nd s ystem a rchitects. Therefore, predicting the behavior of current scientific applications on future HPC infrastructures is a challenging task.**

In this paper we present MUSA, an end-to-end methodology that employs a multi-level simulation infrastructure. By combining different levels of abstraction, MUSA is able to model the communication network, microarchitectural details and system software interactions, providing different trade-offs in terms of simulation cost and accuracy. We compare detailed MUSA simulations with native executions of up to 2,048 cores and find r elative e rrors t hat a re w ithin 1 0% i n t he c ommon case. In addition, we use MUSA to simulate up to 16,384 cores and successfully identify scalability bottlenecks due to different factors, e.g. memory contention or load imbalance. We also compare different system configurations, s howing h ow MUSA can help system designers to assess the usefulness of future technologies in next-generation HPC machines.

I. Introduction

The process of designing next-generation High Performance Computing (HPC) machines is extremely challenging. The increasing amount of computational resources each generation integrates makes this challenge even more difficult. In addition, the trend to use commodity server processors as the common choice for designing such machines is changing, as processors with leaner core designs that feature significantly different microarchirectural characteristics are starting to make their debut in the HPC market [40, 52, 54]. Consequently, the design space for next-generation HPC machines is expanding. Novel solutions are required in order to quickly predict the performance of current and future scientific a pplications on those systems and to identify the best design points.

Besides taking into account the hardware, it is important to also consider its interactions with the system software (e.g. operating system, runtime system) [12, 50]. Hybrid programming models are pervasive nowadays, employing MPI for inter-node communication and a shared-memory programming model for node-level parallelism. Motivated by larger core counts within the same node, sophisticated ways of handling shared memory parallelism are becoming increasingly attractive to reduce load imbalance and thus improve parallel efficiency in large shared-memory multi-core configurations [16, 28, 35]. For example, OpenMP, the most popular approach for shared memory programming, has significantly evolved and currently incorporates advanced features such as tasking support [4, 39]. For all these reasons, parallel operations such as scheduling and synchronization are expected to become key system software components. As a result, simulators targeting next-generation HPC systems must take into account such parallel operations performed at the runtime system level.

Existing tools make simulation of large-scale HPC machines with thousands of cores unfeasible. Conventional cycle-accurate architectural simulators offer a great level of detail, but make simulation times impractical when using more than a few tens [6, 7, 51] or a few hundreds of cores [45]. Higher-level simulators are able to simulate thousands of cores at the cost of not modelling any microarchitectural details or the impact of the system software [2, 14, 55]. Raising the level of abstraction is necessary, but needs to be done to an appropriate degree. Hence, it is critical to develop flexible simulation infrastructures that allow to quickly trim the vast design space while still capturing the impact of the simulated microarchitecture and system software.

In this paper we make the following contributions:

- We present MUSA, a multi-scale simulation approach that enables fast and accurate performance estimations of next-generation HPC machines. Our methodology seamlessly captures inter-node communication as well as intra-node microarchitectural and system software interactions, improving usability and simplifying the simulation workflow. MUSA relies on native execution traces with two levels of detail to allow simulation of different communication networks, numbers of cores per node, and relevant microarchitectural parameters.
- We validate MUSA using the NAS Multi-Zone Parallel Benchmark suite [27], and then evaluate three large-scale case studies (with up to 16,384 cores) using BT-MZ, HYDRO [33], and SPECFEM3D [31]. Our evaluation shows that MUSA provides accurate performance predictions by combining information at different levels of granularity. When comparing native executions and MUSA simulations with up to 2,048 cores, we achieve relative errors within 10% in the common case, demonstrating that our detailed model is able to capture microarchitectural and system software effects. In addition, we show that our simulations complete in an affordable amount of

[1] The last two authors have equally lead the research effort that produced this paper.

time, i.e. less than a day of total aggregated CPU time for detailed 16,384-core simulations. This allows to quickly identify scalability problems in the targeted case studies.
- Finally, we perform a design space exploration analysis using high-performance, low-power, and die-stacked DRAM processor profiles on a system with 16,384 cores. We find that for one of the evaluated HPC applications, HYDRO, the low-power processor can achieve on par performance even with the same number of cores, as the high-performance memory hierarchy and aggressive microarchitecture are over-dimensioned. In contrast, the other two applications benefit from an aggressive out-of-order microarchitecture, and SPECFEM3D achieves better scalability by exploiting the higher memory bandwidth provided by die-stacked DRAM technology.

II. Background and Motivation

This section describes the co-design challenges in next-generation HPC systems. Afterwards, we discuss the difficulties of simulating large HPC applications and the limitations this imposes in exploring designs for future systems.

A. Co-Design of HPC Applications and Systems

In current HPC applications, the Message Passing Interface (MPI) is the most common way to expose parallelism across multiple computing nodes. As the number of nodes increases with the deployment of new HPC systems, node-to-node communication costs become more relevant and need further consideration when designing such systems. For example, certain applications might experience communication time overheads in the presence of load imbalance across different nodes. Finding the right ratio between the number of nodes and the number of processing units per node is a primary design decision that can greatly impact application performance. Hence, exploring such trade offs beforehand is a fundamental step when designing a new system.

In current HPC systems, nodes typically consist of a small number of sockets with shared memory. Shared-memory programming models such as OpenMP are the most common approach to express parallelism within a node. Recently, advanced tasking features or support for accelerators and SIMD constructs have been included in OpenMP. These features allow to exploit the computational power of the node while increasing programmer productivity [4, 16, 39]. In next-generation HPC systems, an appropriate amount of cache per core and enough memory bandwidth are paramount to achieve the desired performance within a node when running one of the targeted applications. Therefore, provisioning a node with enough resources to fit such demands is a design decision that needs to be considered when designing an HPC system.

Hybrid programming models simultaneously employ different paradigms to exploit both inter- and intra-node parallelism, e.g. MPI and OpenMP. To achieve peak performance it is important to have an even amount of computation distributed across the different nodes, and that the available parallelism within a node maps well to the available resources. By properly dimensioning a system the node-to-node communication overheads can be minimized, while at the same time achieving the desired node level performance.

B. Challenges Simulating Large HPC Applications

Simulation is a key tool in order to design next-generation HPC systems and applications. However, simulating future HPC systems at a meaningful scale is challenging due to the large amount of components that need to be considered. Consenquently, HPC system designers have to constantly trade off accuracy for simulation speed. As explained before, the number of nodes in the system and the amount of resources within a node can create performance bottlenecks at the inter-node and intra-node levels. Hence, scaling down the simulated system or focusing only on the node level may lead to suboptimal design decisions. Moreover, applications used in large-scale systems exhibit long execution times and downsizing the input sets to make them more manageable can change the application's characteristics, i.e. the amount of cache or memory bandwidth needed to perform well under the original input sets.

In order to simulate such large HPC systems, new methodologies are needed to gauge the necessary requirements both at the overall inter-node level as well as the intra-node level. In this paper we propose MUSA, a multi-level simulation infrastructure capable of simulating large-scale HPC systems. MUSA combines different levels of abstraction to provide insights on the expected performance of an application on a hypothetical HPC system. The following section describes the proposed infrastructure in detail.

III. Multi-Level Simulation Approach

In this section, we present MUSA, our multi-level simulation infrastructure for hybrid programs running on next-generation HPC systems.

A. MUSA - General Overview

MUSA is an end-to-end methodology that uses traces to enable large-scale simulations with different communication networks, numbers of cores per node, and microarchitectural parameters in a comprehensive HPC environment that considers the effects of system software. To this end, MUSA employs two components: (i) a tracing infrastructure that captures communication, computation and runtime system events; and (ii) a simulation infrastructure that leverages these traces for simulation at multiple levels. Figure 1 illustrates our modular methodology that provides a streamlined workflow from tracing to the final simulation output.

HPC applications stress a system at multiple levels, including both the hardware (i.e. pipeline, core, chip, node, network) and the software (i.e. scheduling, synchronization, communication and computation phases). Using a single simulation approach across all levels would be too rigid to adapt to the degree of detail appropriate for each level. For this reason, MUSA's simulation infrastructure is capable of changing the

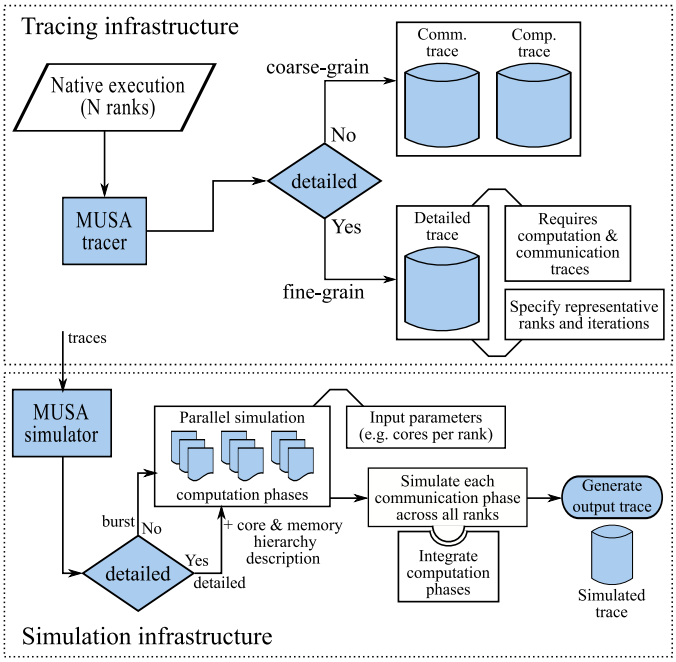


Fig. 1. MUSA tracing and simulation methodology.

level of simulation detail, from cycle-accurate microarchitectural simulations to high-level analytical models. The methodology allows to combine detailed (higher computational cost) and high-level (higher simulation speed) simulations, enabling simulation of large-scale machines with thousands of cores in a reasonable amount of computational time, while guaranteeing a high degree of accuracy. The rest of this section provides further details on the tracing and simulation infrastructures.

B. Tracing - Capture Multi-Level Behavior

The initial step is to trace an application's execution at multiple levels. Given our targeted hybrid programming model, we start tracing each MPI process representing a *rank*. Within a rank multiple threads running in parallel may coexist, managed by a runtime system. As shown in Figure 1, MUSA traces an application by running it natively with the number of ranks to be used in future simulations and instructs the runtime system to execute each rank using a single thread.

The tracer then generates a file with the communication and computation information per rank. This trace file contains information about the MPI communication phases, including: (i) timestamps of beginning and end of each communication phase for all ranks, (ii) the type of communication (e.g. collective or point-to-point), and (iii) the size of the data to be sent. At the same time the computation information for each rank is recorded, storing timestamps for each computation phase and multiple runtime events such as creation and synchronization of parallel sections. The instrumentation required to obtain these traces is coarse-grained, leading to a small overhead that does not significantly affect the application's behavior.

In order to simulate a node in detail, MUSA requires additional instruction-level instrumentation for computational phases; such as the operation code, the program counter and the involved registers and memory addresses. Such detailed instrumentation is deferred to a separate native execution due to its higher overhead that might alter application behavior. Hence, when tracing in detailed mode, the timestamps taken in the first trace are used to correct any deviation in the behavior of the application introduced in the detailed trace step.

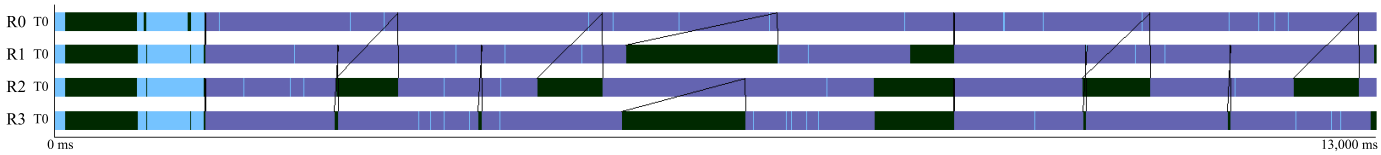
Figure 2a shows a trace generated by MUSA's tracing infrastructure with communication and computation information for a fraction of an application's execution time. The tracing methodology generates traces that allow simulations even if the characteristics of the simulated computational node (e.g. the number of cores, the memory hierarchy) or the communication network change. As a result, we can perform architectural analysis of a large design space using the same set of traces, reducing trace generation time and storage requirements. Section IV-C contains further details on the employed tracing tools and their overheads.

C. Simulation - Leverage Multi-level Traces

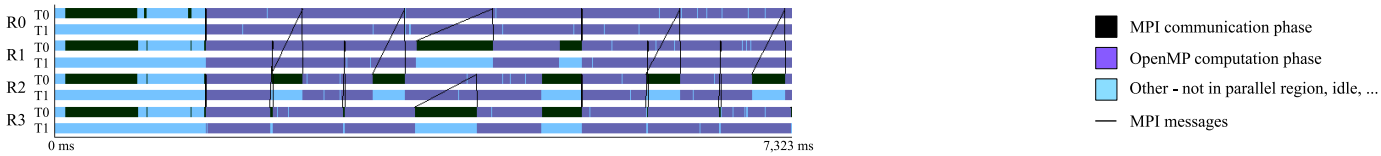
MUSA's simulation step employs the communication and computation events gathered in the tracing step. As shown in Figure 1, the methodology initially identifies the different computation phases for each rank, which are independent and can be simulated in parallel. Each of these rank level computation phases is simulated with the specified number of cores and parameters of the microarchitecture and the memory hierarchy. However, MUSA is able to simulate an arbitrary number of cores per rank. To accomplish this, MUSA injects runtime system API calls by using the runtime system events recorded in the trace, effectively simulating the runtime system, including scheduling and synchronization for the number of simulated cores. The architectural simulator we employ can perform simulations either in *burst* or *detailed* mode, which allow from faster than native simulation speeds to slower but more detailed design space exploration studies, respectively. Details about the chosen architectural and network simulators can be found in Section IV-C.

Burst mode simulation: Simulations using burst mode replay the computation events traced during native execution with coarse grained instrumentation. Burst mode simulations do not take into account the contention that the memory hierarchy of a node might experience when running multiple threads, hence the obtained performance estimations are to be treated as upper bounds. However, MUSA allows the user to specify IPC correction factors to account for the impact of inter-thread contention if there is any application-specific knowledge, making burst simulations more accurate and flexible. Burst mode simulations allow faster than native simulation speeds, thus enabling quick design space exploration studies with a variable number of cores per rank and different communication networks.

Detailed mode simulation: When simulating a computation phase in detail, MUSA also uses the detailed traces, enabling cycle-accurate simulations with detailed models for microarchitecture and memory hierarchy. The detailed information in the instruction-level trace allows to use different cycle-accurate



(a) Coarse-grain instrumentation trace of HYDRO with 4 ranks. Delimits computation and communication phases and includes runtime events.



(b) Simulation output trace for the above input trace when simulating a system with 2 cores per rank. The runtime system is faithfully modeled.

Fig. 2. Traces used in MUSA's methodology: (a) tracing infrastructure and (b) simulation infrastructure output. Traces are shown using the same time scale.

simulators, ranging from component-specific simulators, such as main memory, cache hierarchy, or interconnects, to detailed pipeline microarchitecture simulators. Detailed simulations can be time consuming and an appropriate simulator has to be chosen depending on the envisioned target study.

Simulating all computation phases of an application in detail is feasible for small systems and short execution times. However, when going into the domain of thousands of cores, full detailed simulation becomes prohibitive both in terms of trace size and simulation time. Fortunately, HPC applications follow certain execution patterns that are easy to identify with our visual trace format. We can leverage this information by specifying a subset of the ranks, or even a subset of the iteration phases within a rank, to be traced and simulated in detailed mode. Therefore, MUSA allows the user to define such bounds as input parameters, giving great flexibility in deciding which computation phases are to be simulated in detail, while the performance of the remaining phases is extrapolated. Section III-D details how MUSA performs sampling of computation phases at different levels.

Network simulation and final output: After the computation phases have been simulated, MUSA replays the execution of the communication trace events in order to simulate the communication network and generate the final output trace of the simulation. During this process, the durations of the computation phases are replaced by the results obtained in the simulations (either in burst or detailed mode), and the communication phases are simulated using a network simulator. At the end of this process the entire simulation is complete and the output trace is generated for visualization.

Figure 2b shows an output simulation trace generated by MUSA when simulating two cores per rank. The simulation models the OpenMP scheduling events by calling the actual runtime system through inserted API calls for the traced events, faithfully modeling the impact of having two cores on each node. The MPI communication is processed by thread *T0* on each rank, while the computation phase load of each rank is distributed across the two cores.

D. Sampling - Reducing Simulation Time

Accurate microarchitectural simulation is time consuming. Conventional simulators achieve simulation speeds of 100 to 1000 KIPS [6, 42, 45]. As a consequence, detailed simulation of large systems or long-running applications becomes infeasible. While MUSA allows simulations at different levels of detail, it still requires to simulate some computation phases in detail. In an HPC application, these phases typically run for a few seconds, before starting a new communication phase.

A common technique for reducing simulation time is sampling. Sampling can be employed to allow detailed simulation of larger portions of an application or to further reduce simulation time. Sampling seeks to minimize the amount of detailed simulation by only simulating the representative parts of an application. In the following, we point out how MUSA employs sampling at three orthogonal levels of granularity in an application, namely (i) the whole application, (ii) a single MPI rank, and (iii) a computation phase within an MPI rank.

Application level: Many applications in HPC show iterative behavior, with each iteration representing a step in time or space. In many cases, different iterations show very similar performance. Automatic techniques to identify iterations based on performance monitoring counters or traces of logical events have been proposed in the past [11, 26]. However, the simplest approach relies on directly analyzing the code of the application, annotating the start and end of an iteration. When sampling at the application level, MUSA leverages these techniques to identify repetitive behavior and select a small number of iterations for detailed simulation.

MPI rank level: As described in Section II, a common programming technique in HPC applications is the division of the problem domain into *blocks*. Afterwards, each block is processed by a different MPI rank. Often, different MPI ranks show similar performance across all processes. Consequently, MUSA can select a subset of the MPI ranks for detailed simulation at the microarchitecture level. MUSA adopts a simple approach consisting in simulating one out of every N MPI ranks (periodic sampling). There are existing techniques to automatically select representative computation phases of an application [22, 46].

Computation phase level: After selecting a subset of iterations and MPI ranks, all computation phases have to be simulated in detail. Identifying representative sections of a computation phase can be done automatically [47, 53],

TABLE I
APPLICATION CHARACTERISTICS AND TRACING STATISTICS.

Benchmark		Characteristics				Tracing		
Name	Input	Ranks	Tasks/rank	Iterations	Regions/iteration	Overhead	Burst Trace	Detailed Trace
BT-MZ	Class D	16	2.3M	250	1	3.4%	5.6 GB	53.3 GB
SP-MZ	Class D	16	131K	500	1	1.2%	0.4 GB	13.7 GB
LU-MZ	Class D	16	1.3M	300	1	1.0%	3.2 GB	12.5 GB
HYDRO	big	256	1.0M	200	2	6.0%	16.1 GB	16.9 GB
BT-MZ	Class E	256	1.3M	250	1	8.5%	57.4 GB	120.0 GB
SPECFEM3D	n/a	256	1.9M	10700	1	9.3%	101.4 GB	106.4 GB

and applied to parallel applications with barriers [9], as is the case of typical OpenMP programs with parallel loops. In the case of task-based programs, a computation phase typically comprises several thousands of task instances. These task instances stem from task types which are instantiated many times during the execution of an application. Different instances of the same task type show similar behavior, which allows to periodically sample task instances to be simulated in detail [23]. Then, these results are extrapolated to the rest of the computation phase by fast-forwarding, using the IPC of the representatives simulated in detail [9, 23]. Simulating a reduced set of instructions shows very good accuracy while it reduces simulation time by one or two orders of magnitude in large multi-core systems. MUSA can exploit these sampling techniques to accelerate the simulation of a single computation phase within an MPI rank.

IV. EXPERIMENTAL METHODOLOGY

This section introduces the experimental methodology employed to evaluate our multi-level simulation infrastructure.

A. Applications

To validate MUSA we use the NAS multi-zone benchmarks [15]: BT-MZ, SP-MZ and LU-MZ. These representative HPC benchmarks have been designed to exploit multiple levels of parallelism using a hybrid programming model. For this validation step we use 16 MPI ranks with a mapping of one rank per node. Simulations are performed with 1 to 8 cores per node. We run the benchmarks with the input class D, for which we observe enough parallelism for the 16 MPI ranks employed.

In order to illustrate the potential of MUSA, we evaluate large-scale machines using HYDRO [33], BT-MZ with the large input class E, and SPECFEM3D [31]. HYDRO implements a simplified version of RAMSES [49], a code developed to study large-scale structure and galaxy formation. It uses a fixed rectangular two-dimensional space domain and solves the compressible Euler equations of hydrodynamics. Fluxes at the interface of two neighbouring computational cells are computed using a Riemann solver. SPECFEM3D uses the continuous Galerkin spectral-element method to simulate forward and adjoint seismic wave propagation on arbitrary unstructured hexahedral meshes. For these applications we employ 256 MPI ranks, one per node, and up to 64 cores per node, resulting in simulations of up to 16,384 cores.

All applications use a hybrid programming model based on MPI [38], and a task-based programming model, OmpSs [16]. OmpSs allows to annotate tasks with data inputs and outputs. Using this information, the OmpSs runtime system schedules task instances taking data dependencies into account and performs synchronization only when necessary. These OmpSs features were included into the specifications of OpenMP 3.0 and 4.0 [4, 39]. The whole OmpSs environment is available as open source. Applications are compiled with GCC 4.7.2 and the -O3 optimization flag set.

Table I summarizes the main characteristics of each application. It includes the number of MPI ranks, the total number of tasks per MPI rank, the number of iterations of the application and the number of parallel regions within an iteration. For example, in the case of BT-MZ with input class E there is an average of 5,200 tasks per parallel region ($\frac{tasks/rank}{iterations \times regions}$).

B. Native HPC Infrastructure

We validate MUSA against the MareNostrum 3 supercomputer. Each node has two sockets with an Intel Xeon E5-2670 featuring eight cores running at 2.6GHz. The cores implement aggressive superscalar capabilities, have private L1 and L2 caches, and a shared 20MB L3 cache. The nodes are connected via a high-bandwidth Infiniband FDR10 network. To validate MUSA, we simulate the same HPC infrastructure.

For the native executions, we present results with up to eight cores per node, making use of a single socket. This avoids factoring in non-uniform memory access timings that may bias the results. In addition, we run each native experiment five times and select the measurement that presents the lowest amount of interference due to current system load.

C. Tracing and Simulation Infrastructure

Traces are obtained using different lightweight tracing tools based on *extrae* [19] and *PIN* [36]. To obtain the traces for an application, we instrument a native execution that runs only a single thread per node, i.e. per MPI rank. Extrae generates the high-level trace (*burst trace*) using coarse-grain instrumentation. The tracer instruments the entire application, i.e. all ranks and iterations. However, for the detailed trace, such an approach would be impractical and require too much storage. For the evaluated set of applications, we observe that tracing the second iteration of a single MPI rank is enough to later reconstruct an application's entire execution using this

information and the burst trace. This allows for manageable tracing times and storage requirements.

Table I details the overhead of generating traces at burst level, and the sizes of the burst and detailed traces for each application. The overheads include the trace disk I/O costs, which actually do not affect the application behavior, as I/O is performed at points where the application is halted by the tracer. In terms of trace sizes, burst traces are relatively small, while covering the entire execution of applications running for several minutes on the real machine. On the other hand, detailed traces are bigger, even though they only cover the second iteration of a single MPI rank. Note that a detailed trace for the entire BT-MZ application with input class D would require more than 200 terabytes of storage. The obtained detailed traces are manageable while still allowing MUSA to perform meaningful detailed microarchitectural simulations.

Our methodology requires both an architectural and a communication network simulator. To simulate the computation phases we use *TaskSim*, a detailed multi-core simulator with two operation modes, a fast exploration mode based on precalculated computation phase execution times (*burst*) and a *detailed* mode with accurate microarchitecture and memory models [42, 43]. For the network we employ *Dimemas*, which is able to model MPI communication primitives using analytical models [20]. However, we strongly believe that the MUSA methodology can be applied to nearly any simulator currently available in the community.

V. Evaluation

In this section, we first validate the proposed simulation infrastructure. Afterwards, we apply our methodology to detect scalability bottlenecks in hybrid applications both at the algorithmic level, due to the lack of parallelism, and at the hardware level, due to contention on shared resources. Finally, we also present simulation time results and a design space exploration analysis.

A. Validation

We validate MUSA by performing several experiments with the NAS Multi-Zone benchmarks. As described in Section IV, all validation experiments are done with 16 MPI ranks and the class D input set, always assuming a single MPI rank per node. Figure 3 shows the speedup for a single iteration (Figure 3a), and for the entire application (Figure 3b) when increasing the number of cores per MPI rank. Having both figures is very useful, as the overall execution time of the whole application or a single iteration can be biased by the sequential execution of a particular phase of the application, such as reading input files, initializing data structures or writing output files.

Native executions are performed with up to eight cores per rank, as this is the number of cores per socket on the available machine. Consequently, in our validation we use up to 128 cores, with parallel efficiencies that range from 48% (LU-MZ) to 92% (BT-MZ). Using a performance visualization tool, we observe that in all benchmarks the first iteration is less representative than the others. We therefore chose to trace the second iteration in detail to avoid capturing the impact of cold hardware structures in the processor. Figure 3 shows that scalability in native and simulated executions closely match when comparing a single iteration and the entire application.

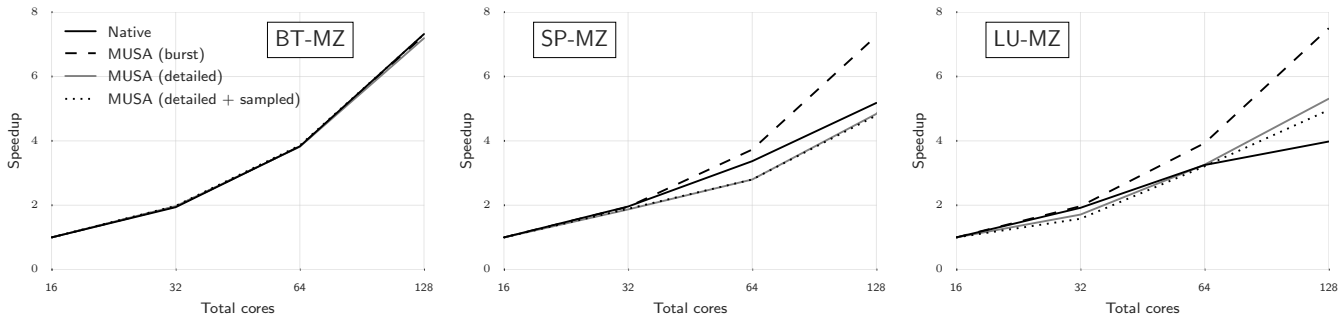
First, we evaluate the accuracy of MUSA with burst simulations, denoted *MUSA (burst)* in the figure. A first observation is that burst simulations accurately model the system for BT-MZ, with negligible relative errors. This is due to the fact that BT-MZ is compute bound and contention on shared resources does not increase significantly with larger core counts, leading to a speedup of 7.3× on an 8 core node. However, SP-MZ and LU-MZ have higher memory contention and performance predictions start to differ from the native execution as the number of cores per node increases. For SP-LU and LU-MZ, *MUSA (burst)* predicts speedups of 6.9× and 7.5× with relative errors of 33% and 88% with respect to native runs.

The results obtained in burst simulation clearly indicate that, as we scale the number of cores in the system, cycle-accurate memory simulations are necessary to capture contention on shared resources. We perform a second set of simulations with MUSA using detailed microarchitectural and memory models, denoted *MUSA (detailed)* in the figure. In this case, MUSA simulates one iteration of a single MPI rank and extrapolates the results to the remaining MPI ranks and iterations.

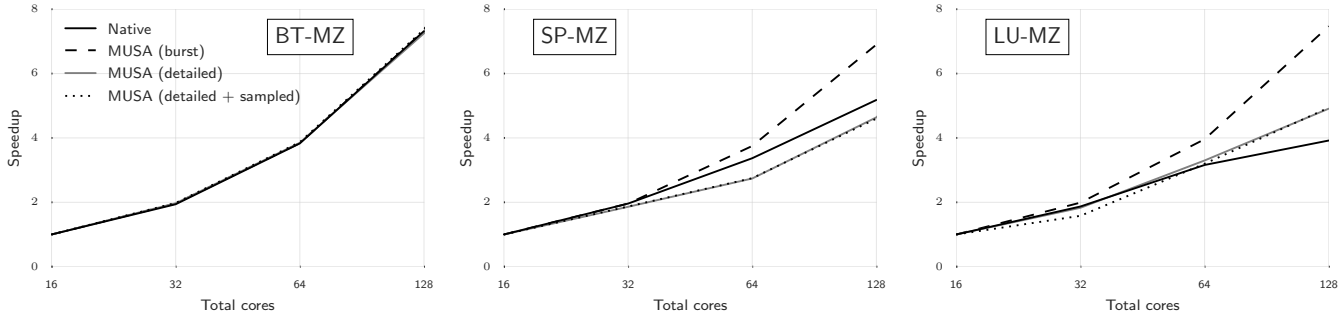
MUSA (detailed) improves accuracy with respect to *MUSA (burst)* for both SP-MZ and LU-MZ when simulating a system with 128 cores. In the case of SP-MZ, the relative error is reduced from 33% to 10%, capturing the trend observed in native execution. For LU-MZ the error is reduced from 88% to 25%. However, the trend is not captured as accurately as in the other two benchmarks due to modeling inaccuracies in the simulated DRAM subsystem. LU-MZ has poor row-buffer locality and internal bank conflicts, and thus needs a detailed component-specific simulator to capture these behavior. Therefore, for this application we would suggest to use tools like DRAMSim2 [44] or Ramulator [30]. In the case of BT-MZ, the error is negligible as happens in the burst simulation and, as expected, the performance is again accurately predicted.

Next, we evaluate the accuracy of MUSA using TaskPoint [23] to speed up detailed simulation, denoted *MUSA (detailed+sampling)* in the figure. In this case, we only perform detailed microarchitectural simulation on a fraction of the task instances of the application. We apply TaskPoint's default parameters: first, we simulate 2 task instances in each thread in order to warm up microarchitectural state. Afterwards, we simulate a total of 4 task instances of each task type as samples. This reduces the total simulation time by a factor of 2.5× in BT-MZ, 1.9× in SP-MZ, and 3.0× in LU-MZ. As shown in Figure 3b, *MUSA (detailed+sampling)* predicts nearly the same speedups as *MUSA (detailed)*. The average difference between these approaches is less than 3%. These results are consistent with previously published results with TaskPoint [23].

Our validation shows that MUSA provides accurate performance predictions by combining information at different levels of granularity. When comparing native executions of



(a) A single iteration of the benchmark



(b) Entire execution of the benchmark

Fig. 3. MUSA validation using the NAS Multi-Zone Parallel Benchmarks: BT-MZ (left), SP-MZ (middle) and LU-MZ (right). Benchmarks are run natively and simulated using MUSA with 16 MPI ranks and up to eight cores per node.

the entire application with MUSA simulations, we can see that the relative errors are low and that the detailed models are able to capture microarchitectural details such as memory contention. In addition, we can do this in an affordable amount of time, as even detailed simulations complete within a few hours. A more comprehensive study in terms of simulation time is shown for our large-scale simulations in Section V-C.

B. Large-scale Simulations

We present large-scale simulations of BT-MZ with input class E, HYDRO and SPECFEM3D for the entire application. Table I lists the relevant application characteristics. We employ 256 MPI ranks, one per node, with up to 8 cores per node (2,048 cores) for native executions and up to 64 cores for simulations with MUSA (16,386 cores). These simulations allow us to identify scalability bottlenecks occurring for large core counts per node, a trend that continues to manifest.

Figure 4a shows speedup estimations for BT-MZ. Results with up to 8 cores per node (2,048 total) are validated against the native execution of the application, showing a good level of accuracy. With 8 cores per node, the parallel efficiency reaches 82% for the overall execution of the native application, and MUSA predicts the parallel efficiency with an error of less than 5% for all simulation modes.

When performing burst simulations with larger core counts, the parallel efficiency significantly degrades, reaching 26% for 64 cores (16× speedup). We analyze if task management is the limiting factor to scalability. To this end, we run the master thread with a significantly higher speed and observe no significant change in scalability. From this experiment we conclude that BT-MZ does not expose sufficient task parallelism to achieve a higher parallel efficiency at large core counts. One possible solution is to reduce task granularity and thus increase the number of task instances. As this approach also increases the task management overhead, it poses an interesting optimization problem. MUSA predicts similar scalability trends with all simulation modes because this application is not memory intensive, as stated in the previous subsection.

In conclusion, we identify that BT-MZ lacks task parallelism and thus shows limited scalability in executions with larger core counts per MPI rank. Scalability can be improved by reducing task granularity, but only if this does not increase the effort of task management to a point where it becomes the new limiting factor to scalability.

Figure 4b shows speedup estimations for HYDRO. Results with up to 8 cores per node (2,048 total) are validated against the native execution of the application. For up to 8 cores, detailed simulation modes predict parallel efficiency with an error of less than 8%. For higher core counts, all simulation modes predict similar results. We attribute this to HYDRO's low memory intensity.

As we increase the number of cores, parallel efficiency significantly degrades, reaching a value of only 17% at 64 cores per node. A significant percentage of parallel efficiency is lost due to communication (MPI) overheads. We find the parallel efficiency of the computation phases to be 31% when communication is ignored. Therefore, the computational part

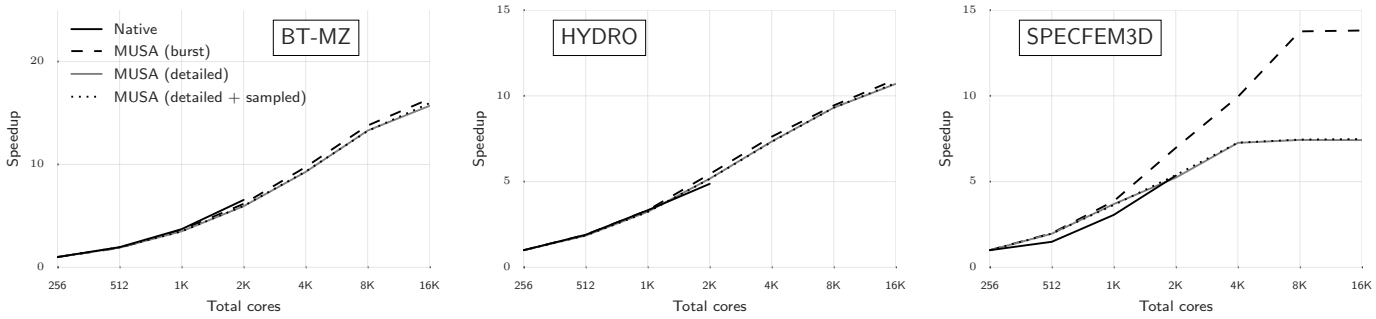


Fig. 4. Performance estimations of (a) BT-MZ with input class E, (b) HYDRO, and (c) SPECFEM3D for the entire application on 256 MPI ranks. Native runs with up to 8 cores per node (2,048 cores), and simulated runs with MUSA on up to 64 cores per node (16,384 cores).

of the application has room for improvement. With the help of conventional performance analysis tools for MPI applications, we observe that the sequential part in each iteration is limiting the scalability of the application for core counts larger than 8. To avoid this limitation, the application needs to be restructured to reduce the amount of sequential computation.

Furthermore, for 32 and 64 cores per node the time devoted to task creation and scheduling limits the scalability of the application. There are multiple solutions to alleviate this problem. The first solution consists in increasing the granularity of the executed tasks, as this reduces the total number of task instances and thus the management effort. A second option is having multiple threads creating and scheduling tasks using nested parallelism. Finally, a third alternative consists in using hardware support for the runtime system [18].

Figure 4c shows speedup estimations for SPECFEM3D. Results for up to 8 cores per node (2,048 total) are compared to the native execution of the application. For 2 and 4 cores per node, we observe notable relative errors when comparing MUSA simulation modes and native execution. However, for 8 cores per node the detailed simulation modes predict parallel efficiency with an error of less than 3%. In addition, we observe that for core counts per node of 8 and more, performance estimations with burst and detailed mode differ significantly due to increasing off-chip memory contention, leading to performance overestimations in burst mode.

As we increase the core count in burst simulation mode, we observe that the application's scalability suddenly saturates from 32 to 64 cores per node. We find that this is because the number of task instances for this application is small, less than 200 per parallel region (see Table I). Moreover, there are several task types that feature significantly different execution times, which eventually leads to severe load imbalance, limiting scalability. Since MUSA faithfully models task scheduling in burst mode, we correctly identify this bottleneck.

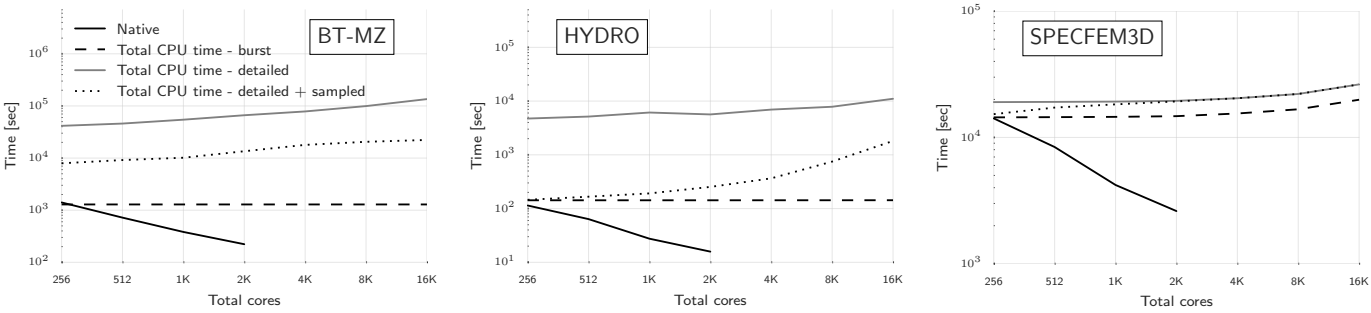
However, for detailed simulations we see that the performance actually saturates when moving from 16 to 32 cores per node. This is due to the combined effect of load imbalance and significant off-chip memory contention, which especially penalizes long running tasks that now execute for an even longer period of time, exacerbating load imbalance. With MUSA we are able to identify a bottleneck that manifests due to the combination of two factors, and gain insight on the performance penalty each factor imposes.

C. Simulation Time Cost Analysis

Figure 5a shows the time required to run native and simulated executions for BT-MZ (input class E) with 256 ranks. We plot time-to-solution for native executions and total aggregated CPU time for simulated runs with MUSA. The total CPU time required for simulations in burst mode is nearly constant and comparable to the native execution with 1 thread per rank, as it uses pre-calculated task execution times. Speedup of sampled over detailed simulation remains constant, providing around one order of magnitude simulation time improvements. A sampled simulation for 16,384 cores requires less than 6 hours of total CPU time, while the native execution for 1 thread per rank takes about 24 minutes - only one order of magnitude of slowdown, even when considering sequential simulation.

Figure 5b shows the same data for HYDRO with 256 ranks. Again, the simulation time in burst mode is nearly independent from the number of simulated cores. A detailed simulation of HYDRO on 16,384 cores requires less than 3 hours. This time is reduced to less than an hour when performing sampled simulation. We observe that the speedup of sampled over detailed simulation decreases with increasing core counts. HYDRO has two computation phases per iteration. Therefore, architectural warmup and measuring of samples is performed twice per iteration. In addition, the number of tasks per computational phase is lower than in the case of BT-MZ. Both aforementioned effects hinder effective simulation sampling.

Figure 5c shows similar data for SPECFEM3D with 256 ranks. In this case we see that burst and detailed executions take a similar amount of time. This is because this application has a large number of iterations (i.e. 10,700). However, only one iteration is simulated in detailed mode. As a consequence, the time it takes to simulate the burst trace for the entire application is similar. Also note that sampling is not effective and its simulation time eventually converges to the detailed simulation time. The number of tasks per computational phase is so small that all of them are simulated in detail as samples.



(a) (b) (c)

Fig. 5. Total aggregated CPU time for MUSA simulations versus time-to-solution for native executions.

TABLE II
ARCHITECTURAL PARAMETERS OF HIGH-PERFORMANCE, LOW-POWER AND DIE-STACKED DRAM CONFIGURATIONS FOR A 64 CORE PROCESSOR.

Parameter	High-perf.	Low-power	Stacked DRAM
ROB	168 entries	40 entries	72 entries
Issue width	1/2/4	1/2/4	1/2/4
L1 cache	32KB private 4 cycles 8-way	32KB private 4 cycles 2-way	32KB private 4 cycles 8-way
L2 cache	256KB private 11 cycles 8-way	8MB shared 21 cycles 16-way	32MB shared 16 cycles 16-way
L3 cache	128MB shared 28 cycles 20-way	none	none
DRAM	off-chip 4 channels DDR3-1600	off-chip 3 channels DDR3-1600	die-stacked 8 channels DDR3-3200

For 16,384 cores detailed simulation and native execution with 1 thread require 7.3 hours and 5.6 hours, respectively.

D. Design Space Exploration

We demonstrate the usefulnes of the MUSA infrastructure by performing a design space exploration study. Prior simulations focused on increasing the core count per node while leaving microarchitectural and memory parameters unchanged. Given that the trend to use commodity server processors is starting to change and that new technologies like die-stacked DRAM start to be available [48], we show how MUSA can aid to explore this vast design space with simulations using 16,384 cores - i.e. 256 MPI ranks and 64 cores per node - on BT-MZ with input class E, HYDRO and SPECFEM3D.

With this objective, we study the performance of these applications on three different multi-core architectures. The first system resembles a high-end server-class processor with a large reorder buffer and a three-level cache hierarchy, as found in traditional HPC environments. The second configuration is inspired by a low-power mobile platform. It has a smaller reorder buffer and only two levels of cache, as is typical for battery-powered mobile systems. The third configuration represents an emerging many-core chip with die-stacked DRAM, featuring medium cores and moderate LLC capacity, but lower latency and higher bandwidth access to DRAM. Table II lists the key characteristics of the simulated architectures.

Figure 6 shows the predicted performance on these platforms for different issue width values of 1, 2, and 4 instructions per cycle. The reported speedup is normalized to an execution with one thread per rank using the high-performance configuration. The evaluated applications show very different behavior. BT-MZ benefits from running on a high-performance processor, achieving more than 35% additional performance compared to the speedup of the low-power processor for an issue width of 4. This compute intensive application favors the combination of a large reorder buffer with quad-issue width, which also outperforms the die-stacked DRAM configuration that has a medium sized reorder buffer. A final observation is that, for the low-power configuration, increasing the issue width from 2 to 4 improves performance by merely 6%, while significantly increasing the complexity of the core.

In contrast, HYDRO shows a completely different behavior. The speedup achieved by the low-power processor nearly matches the speedup of the high-performance and die-stacked DRAM configurations. Since HYDRO has low memory intensity, deep cache hierarchies or low-latency and high-bandwidth DRAM memory does not improve performance significantly. Moreover, as explained in Section V-B; existing factors that limit the scalability of the application, such as communication overheads and sequential code, hinder the performance of the aggressive cores. Furthermore, HYDRO benefits much less from an increased issue width - performance improves by less than 25% when increasing the issue width from 1 to 4 instructions per cycle. Thus, we conclude that the much simpler low-power architecture can deliver competitive performance for HYDRO.

Finally, for SPECFEM3D we observe that for issue widths of 2 and 4, the low-power configuration falls behind due to a less performing memory hierarchy. This application has a significant degree of memory contention. For this reason, the die-stacked DRAM configuration is able to outperform the high-performance configuration even though it features a less agressive core. However, the gains are not as significant as one might expect. This is due to the fact that the performance is limited by severe load imbalance at the node level due to

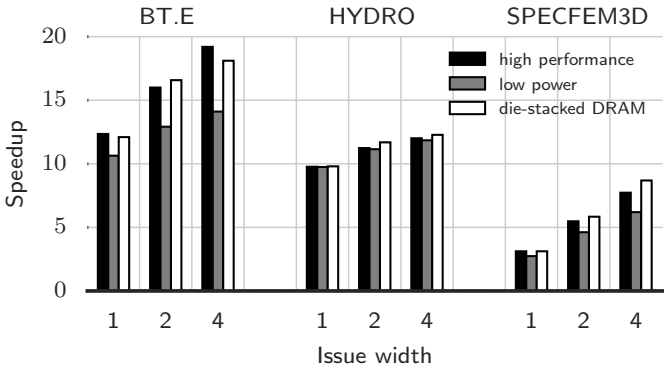


Fig. 6. Design space exploration of BT-MZ, HYDRO and SPECFEM3D for different issue widths and processor profiles for 256 MPI ranks (16,384 cores).

the small number of tasks per parallel region, as explained in Section V-B. Nontheless, we conclude that die-stacked DRAM is beneficial over an agressive core design for SPECFEM3D.

VI. Related Work

In this section, we review prior work on simulation of both shared and distributed memory machines as well as techniques to speed up simulation of parallel applications.

Simulating distributed machines: Prior work proposed simulation methodologies to evaluate the performance of large-scale parallel applications. Some proposals also employ a multi-level approach, combining different simulation layers. However, only a few evaluate scenarios with thousands of cores, but at the cost of not modeling microarchitectural details or system software interaction [14, 24, 55]. The other proposals evaluate lower core counts [1], while also lacking important features, e.g. detailed microarchitectural simulation [10], or support to capture operating system or runtime system interactions [21]. Finally, in other infrastructures each simulation requires a large computational effort due to the use of full system simulation [25] or the lack of sampling techniques [34], making them impractical for large-scale studies.

The usefulness of parametric models based on basic machine performance metrics and application characteristics has also been explored [5, 29]. These models are applied to understand the performance of current systems, to unveil bottlenecks, and to show where tuning efforts can be useful, but are tailored to specific applications.

Simulating shared-memory systems: Most simulation infrastructures at this level tend to be cycle-accurate to faithfully model the processing cores and the memory hierarchy. However, this level of detail comes at a significant slowdown, making simulations with more than a few tens or hundreds of cores impractical [6, 7, 45].

Sampling techniques: To reduce simulation time, statistical sampling is applied to identify a representative section of an application or even a synthetic trace, much shorter than the original one [8, 17, 32, 46, 53]. This representative section is then executed in a cycle-accurate simulator. However, the accuracy of these simulations is tied to the quality of the selected representative section of the application.

Finally, to further reduce simulation time and allow the simulation of larger multi-core processors, parallel simulators have been proposed [3, 7, 13, 37, 41, 45]. The main drawback of these proposals lies in the synchronization overhead. This overhead can be reduced at the expense of sacrificing accuracy in the final results of the simulation.

VII. Conclusions

In this paper we have introduced MUSA, a multi-level simulation approach that enables fast and accurate performance estimations of large-scale next-generation HPC machines. MUSA can model microarchitectural and runtime system effects by leveraging multi-level traces. These traces also allow for different simulation modes and execution replay to quickly extrapolate results of entire hybrid applications running on tens of thousands of cores.

MUSA has been validated using a production supercomputer with up to 2,048 cores showing high accuracy, with relative errors below 10% in the common case. For native codes that run for several minutes, MUSA allows detailed simulation of systems with more than ten thousand cores within a few hours of total aggregated CPU time. Our 16,384-core simulations revealed scalability bottlenecks in the evaluated applications that were easily identifiable using the simulation output trace and conventional performance analysis tools.

The main advantage of MUSA is that it provides results not only across known systems, but also for future systems not yet available on the market. Our design space exploration analysis provides useful insights on the different microarchitectural requirements of three applications to achieve good scalability, showing the potential MUSA offers in predicting the performance of applications on next-generation HPC machines.

Acknowledgements

This work has been supported by the Spanish Government (Severo Ochoa grants SEV2015-0493, SEV-2011-00067), by the Spanish Ministry of Science and Innovation (contracts TIN2015-65316-P), by the Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272), by the RoMoL ERC Advanced Grant (GA 321253) and the European HiPEAC Network of Excellence. The Mont-Blanc project receives funding from the EUs Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 610402 and from the EUs H2020 Framework Programme (H2020/2014-2020) under grant agreement no. 671697. M. Moretó has been partially supported by the Ministry of Economy and Competitiveness under Juan de la Cierva postdoctoral fellowship number JCI-2012-15047. M. Casas is supported by the Secretary for Universities and Research of the Ministry of Economy and Knowledge of the Government of Catalonia and the Cofund programme of the Marie Curie Actions of the 7th R&D Framework Programme of the European Union (Contract 2013 BP_B 00243). T. Grass has been partially supported by the AGAUR of the Generalitat de Catalunya (grant 2013FI B 0058).

REFERENCES

[1] V. S. Adve, R. Bagrodia, J. C. Browne, E. Deelman, A. Dube, E. N. Houstis, J. R. Rice, R. Sakellariou, D. J. Sundaram-Stukel, P. J. Teller, et al. Poems: End-to-end performance design of large parallel adaptive computational systems. *IEEE Transactions on Software Engineering*, 26(11):1027–1048, 2000.

[2] E. Anger, D. Dechev, G. Hendry, J. Wilke, and S. Yalamanchili. Application Modeling for Scalable Simulation of Massively Parallel Systems. In *17th International Conference on High Performance Computing and Communications (HPCC)*, pages 238–247, 2015.

[3] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega. Cotson: infrastructure for full system simulation. *ACM SIGOPS Operating Systems Review*, 43(1):52–61, 2009.

[4] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of OpenMP tasks. *IEEE Transactions onParallel and Distributed Systems*, 20(3):404–418, 2009.

[5] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho. Entering the petaflop era: the architecture and performance of roadrunner. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, 2008.

[6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[7] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 52:1–52:12, 2011.

[8] T. E. Carlson, W. Heirman, and L. Eeckhout. Sampled simulation of multi-threaded applications. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 2–12, 2013.

[9] T. E. Carlson, W. Heirman, K. Van Craeynest, and L. Eeckhout. Barrierpoint: Sampled simulation of multi-threaded applications. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 2–12, 2014.

[10] L. Carrington, A. Snavely, X. Gao, and N. Wolter. A performance prediction framework for scientific applications. In *Workshop on Performance Modeling and Analysis (PMA)*, pages 926–935, 2003.

[11] M. Casas, R. M. Badia, and J. Labarta. Automatic phase detection and structure extraction of mpi applications. In *International Journal of High Performance Computing Applications*, pages 335–360, 2010.

[12] M. Casas, M. Moretó, L. Alvarez, E. Castillo, D. Chasapis, T. Hayes, L. Jaulmes, O. Palomar, O. S. Unsal, A. Cristal, E. Ayguadé, J. Labarta, and M. Valero. Runtime-aware architectures. In *European Conference on Parallel Processing*, pages 16–27, 2015.

[13] J. Chen, L. K. Dabbiru, D. Wong, M. Annavaram, and M. Dubois. Adaptive and speculative slack simulations of cmps on cmps. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 523–534, 2010.

[14] W. E. Denzel, J. Li, P. Walker, and Y. Jin. A framework for end-to-end simulation of high-performance computing systems. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, pages 21:1–21:10, 2008.

[15] R. F. V. der Wijngaart and H. Jin. NAS Parallel Benchmarks, Multi-Zone Versions. Technical Report NAS-03-010, NAS, 2003.

[16] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[17] L. Eeckhout, S. Nussbaum, J. Smith, and K. D. Bosschere. Statistical simulation: adding efficiency to the computer designer's toolbox. *IEEE Micro*, 23(5):26–38, 2003.

[18] Y. Etsion, F. Cabarcas, A. Rico, A. Ramírez, R. M. Badia, E. Ayguadé, J. Labarta, and M. Valero. Task superscalar: An out-of-order task pipeline. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 89–100, 2010.

[19] Extrae: User guide manual version 3.2.1, Nov 2015.

[20] S. Girona, J. Labarta, and R. M. Badia. Validation of dimemas communication model for mpi collective operations. In *Proceedings of the 7th European PVM/MPI Users' Group Meeting*, pages 39–46, 2000.

[21] J. Gonzalez, J. Gimenez, M. Casas, M. Moreto, A. Ramirez, J. Labarta, and M. Valero. Simulating whole supercomputer applications. *IEEE Micro*, 31(3):32–45, 2011.

[22] J. Gonzalez, J. Gimenez, and J. Labarta. Automatic evaluation of the computation structure of parallel applications. In *2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 138–145, 2009.

[23] T. Grass, A. Rico, M. Casas, M. Moreto, and E. Ayguadé. Taskpoint: Sampled simulation of task-based programs. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software*, 2016.

[24] E. Grobelny, D. Bueno, I. Troxel, A. D. George, and J. S. Vetter. Fase: A framework for scalable performance prediction of hpc systems and applications. *Simulation*, 83(10):721–745, 2007.

[25] M. Hsieh, K. Pedretti, J. Meng, A. Coskun, M. Levenhagen, and A. Rodrigues. SST + gem5 = a scalable simulation infrastructure for high performance computing. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, pages 196–201, 2012.

[26] K. E. Isaacs, A. Bhatele, J. Lifflander, D. Böhme, T. Gamblin, M. Schulz, B. Hamann, and P.-T. Bremer. Recovering logical structure from charm++ event traces. In *Conference on High Performance Computing Networking, Storage and Analysis*, pages 49:1–49:12, 2015.

[27] H. Jin and R. F. V. der Wijngaart. Performance characteristics of the multi-zone NAS parallel benchmarks. *Journal of Parallel and Distributed Computing*, 66(5):674 – 685, 2006.

[28] L. V. Kalé and S. Krishnan. Charm++: A portable concurrent object oriented system based on c++. In *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 91–108, 1993.

[29] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pages 37:1–12, 2001.

[30] Y. Kim, W. Yang, and O. Mutlu. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters*, 15(1):45–49, 2016.

[31] D. Komatitsch and J. Tromp. Introduction to the spectral element method for three-dimensional seismic wave propagation. *Geophysical Journal International*, 139(3), 1999.

[32] T. Lafage and A. Seznec. Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream. *Workload Characterization of Emerging Computer Applications*, page 163, 2001.

[33] P.-F. Lavallée, G. C. de Verdière, P. Wautelet, D. Lecas, and J.-M. Dupays. Porting and optimizing hydro to new platforms and programming paradigms lessons learnt, 2012.

[34] E. A. León, R. Riesen, A. B. Maccabe, and P. G. Bridges. Instruction-level simulation of a cluster at scale. In *Conference on High Performance Computing Networking, Storage and Analysis*, pages 3:1–3:12, 2009.

[35] J. Lifflander, S. Krishnamoorthy, and L. V. Kale. Work stealing and persistence-based load balancers for iterative overdecomposed applications. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 137–148, 2012.

[36] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM Sigplan Notices*, pages 190–200, 2005.

[37] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *2010 IEEE 16th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, 2010.

[38] MPI: A Message-Passing Interface Standard. 2003.

[39] OpenMP Architecture Review Board. OpenMP: Application program interface, 2013.

[40] N. Rajovic, L. Vilanova, C. Villavieja, N. Puzovic, and A. Ramírez. The low power architecture approach towards exascale computing. *Journal of Computational Science*, 4(6):439–443, 2013.

[41] P. Ren, M. Lis, M. H. Cho, K. S. Shim, C. W. Fletcher, O. Khan, N. Zheng, and S. Devadas. Hornet: A cycle-level multicore simulator. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(6):890–903, 2012.

[42] A. Rico, F. Cabarcas, C. Villavieja, M. Pavlovic, A. Vega, Y. Etsion, A. Ramirez, and M. Valero. On the simulation of large-scale architectures using multiple application abstraction levels. In *ACM Transactions on Architecture and Code Optimization (TACO)*, volume 8, page 36. ACM, 2012.

[43] A. Rico, A. Duran, F. Cabarcas, Y. Etsion, A. Ramirez, and M. Valero. Trace-driven simulation of multithreaded applications. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 87–96, 2011.

[44] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters*, 10(1):16–19, 2011.

[45] D. Sanchez and C. Kozyrakis. Zsim: fast and accurate microarchitectural simulation of thousand-core systems. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 475–486, 2013.

[46] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 3–14, 2001.

[47] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. *ACM SIGOPS Operating Systems Review*, 36(5):45–57, 2002.

[48] A. Sodani, R. Gramunt, J. Corbal, H. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. Liu. Knights landing: Second-generation intel xeon phi product. *IEEE Micro*, 36(2):34–46, 2016.

[49] R. Teyssier. Cosmological hydrodynamics with adaptive mesh refinement-a new high resolution code called ramses. *Astronomy & Astrophysics*, 385(1):337–364, 2002.

[50] M. Valero, M. Moreto, M. Casas, E. Ayguadé, and J. Labarta. Runtime-aware architectures: A first approach. *International Journal on Supercomputing Frontiers and Innovations*, 1(1):29–44, 2014.

[51] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. Simflex: Statistical sampling of computer system simulation. *IEEE Micro*, 26(4):18–31, July 2006.

[52] S. White. The AMD Opteron Seattle: A 64b ARM Dense Server Processor. In *Hot Chips*, 2014.

[53] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. Smarts: accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 84–95, 2003.

[54] C. Zhang. Mars: A 64-core ARMv8 Processor. In *Hot Chips*, 2015.

[55] G. Zheng, G. Gupta, E. J. Bohm, I. Dooley, and L. V. Kalé. Simulating large scale parallel applications using statistical models for sequential execution blocks. In *International Conference on Parallel and Distributed Systems (ICPADS)*, pages 221–228, 2010.