Sampled Simulation of Task-Based Programs

Thomas Grass, Trevor E. Carlson, *Senior Member, IEEE,* Alejandro Rico, *Member, IEEE,* Germán Ceballos, *Member, IEEE,* Eduard Ayguadé, Marc Casas, and Miquel Moreto

Abstract—Sampled simulation is a mature technique for reducing simulation time of single-threaded programs. Nevertheless, current sampling techniques do not take advantage of other execution models, like task-based execution, to provide both more accurate and faster simulation. Recent multi-threaded sampling techniques assume that the workload assigned to each thread does not change across multiple executions of a program. This assumption does not hold for dynamically scheduled task-based programming models. Task-based programming models allow the programmer to specify program segments as tasks which are instantiated many times and scheduled dynamically to available threads. Due to variation in scheduling decisions, two consecutive executions on the same machine typically result in different instruction streams processed by each thread. In this paper, we propose TaskPoint, a sampled simulation technique for dynamically scheduled task-based programs. We leverage task instances as sampling units and simulate only a fraction of all task instances in detail. Between detailed simulation intervals, we employ a novel fast-forwarding mechanism for dynamically scheduled programs. We evaluate different automatic techniques for clustering task instances and show that DBSCAN clustering combined with analytical performance modeling provides the best trade-off of simulation speed and accuracy. TaskPoint is the first technique combining sampled simulation and analytical modeling and provides a new way to trade off simulation speed and accuracy. Compared to detailed simulation, TaskPoint accelerates architectural simulation with 8 simulated threads by an average factor of 220x at an average error of 0.5% and a maximum error of 7.9%.

Index Terms—Sampled simulation, task-based, analytical performance modeling.

1 Introduction

Computer architecture research heavily relies on simulation. Increasing design complexity and core counts in modern multi-core processors present new challenges to architectural simulation. The increasing amount of state-holding elements, e.g. cache memories, in future systems tends to increase the time required to simulate a system executing a given workload. Larger cache memories can require more simulation to warm up micro-architectural state. Designs with higher thread counts can additionally slow down the speed of detailed simulation, especially when the simulated threads interact with each other.

One popular technique to reduce simulation time is sampling. Sampled simulation reduces simulation time by only simulating a fraction of the workload in detail. Sampling is a well-established technique for simulation of single-threaded architectures. The prevalent techniques perform detailed simulation of either the representative program parts identified in profiling [33] or periodically via time-based sampling [38].

While sampled simulation is widely used for simulation of single-threaded architectures, techniques targeting multi-threaded applications have only been recently proposed. The main challenge in sampling simulations of multi-threaded programs is to ensure that at the beginning of each detailed simulation interval all threads have made the same amount of progress as in a full detailed simulation. A technique proposed by Casas et al. [11] detects periodic behavior in parallel applications using signal processing techniques. Carlson et al. [8] fast-forward different threads at different rates between intervals of detailed simulation. Carlson et al. [10] also propose a technique based on the insight that after a global barrier all threads are synchronized and resume execution simultaneously. The technique leverages the inter-barrier regions in barrier synchronized programs as sampling units. In this work we present a sampled simulation methodology leveraging the properties of task-based programming models.

Task-based programming models have been proposed to reduce load imbalance and thus increase parallel efficiency of future large-scale multi-core machines [3]. A task-based programming model allows the programmer to define program parts as *tasks* and to specify dependencies between those tasks. Tasks are typically instantiated many times during the execution of a program. *Over-decomposition* ensures that there are many more task instances than execution threads. The over-decomposition of a parallel program into tasks, together with dynamic scheduling of task instances to threads, transparently balances the amount of work assigned to each thread. Inter-task dependencies enforce synchronization only when necessary.

In this work we present TaskPoint, a sampled simulation methodology for dynamically scheduled task-based programs executed on multi-core machines. TaskPoint leverages task instances as sampling units and only simulates a small number of them in detail. The other task instances are simulated in a faster simulation mode, ensuring that progress in different simulated threads is

- *T. Grass conducted this work as a Ph.D. student at Barcelona Supercomputing Center (Spain). He is currently with RWTH Aachen (Germany). E-mail: thomas.grass@ice.rwth-aachen.de*
- *M. Casas, M. Moreto and E. Ayguadé are with Barcelona Supercomputing Center (Spain). M. Moreto and E. Ayguadé are also affiliated with Universitat Politècnica de Catalunya. E-mail: {firstname}.{lastname}@bsc.es*
- *T. E. Carlson is with the National University of Singapore (NUS). E-mail: tcarlson@comp.nus.edu.sg*
- *A. Rico is with Arm Ltd. (Austin, TX). E-mail: alejandro.rico@arm.com*
- *G. Ceballos is with Uppsala university. Email: german.ceballos@it.uu.se*

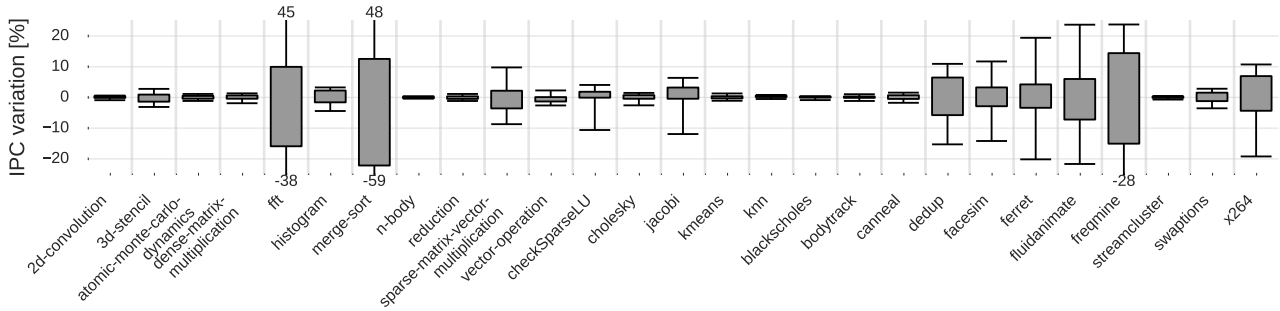


Fig. 1. IPC variation across all task instances for native execution with 8 threads, normalized per task type

modeled correctly.

In this paper, we make the following contributions:

- We present TaskPoint, a sampled simulation technique for multi-core architectures programmed with a dynamically scheduled, task-based programming model. We propose a mechanism to accurately fast-forward an architectural simulation of a task-based program. During fast-forwarding, we simulate task instances at the IPC of previous, similar instances which have been simulated in detail. We evaluate different techniques for identifying classes of similar task instances.
- For applications with varying behavior across instances of the same task type, we employ basic-block vectors (BBVs) and clustering to identify classes of similar behavior. We show, how we (i) identify multiple classes of behavior among task instances of the same task type and (ii) merge task instances with similar behavior belonging to different types. We use an analytical performance model to improve simulation accuracy during simulation in fast-forwarding mode. Our approach combines the speed of analytical models with the accuracy of detailed simulation.
- We evaluate TaskPoint simulating 27 task-based parallel benchmarks, including a task-based version of the PARSEC benchmark suite. We evaluate different clustering techniques for identifying classes of task instances which can serve as samples for one another. By combining DBSCAN clustering with analytical performance modeling, we speed up architectural simulation by a factor of 220x for 8 threads and 23x for 64 threads. The average simulation error is 0.46% for 8 threads and 1.32% for 64 threads. To the best of our knowledge, this is the first technique combining detailed simulation with analytical performance modeling, in which the analytical performance model drives the sampling process. Compared to k-means clustering and clustering based on the task type, DBSCAN with analytical modeling offers the best combination of speedup and error at a low error variance.

The remainder of this paper is organized as follows. In Section 2, we provide background and motivation of our work. In Section 3, we present TaskPoint. Next, we introduce the experimental setup in Section 4. We evaluate TaskPoint in Section 5. Finally, we present related work in Section 6, before we conclude in Section 7.

2 Background and Motivation

This section provides background on task-based programming models. We then motivate our work with an analysis of performance variation in native execution of 27 task-based parallel benchmarks.

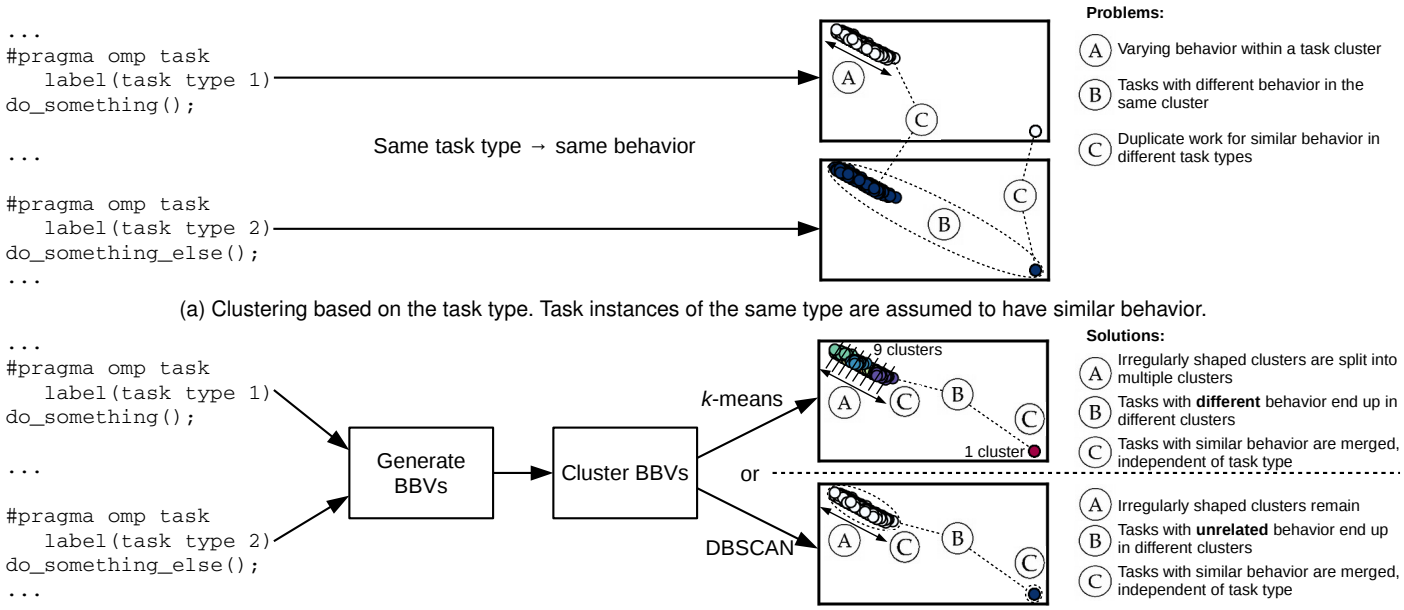
2.1 Parallel Programming Models

In traditional parallel programming models for shared memory systems, like *POSIX Threads* [6], the programmer explicitly decomposes an application into concurrent instruction streams and manages synchronization between those. These instruction streams are processed simultaneously by different threads. A common problem with multi-threaded programs is load imbalance. Load imbalance occurs when different threads reach a synchronization point at different points in time.

Task-based programming models have the potential to alleviate load imbalance and thus increase parallel efficiency. When implementing a parallel program using a task-based programming model, the programmer specifies program parts as *tasks* and, optionally, data dependencies between these tasks. Tasks are instantiated many times during the execution of a program, resulting in a large number of task instances, each of which operates on different data. A runtime system dynamically schedules task instances to execution threads.

Due to a fine-grained *over-decomposition* of the application, the number of task instances is typically much larger than the number of execution threads. This allows the runtime environment to dynamically balance the workload assigned to each thread. There are proposals for further optimizations which require the architecture to interface directly with the runtime environment [12], [34].

In this work, we differentiate between *task types* and *task instances*. In OpenMP, task types are declared by means of a `#pragma omp task` statement preceding a function declaration, a function call or a code block. Task declaration statements can contain additional information, e.g. data inputs and outputs of a specific execution of the statement. Every execution of a task declaration statement at execution time results in the creation of a task instance. All task instances resulting from the same task declaration statement in the source code are said to be of the same task type. In a typical task-based program, the number of task types is up to a few tens, whereas the number of task instances lies in the order of tens to hundreds of thousands.



(a) Clustering based on the task type. Task instances of the same type are assumed to have similar behavior.

(b) Using BBVs and clustering algorithms. Classes of behavior are identified using clustering.

Fig. 2. Overview of TaskPoint without (2a) and with (2b) clustering. Clustering detects multiple classes of behavior within a task type and merges tasks with similar behavior belonging to different task types.

2.2 Performance Variation of Task-Based Programs

In order to motivate TaskPoint, we analyze performance variation in native execution of 27 benchmarks. The investigated benchmarks are introduced in Section 4.

Figure 1 shows IPC variation across task instances observed in native execution with 8 threads on a system with an Intel SandyBridge-EP E5-2670 CPU running at 2.6 GHz and 128 GB of DDR3-1600 as main memory. For an easy comparison across different benchmarks, we normalize the IPC of all task instances to the average IPC of their respective task type. The solid box of each box plot indicates the range from the first to the third quartile of the normalized IPC values, while the whiskers extend from the fifth to the 95th percentile. IPC values of task instances below the fifth and above the 95th percentile are treated as outliers.The Figure shows that for 16 out of 27 benchmarks performance variation lies within $\pm 5\%$. However, for the remaining benchmarks instances belonging to the same task type exhibit significant performance variation.

In simulations of regular applications, TaskPoint uses instances of the same task type as samples for one another. In irregular applications, we apply basic-block vectors (BBVs) [33] and clustering to identify classes of task instances with similar behavior. A performance prediction obtained from an analytical performance model is used to refine the result.

2.3 Identifying Representative Task Instances

Our analysis of performance regularity on a per-task-type basis in Figure 1 shows that, for many applications, task instances of the same task type behave similarly in terms of performance. Therefore, it is reasonable to assume that, in those cases, instances of the same task type can serve as performance samples for one another. However, the figure also shows that some benchmarks expose a significant performance variation among task instances. Examples are *merge-sort*, *fft*, *freqmine* and *dedup*. These applications require more sophisticated techniques in order to achieve high simulation speed and low simulation error.

BBVs [33] have been used in the past to characterize phases of a workload and identify representative workload regions. A BBV is a vector with as many dimensions as there are static basic blocks in the simulated application. Each dimension contains the number of executed dynamic instructions of the corresponding basic block during a certain time interval. In this work, we evaluate one BBV per executed task instance.

Figure 2a illustrates TaskPoint without BBVs, clustering and analytical modeling. In this figure BBVs are solely used for the purpose of illustration and not for task clustering or analysis. The figure shows the BBVs of two task types of *merge-sort*, projected into a two-dimensional plane. During *merge-sort*'s recursive phase, each task instance creates two child instances which belong to different task types, but consist of the same static code. When the recursion terminates the behavior of the last generation of task instances changes. The results are (i) the two distinct clusters of task instances for each task type in Figure 2a and (ii) the pairwise similarity between the upper-left and lower-right clusters of both task types. In summary, instances belonging to the same task type can behave differently, while instances belonging to different task types can expose the same behavior.

Each task type consists of two clearly distinct clusters of BBVs which expose **different behavior** and performance at execution time. One of these clusters is eccentrically shaped (A). Task instances of this cluster which are located at some distance of each other show **different performance**. Furthermore, treating both clusters of a task type as if they showed the same performance (B) leads to a simulation error of more than 40% in the case of *merge-sort*. In addition, each cluster observed in one task type is similar to a cluster in the other task type (C), resulting in **duplicated work**

during sampled simulation. Taken together, the result is an inefficiently partitioned sample that leads to simultaneously larger errors and slower performance. An ideal clustering would consist in two clusters, each of which containing two of the pairwise similar clusters shown in Figure 2a.

Figure 2b shows how TaskPoint uses BBVs and clustering to detect classes of task instances in an irregular application. We create a BBV for each task instance. If two task instances behave similarly, they typically have similar BBVs [24]. Task instances with dissimilar behavior are likely to also have dissimilar BBVs. The figure illustrates two possible clustering algorithms. k-means [27] tends to split irregularly shaped clusters into many sub-clusters. In the case of DBSCAN [17], task instances which are connected by a dense region of other task instances are clustered together. DBSCAN has been previously used to detect related computation phases in parallel applications [21].

2.4 Analytical Performance Modeling

Figure 2b illustrates that clusters detected by DBSCAN can have a large diameter (A). If this happens, using a sample to predict the performance of a task instance, which is further away in the same cluster, can introduce a simulation error. We leverage the relative accuracy of an analytical model [35] to correct this error during simulation.

Analytical performance models have been extensively used for sequential applications [23], [28] and can be classified into *empirical* and *mechanistic* models. Empirical models capture a system's behavior with machine learning techniques, e.g. support vector machines or artificial neural networks. While they can achieve good accuracy, they do not provide much inside into why a certain design achieves better or worse performance than another. Mechanistic models employ mathematical formulas to model the effect of the key architectural parameters on performance. Mechanistic models allow to study the sources of particularly good or bad performance by comparing the contribution of the different terms of the model's formula. Since they provide more insight into the sources of performance, in this work we use a mechanistic performance model.

In this work, we use an analytical performance model proposed by Van den Steen et al. [35]. The model is an extension of Interval Simulation [19]. Van den Steen et al. replace the micro-architecture dependent input of Interval Simulation by a micro-architecture independent application profile and generate the micro-architecture dependent elements of the model input using analytical models for caches, branch predictors and memory level parallelism (MLP). Caches are modeled using *StatStack* [16], a technique for modeling arbitrarily sized LRU caches. Branch predictors are modeled using the *Linear Entropy* model [28]. MLP, i.e. the number of memory accesses which can be served by the DRAM subsystem in parallel, is modeled according to a model also proposed by Van den Steen et al. [35].

3 Sampled Simulation of Task-Based Programs

In this section, we present TaskPoint. First, we introduce the requirements which need to be fulfilled by an architectural simulator in order to serve as an implementation platform for TaskPoint. Next, we show how TaskPoint uses clustering and analytical modeling to improve simulation accuracy and speedup. Then, we present the different phases of TaskPoint's sampling mechanism, namely warm-up, sampling and fast-forwarding. Afterwards, we introduce our periodic and lazy sampling policies.

3.1 Requirements for the Architectural Simulator

Our objective is to provide a sampled simulation methodology for task-based programs which does not depend on a specific architectural simulator. Therefore, we keep the requirements for the target simulator to a minimum. A simulator needs to fulfil the following two requirements to support the TaskPoint methodology:

1) The simulator needs to feature detailed, warmup and fast-forwarding simulation modes.
2) The fast-forwarding mode has to be capable of simulating each thread at a user-specified IPC.

Most contemporary architectural simulators feature several levels of detail [2], [5], [32], allowing to trade off speed for accuracy. Thus, we assume the first requirement to be trivially fulfilled. If a simulator does not support fixed-IPC simulation by default, we consider the implementation of this functionality to be a minor effort.

3.2 Clustering Task Instances

In regular applications, TaskPoint clusters task instances according to their task type. The underlying idea is that instances of the same user-defined task type perform similar computations and are thus naturally clustered. In the case of applications which show different behavior within a task type, TaskPoint identifies classes of task instances with similar behavior prior to simulation. Instances with similar behavior belonging to different task types are merged into the same cluster. In a profiling step, we determine the BBVs of all task instances of the application. Since BBVs are micro-architecture independent, the costs of BBV generation and clustering are amortized across all simulations of the application. In a trace-based simulation environment, BBVs can be generated together with the application trace.

The amount of detailed simulation required for simulations with TaskPoint increases with the number of task instance clusters of the application. In order to maximize simulation performance, it is desirable to find the minimum number of clusters which maintains high simulation accuracy. Therefore, TaskPoint relies on DBSCAN clustering. As illustrated in Figure 2a, DBSCAN typically reports a smaller number of clusters than other techniques, e.g. k-means. These clusters can be shaped irregularly, since DBSCAN finds clusters of task instances which are densely connected in the BBV space. Nevertheless, task instances of the same cluster can have different performance. We minimize the potential impact on simulation error by determining per-task-instance performance factors obtained from an analytical model, which we introduce in the following section.

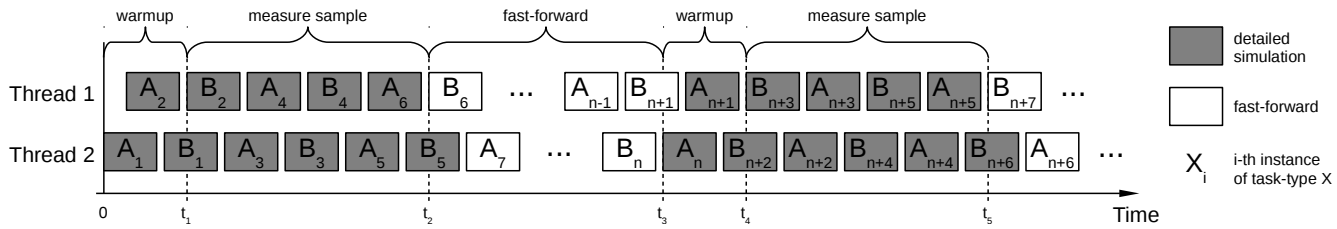


Fig. 3. Initial warmup, sampling, fast-forwarding and resampling in TaskPoint

3.3 Analytical Modeling

As stated earlier, TaskPoint clusters task instances using DBSCAN in order to achieve high simulation performance at high simulation accuracy. DBSCAN can cluster task instances with similar but not equal behavior into the same cluster, as illustrated in Figure 2b. For this reason, the performance of a task instance observed in detailed simulation may not be representative for all task instances of a cluster. TaskPoint employs an analytical performance model to determine a more accurate performance result.

First, we generate a profile of the simulated application. This profile includes the micro-architecture independent performance metrics required as inputs to the analytical model and is generated in the same profiling run as the per-task-instance BBVs used for clustering. As with the BBVs, this profile is only generated once per application.

During simulation, we apply the analytical model to predict the performance of all simulated task instances. For each simulated application, the model needs to be evaluated only once per architectural configuration. Changing only the number of simulated threads does not necessarily require one to reevaluate the model.

We use the performance information obtained from the analytical model as follows: assume that two task instances i and j belong to the same cluster. Furthermore, i has been simulated in detail and j is to be simulated in fast-forwarding mode, using i as performance sample. Let $IPC_{i,m}$ and $IPC_{j,m}$ be the IPC of task instances i and j, respectively, as predicted by the model, and $IPC_{i,d}$ the IPC obtained in detailed simulation of i. We estimate the performance $IPC_{j,ff}$ of j in fast-forwarding mode according to Equation 1:

$$IPC_{j,ff} = IPC_{i,d} \cdot \frac{IPC_{j,m}}{IPC_{i,m}} \quad (1)$$

In other words, the IPC of the sample is multiplied with the performance ratio of sample and fast-forwarded task instance. By following this approach, we combine the accuracy of detailed simulation with the relative accuracy and speed of the analytical model.

3.4 Sampling Mechanism

TaskPoint operates on the level of granularity of task instances. A task instance is simulated either in warming, detailed or fast-forwarding mode. Simulation in warming mode serves for warming architectural state. Samples are measured in detailed mode. Fast-forwarding mode accurately fast-forwards simulation time. Switching between different modes only occurs between two consecutive task instances.

Figure 3 illustrates the different phases of TaskPoint. For each task cluster, we maintain two vectors holding the IPC histories of the most recently simulated task instances. The size H of these vectors is a parameter referred to as the *history size*. Both vectors are FIFO buffers in which a newly added element replaces the oldest one. The first vector contains the history of task instances which are valid samples, i.e. which are simulated after warming up architectural state. We refer to it as the *history of valid samples*. The second vector holds the history of all task instances simulated in detailed mode, regardless of the simulation being properly warmed. We refer to it as the *history of all samples*. The former is the sample history we usually use to determine which IPC to use in fast-forwarding mode. The latter is needed if there are task clusters with instances which occur infrequently and can not be sampled in a single sampling interval. We refer to these *rare task clusters*.

In multi-threaded applications, co-running threads can interfere with each other, e.g. by competing for shared resources, through inter-thread synchronization or by invalidating data residing in remote caches. In order to correctly model contention between different threads, we simulate all threads either in detailed mode or in fast-forwarding mode. Since we assume that mode switching only occurs between two consecutive task instances, there are short phases during which some threads are simulated in fast-forwarding mode, while others are simulated in detailed mode (see t_2, t_3 and t_5 in Figure 3).

Simulation Warmup

Before conducting performance measurements, a simulation needs to be *warmed*, i.e. it needs to be put in a representative state. Warming micro-architectural state in sampled simulation is well-studied [15], [22], [26], [33], [37], [38]. In this paper, we warm the simulation by simulating an empirically determined number of task instances in detail and avoid complex warmup schemes. Instead, we focus on the sampling methodology itself. When a task instance simulated for warmup finishes execution, its IPC is added to the history of all samples.

At simulation start, all simulated micro-architectural structures are in their initial (*cold*) state. During detailed simulation, state-holding elements begin to fill until occupancy reaches a steady state. In this work, we assume that simulating W task instances per thread at simulation start is sufficient for putting the simulator into a representative (*warm*) state. We refer to W as the *size of the warm-up interval* and evaluate different values for W in Section 5.

After a simulation phase in fast-forwarding mode, micro-architectural state is stale. Before resampling the simulation, warmup makes sure that micro-architectural state is (approximately) the same as if the whole program was

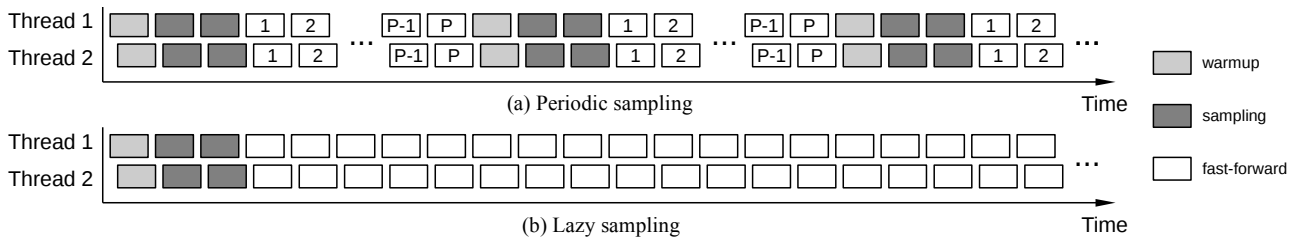


Fig. 4. Illustration of periodic sampling (a) and lazy sampling (b) as a special case of periodic sampling with infinite sampling period P

simulated in detail. Before resampling, we perform detailed simulation until every thread has simulated one task instance in detail.

Sampling

Like simulation warmup, sampling is performed in detailed simulation mode. When warmup is finished, we start treating the simulated task instances as valid samples. When a valid sample task instance finishes simulation, its average IPC is added to the history of valid samples and to the history of all samples. We trigger the transition to fast-forwarding mode when one of the following two conditions is fulfilled:

1) The history of valid samples is fully populated.
2) A certain number of task instances has been simulated without encountering any additional instance of a previously seen rare task cluster.

The first condition means that all task clusters are fully sampled. The second condition avoids spending an excessive amount of time on detailed simulation in the presence of task clusters occurring infrequently over time. Independent from the size H of the sample history, we switch to fast-forwarding when all threads have simulated 5 task instances without encountering any additional instance of a partially sampled task cluster. Thus, we avoid simulating large fractions of an application in detail while attempting to fill all sample histories.

Accurate Fast-Forwarding

After the transition to fast-forwarding mode, all task instances starting in the future are simulated in fast-forwarding mode. However, instances which started in the past are simulated in detailed mode until they complete. Task instances finishing simulation after the transition to fast-forwarding mode are only added to the history of all samples.

A task instance simulated in fast-forwarding mode is simulated with the average IPC of the history of valid samples of its cluster. If a task instance belongs to a rare task cluster whose history of valid samples is empty, we use the average IPC of the history of all samples instead. If the history of all samples of the corresponding cluster is also empty, we trigger resampling.

Instances belonging to rare task clusters tend to occur infrequently during the execution of an application. They account only for a small percentage of the total instruction count of an application and are used for infrequent tasks, e.g. setting up and deleting data structures. We find the impact of using non-representative samples for fast simulation of rare tasks to be negligible.

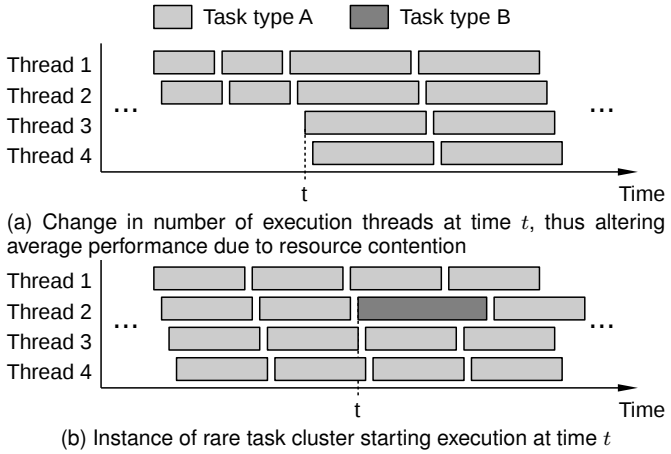


Fig. 5. Illustration of changing number of execution threads (a) and rare task (b)

One contribution of this paper is the presented fast-forwarding mechanism for architectural simulation of task-based parallel programs. Our technique fast-forwards each thread at a rate depending on the cluster of the task instance currently being simulated.

3.5 Periodic Sampling Policy

A sampling policy decides when to resample a simulation running in fast-forwarding mode. The *periodic sampling* policy, illustrated in Figure 4a, warms and samples a simulation at simulation start. Afterwards, it switches the simulation to fast-forwarding mode. When a thread has executed a number P of task instances of any cluster in fast-forwarding mode, the simulation is resampled. We refer to the parameter P as the *sampling period*. When a simulation is resampled, the entries of the history of valid samples are discarded. When resampling is complete, the simulation returns to fast-forwarding mode and the process repeats.

Simulation speedup is determined by the size of the sampling period. The larger the sampling period, the more task instances are simulated in fast-forwarding mode. In the special case of an infinite sampling period, resampling is never triggered by the sampling policy. We refer to this case as *lazy sampling*. Lazy sampling is illustrated in Figure 4b. If the number of task instances of a program is too small or the sampling period is too large, a simulation finishes during the first fast-forwarding interval, before any thread has simulated P task instances. In this case, periodic sampling is equivalent to lazy sampling.

Besides the aforementioned case of a thread having simulated P task instances in fast-forwarding mode, resampling is also triggered when it is impossible to accurately simulate

TABLE 1
Task-based parallel benchmarks used for the evaluation of TaskPoint

Benchmark	# Task Types	# Task Instances	Simulation time 8 threads $[h:min]$ Detailed	KMEANS	DBS+MOD	Properties
2d-convolution (2d-conv)	1	16384	44:43	0:07	0:06	Kernel: strided memory accesses
3d-stencil (3d-st)	1	16370	28:25	0:04	0:04	Kernel: strided memory accesses
atomic-monte-carlo-dynamics (at-mc)	1	16384	5:44	0:01	0:01	Kernel: embarrassingly parallel
dense-matrix-multiplication (dmtmul)	1	17576	44:30	0:06	0:05	Kernel: high data reuse, compute bound
fft	8	25024	47:28	16:08	21:03	Kernel: variable stride memory accesses
histogram (hist)	1	16384	12:45	0:02	0:02	Kernel: atomic operations
merge-sort (m-sort)	4	20480	16:41	1:48	2:18	Kernel: recursive task instantiation
n-body	2	25000	13:05	0:04	0:03	Kernel: irregular memory accesses
reduction (reduct)	2	16384	40:39	0:19	0:18	Kernel: parallelism decreases over time
sparse-matrix-vector-multiplication (sp-mv)	1	1024	19:24	0:14	0:02	Kernel: load imbalance, memory bound
vector-operation (vec-op)	1	16400	32:27	0:08	0:09	Kernel: regular, memory bound
sparseLU (splu)	11	22058	19:25	0:40	0:40	Decomposition of large, sparse matrices
cholesky (chol)	4	19600	51:31	0:38	0:30	Decomposition of Hermitian matrices
jacobi	9	20480	19:14	0:27	0:36	Jacobi iterative method
kmeans	6	16337	22:58	0:43	0:28	Clustering based on Lloyd's algorithm
knn	2	18400	34:31	3:22	0:06	Instance-based machine learning algorithm
blackscholes (bkschl)	2	24500	12:27	0:02	0:02	Option price calculation
bodytrack (bdytrk)	7	21439	54:35	3:36	2:04	Human body tracking with multiple cameras
canneal (cneal)	1	16384	12:21	0:16	0:02	Cache-aware simulated annealing
dedup	4	15738	80:01	2:23	0:11	Combination of global and local compression
facesim (fcsim)	12	20086	17:45	3:26	13:17	Physical modeling of the human face
ferret	6	12288	77:39	1:41	0:21	Image similarity search
fluidanimate (fldanm)	9	8225	31:53	1:16	1:49	Simulation of incompressible fluids
freqmine (frqmn)	7	1932	12:32	0:59	1:02	Frequent Pattern Growth method
streamcluster (strclr)	10	14656	27:41	0:42	0:31	Online clustering algorithm
swaptions (swptns)	1	16384	71:46	1:17	0:10	Monte-Carlo simulation of swaption prices
x264	3	383	123:44	62:15	67:07	H.264 video compression

a task instance in fast-forwarding mode. This happens in the following two cases.

Figure 5a shows a case where the number of threads participating in task execution changes at runtime, e.g. when the simulated application enters a phase exposing more parallelism. When the number of execution threads changes, so does the contention on shared resources, like shared caches and main memory. This affects per-thread performance and invalidates previously measured samples. Resampling avoids prediction errors due to non-representative samples.

Figure 5b shows a case where the first instance of a new task cluster is encountered while simulating in fast-forwarding mode. When encountering an instance of a previously unknown cluster, the cluster's sample history is empty. Therefore, it is impossible to accurately simulate this task instance in fast-forwarding mode. We circumvent this problem by triggering resampling.

With this resampling strategy, both periodic sampling and lazy sampling account for phase changes in the application. If a new phase starts with task instances of a new cluster, the simulation is resampled. The same holds for changes in the available computation resources or the available parallelism.

4 Experimental Setup

In this section, we introduce the experimental setup we use to implement and evaluate TaskPoint. First, we introduce the task-based programming model OmpSs. Subsequently, we present the 27 benchmarks and the two architectures we use in our evaluation. Finally, we elaborate on the TaskSim simulator and our implementation of fast-forwarding a simulation at arbitrary IPC.

The OmpSs Programming Model

For our evaluations we choose the OmpSs programming model [3]. The OmpSs compiler and runtime environment are available as open source software. OmpSs allows to declare tasks and annotate them 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 are part of the specifications of OpenMP 3.0 and 4.0.

Benchmarks

Table 1 lists the benchmarks used in our evaluation. They represent a variety of workloads and are implemented using the OmpSs programming model. While the majority of benchmarks represent workloads common to high-performance computing (HPC), *blackscholes*, *bodytrack*, *canneal*, *dedup*, *facesim*, *ferret*, *fluidanimate*, *freqmine*, *streamcluster*, *swaptions* and *x264* are part of the PARSEC benchmark suite [4], [14]. We do not use *raytrace* and *vips* because task-based versions of these applications are not currently available. Whenever possible, we simulated the application for an equivalent of at least ten seconds of single-threaded execution on a state-of-the-art machine. For the PARSEC benchmarks we used the *simlarge* input sets. Table 1 lists the number of task instances and the time required for simulating each benchmark with 8 simulated threads in

detail. The table also lists the simulation time for each benchmark when using two of the techniques proposed in this paper, namely clustering of task instances based on k-means (KMEANS) and DBSCAN clustering in combination with analytical performance modeling (DBS+MOD).

Simulated Architectures

We evaluate the fidelity of our methodology by investigating simulation speedup and execution time error of multi-threaded simulations of two different multi-core architectures. One resembles a server-class system, while the other resembles a low-power mobile platform. Table 2 lists the key characteristics of the simulated architectures. The high performance architecture features a large reorder buffer and a three-level cache hierarchy, as found in HPC systems. The low-power architecture has a smaller reorder buffer and two levels of cache memories, as is typical for battery powered mobile systems. Recently, low-power systems are gaining interest for applications in HPC [29]. We used the same parameters for simulation and analytical performance modeling.

TABLE 2
Architectural parameters of high-performance and low-power configurations used for simulation and analytical modeling

Parameter	High-perf.	Low-power
Reorder-buffer size	168	40
Issue width	4	3
Commit rate	4	3
Cache line size	64 B	64 B
L1 cache	32 kB private 4 cycles latency 8-way associative	32 kB private 4 cycles latency 2-way associative
L2 cache	2 MB private 11 cycles latency 8-way associative	1 MB shared 21 cycles latency 16-way associative
L3 cache	20 MB shared 28 cycles latency 20-way associative	none

The TaskSim Simulator

We evaluate our methodology using the TaskSim simulator [30], [31]. TaskSim is a cycle-accurate, trace-driven performance simulator for multi-core architectures. It interfaces with an unmodified version of the OmpSs runtime system. The runtime system schedules the task instances of the simulated application for execution on the simulated processor cores.

TaskSim has a detailed and a fast-forwarding simulation mode. The detailed mode is based on the *Reorder-Buffer Occupancy Analysis* model proposed by Lee et al. [25]. When running in detailed mode, TaskSim models a user-defined memory hierarchy including private and shared cache memories, interconnect structures and DRAM.

In the fast-forwarding mode, called *burst mode*, TaskSim only accounts for the number of CPU cycles between events, in our case between the beginning and the end of the execution of a task instance. In the existing implementation, TaskSim reads a task instance's cycle count from the application trace. This is the number of cycles it takes to execute the task instance on the system used for trace generation. In contrast, our fast-forwarding mechanism calculates the duration of a task instance at the beginning of its simulation. Using the mean IPC of the sample history of a task instance i's task type or cluster T and its dynamic instruction count I_i, we estimate its number of execution cycles C_i according to $C_i = \frac{I_i}{IPC_T}$. The result is the number of cycles it takes to execute the task instance at an IPC of IPC_T, the average IPC of the instance's task type. The dynamic instruction count is read from the application trace.

In the scope of this work, we extended TaskSim with the capability to switch between warming, detailed and fast-forwarding mode at runtime. Instead of using previously recorded cycle counts from a trace, our implementation of fast-forwarding mode uses cycle counts predicted by our fast-forwarding mechanism. To the best of our knowledge, this is the first fast-forwarding mechanism applying different IPCs to different task instances of a task-based program. Our mechanism allows fast-forwarding of dynamically scheduled parallel programs, in which the per-thread instruction stream is a-priori unknown.

5 Evaluation

In this section, we conduct a sensitivity analysis of TaskPoint's model parameters. Then, we evaluate the simulation error and speedup for different ways of identifying clusters of task instances which are used as samples for prediction the performance of other instances of the same cluster. We find that k-means clustering achieves high accuracy at a significant simulation speed reduction. However, DBSCAN clustering combined with analytical modeling is superior in terms of simulation speed without a significant loss in accuracy. Finally, we compare TaskPoint to an existing sampled simulation technique.

5.1 Overview of the Results

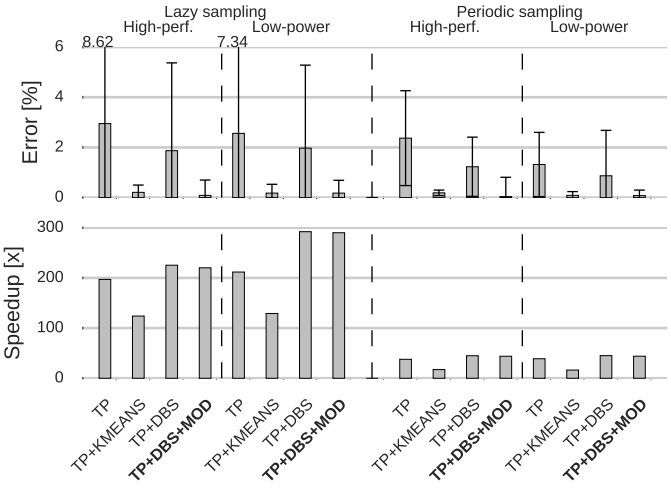


Fig. 6. Summary of results. Average error and speedup of high-performance and low-power architectures with 8 threads for TaskPoint with clustering based on the task type (TP), using k-means (TP+KMEANS), using DBSCAN (TP+DBS) and using DBSCAN in combination with analytical modeling (**TP+DBS+MOD**). Only the latter combines high simulation speed, low error and low error variance.

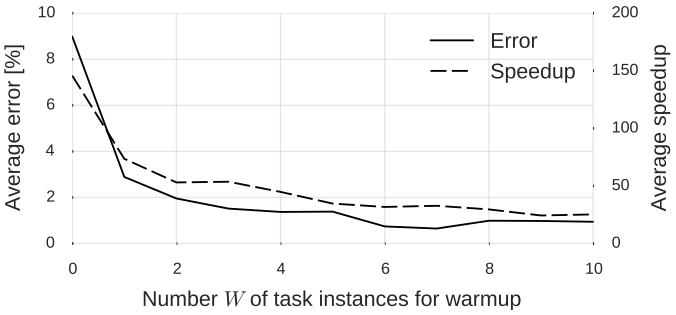
Figure 6 gives an overview of the results for simulations of the two architectures introduced in Tab. 2. The results are shown for 8 simulated threads, assuming the sampling parameters we evaluate in the next section. The figure shows error and speedup for clustering (i) based on the task type (TP), (ii) using k-means (TP+KMEANS), (iii) using DBSCAN (TP+DBS) and using DBSCAN in combination with analytical modeling (TP+DBS+MOD). The error is the relative execution time difference between a full detailed simulation and a simulation using TaskPoint. The figure clearly shows that k-means clustering improves simulation error over only using the task type, but does so at poor simulation performance. DBSCAN alone achieves the highest simulation performance, but at a large simulation error. We leverage the relative accuracy of analytical modeling to correct this error on a per-cluster basis, effectively combining the accuracy of detailed simulation with the speed of analytical models. This approach reduces the simulation error for 8 threads to 0.46% on average and 7.9% at most. The simulation speedup of 220x is only 2.3% slower than DBSCAN alone, but 11.7% faster than clustering based on the task type, and 77.5% faster than k-means. Overall, DBSCAN with analytical modeling achieves a high simulation speed at a low simulation error and low error variance, offering a superior trade-off between simulation speed and complexity.

5.2 Evaluating the Model Parameters

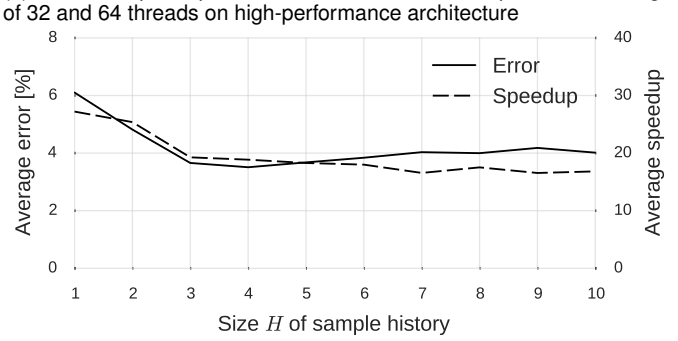
We determine the optimal warmup and sampling parameters following an incremental approach. First, we determine the optimal number of task instances (W) needed for warmup at simulation start. Afterwards, we consider different numbers of task instances constituting the sample history (H). Finally, we explore a range of values for the sampling period (P). We evaluate the model parameters simulating the high-performance architecture in Table 2 and clustering of task instances based on their task type. We apply the same model parameters to the other clustering techniques. Our results indicate that the model parameters, once determined, can be applied to different clustering techniques without significantly increasing the simulation error. We also considered methods to dynamically determine the sampling parameters. This more complex technique achieves similar results in terms of simulation error and speedup, while considerably increasing complexity.

In order to determine the optimal value for W we set $H = 10$ and $P = \infty$ and evaluate different values ranging from $W = 0$ (no warmup) to $W = 10$. Figure 7a shows error and speedup w.r.t. full detailed simulation, averaged over simulations with 32 and 64 threads. The reported values are averaged over the benchmarks and kernels with an error $> 5\%$ for at least one value of W, namely *2d-convolution*, *3d-stencil*, *atomic-monte-carlo-dynamics*, *knn*, *fft*, *merge-sort*, *blackscholes*, *dedup*, *freqmine* and *x264*. We found that $W = 2$ yields an average error of less than 2%. Larger values of W do not significantly reduce the average error, but for some benchmarks they significantly reduce simulation speedup by requiring more detailed simulation. Therefore, for the remainder of this paper, we set $W = 2$.

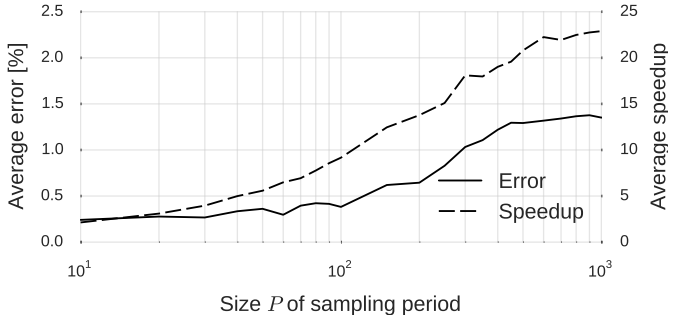
Next, we evaluate different values for H, the size of the sample history. For this purpose, we set $P = \infty$. Note that



(a) Error and speedup for different sizes W of warmup interval, average of 32 and 64 threads on high-performance architecture



(b) Error and speedup for different sizes H of task instance history, average of 32 and 64 threads on high-performance architecture



(c) Error and speedup for different sizes P of sampling period, average of 32 and 64 threads on high-performance architecture

Fig. 7. Error and speedup for different sizes of warmup interval (a), sample history (b) and sampling period (c)

we already set $W = 2$. Figure 7b shows error and speedup for different sizes H of the sample history, averaged over simulations with 32 and 64 threads of the aforementioned benchmarks. We found that $H = 4$ minimizes the average error. This value also minimizes the standard deviation of the average error, which is not shown in the Figure. Larger values of H do not only result in a larger average error, but also in lower simulation speedup. Therefore, for the remainder of this paper, we set $H = 4$.

5.3 Clustering based on the task type

Finally, we explore different sizes of the sampling period P. With $W = 2$ and $H = 4$ already fixed, P is the only remaining parameter. Figure 7c shows the average error for values of P ranging from 10 to $1,000$. We find that average error and speedup increase with the size of the sampling period. The larger the value of P, more task instances are simulated in fast-forwarding mode. Since the total number of task instances of a program is constant, the fraction of detailed simulation decreases, resulting in increasing speedup. For $P \geq 1000$ error and speedup remain constant. At this

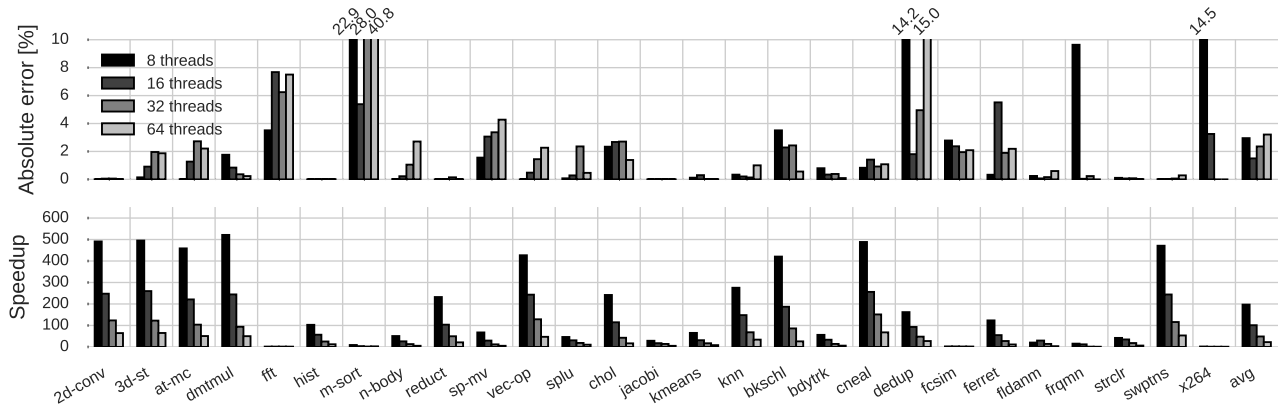


Fig. 8. Error and speedup for simulations of the high-performance architecture with different numbers of threads, using clustering based on the task type and lazy sampling.

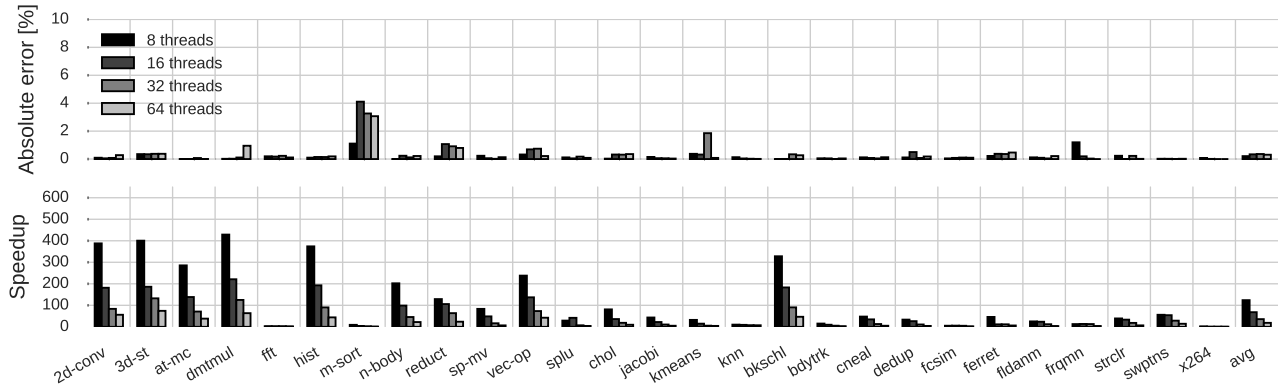


Fig. 9. Error and speedup for simulations of the high-performance architecture with different numbers of threads, using k-means clustering and lazy sampling.

point, none of the investigated programs has a sufficient number of task instances for resampling the simulation at least once and periodic sampling becomes equivalent to lazy sampling.

We aim for a simulation error of less than 1%. A sampling period $P = 250$ yields an error of 0.8% and a simulation speedup of 15.1x, averaged over the benchmarks used in our sensitivity analysis. In the remainder of this section, we evaluate TaskPoint for periodic sampling with $P = 250$ and for lazy sampling (periodic sampling with $P = \infty$). For the remainder of this paper, we focus on simulations of the high-performance architecture introduced in Table 2. The overview of our results in Figure 6 indicates that for the low-power architecture similar results can be expected.

Figure 8 shows simulation error and speedup when clustering task instances based on their task type. The average simulation error is up to 3.21% for 64 threads. At 40.8%, we observe the largest simulation error for *merge-sort*, which is also the benchmark showing the largest performance variation ranging from -59% to 45% per task type in Figure 1. Other benchmarks showing high simulation errors are *fft*, *freqmine* and *dedup*. These benchmarks are also among the benchmarks showing the largest performance variation (fft: -38% to 45%, freqmine: -28% to 24%, dedup: -15% to 11%).

On the other hand, benchmarks showing low performance variation in Figure 1 also show low simulation error when clustering task instances based on their task type. Examples are *2d-convolution*, *3d-stencil* and *atomic-monte-carlo-dynamics*.

For some benchmarks, e.g. *3d-stencil*, *atomic-monte-carlo-dynamics* and *n-body*, the simulation error increases for increased numbers of simulated threads. We find that for these benchmarks the number of last-level cache misses per kilo instruction (MPKI) increases for larger numbers of threads, which also increases performance variation. These errors can be improved by increasing the size of the sample history at the expense of simulation speedup.

The largest simulation speedup is observed for benchmarks which have only one task type, e.g. *swaptions*, which shows a speedup of 471x. The average speedup ranges from 197x for 8 threads to 22x for 64 threads. For lazy sampling, simulation speedup is mainly limited by the number of task instances of the simulated application. For increasing numbers of simulated threads, an increasing fraction of a program's task instances needs to be simulated in detail, decreasing the fraction which can be simulated in fast-forwarding mode.

As illustrated in Figure 2a, clustering of task instances based on the task type can produce clusters of task instances of dissimilar performance. This is especially true for applications with irregular performance or multiple classes of behavior among instances of the same task type. This justifies the use of more elaborate clustering techniques to identify classes of task instances which can serve as samples for one another.

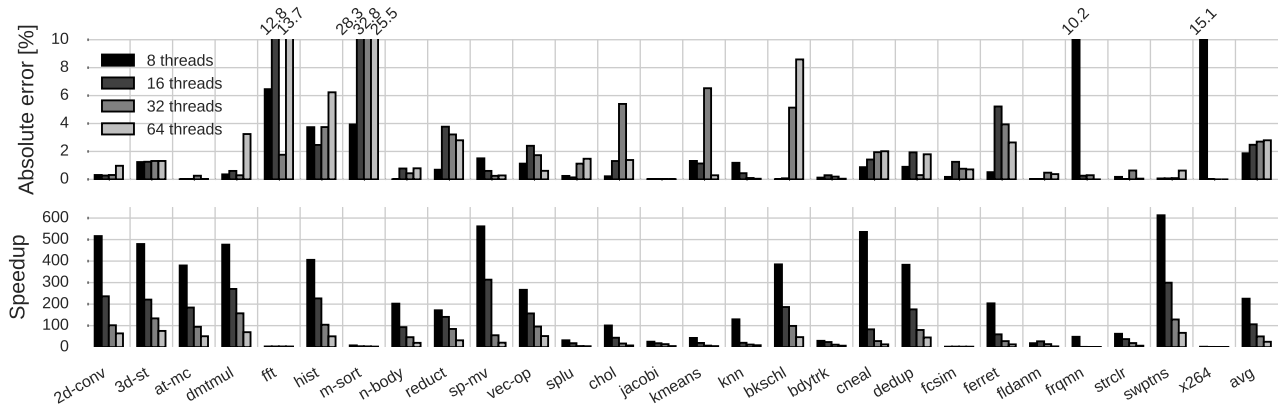


Fig. 10. Error and speedup for simulations of the high-performance architecture with different numbers of threads, using DBSCAN clustering and lazy sampling.

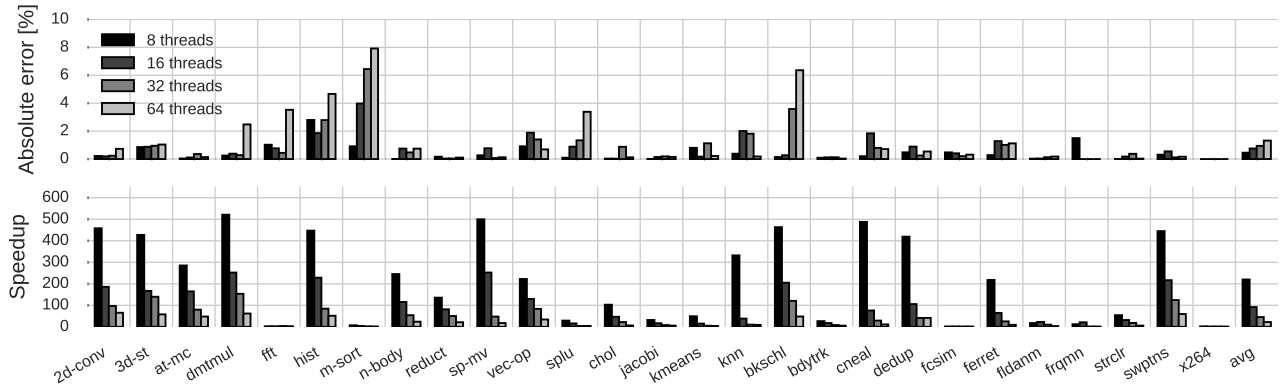


Fig. 11. Error and speedup for simulations of the high-performance architecture with different numbers of threads, using DBSCAN clustering in combination with analytical performance modeling and lazy sampling.

5.4 Clustering using k-means

Figure 9 shows simulation error and speedup for clustering of task instances based on k-means. We cluster task instances by using their BBVs as an input to the SimPoint [33] tool, which relies on k-means. We set SimPoint's *maxK* parameter to 20, limiting the maximum number of clusters to 20. SimPoint finds the clustering with the smallest number of clusters which still achieves a low per-cluster variance. As illustrated in Figure 2b, k-means classifies task instances into clusters of regular performance, whereas irregular clusters get sub-clustered into regular ones. In our evaluation, *atomic-monte-carlo-dynamics* is the benchmark resulting in the largest number of clusters ($k = 18$). The simulation error when applying k-means is the lowest of all investigated clustering techniques, with an average error of up to 0.36% and a maximum error of 4.1% in case of *merge-sort* simulated with 16 treads. Simulation based on k-means clustering achieves a very high average simulation accuracy at the expense of simulation time.

At an average simulation speedup of 124x for eight simulated threads simulations using k-means are 59% slower than when clustering task instances based on their task type. The reason for this slowdown lies in the increased number of clusters for some applications, resulting in detailed simulation of a larger fraction of the workload compared to clustering based on the task type. The large loss in simulation speed indicates that it is desirable to keep the total number of clusters as low as possible. This can be achieved by relying on a density-based clustering algorithm, e.g. DBSCAN.

5.5 DBSCAN clustering

Figure 10 shows simulation error and speedup for simulations using DBSCAN to cluster task instances. The figure shows that regular applications achieve a simulation accuracy similar to clustering based on the task type. However, compared to clustering based on the task type, irregular benchmarks expose a higher simulation error. We identify two sources for this increase in simulation error: First, similar instances of different task types get merged into one cluster due to the similarity of their BBVs. Second, DBSCAN can report clusters with large diameter, as in the example of *merge-sort* illustrated in Figure 2b.

In *fft*, task instances with similar BBVs belonging to different task types get merged by DBSCAN into one cluster. The same effect occurs in *histogram*, *swaptions* and *x264*. For these benchmarks, DBSCAN finds a total number of clusters which is lower than the number of task types. Although task instances in the same cluster are generally similar, they can have different performance. The benchmarks *merge-sort* and *freqmine* contain input-dependent task types whose instances get clustered into clusters with a large diameter. A result is performance variation among task instances of the same cluster, which increases simulation error.

On average, DBSCAN finds a lower number of clusters and thus achieves larger simulation speedup than k-means and clustering based on the task type. Simulation speedup ranges from 225x for 8 threads to 25x for 64 threads. Clusters

can contain task instances of different performance. If unaccounted for, these differences can cause high simulation errors, as shown in Figure 10.

5.6 DBSCAN clustering and analytical modeling

Figure 11 shows simulation error and speedup for simulations using DBSCAN clustering to identify classes of task instances and performance estimations obtained from the analytical performance model. The difference to only using DBSCAN clustering is that, during fast-forwarding, task instances are simulated at an IPC according to Equation 1 instead of the average IPC of their respective task cluster.

The results show that analytical modeling significantly improves simulation accuracy over using only DBSCAN. The *merge-sort* benchmark shows errors of up to 32.8% with DBSCAN alone. Analytical modeling reduces this error to 7.9% in the worst case. The simulation error of *fft* is improved from 13.7% to 3.5%. For *freqmine* and *x264* the error is reduced from more than 10% to 1.5% and 0.1%, respectively.

The average speedup of DBSCAN combined with analytical modeling is slightly lower than the speedup achieved when only using DBSCAN due to the cost of evaluating the analytical model for each task instance. The biggest relative decrease in simulation speed amounts to 13.5% and is observed for 16 simulated threads. However, given the significant improvement of simulation accuracy we find the slight loss in simulation speed to be acceptable.

5.7 Comparison to Dynamic Sampling

In the following, we compare error and speedup of TaskPoint and Dynamic Sampling [18], a technique proposed by Falcón et al. During simulation, Dynamic Sampling advances threads in fast-forward mode based on previously seen performance and reevaluates this progress whenever changes to internal simulator statistics are seen. For a brief introduction to Dynamic Sampling we refer to the subsection on multi-threaded simulation sampling in Section 6.

For this comparison, we use the COTSon simulator [2], which is distributed with an implementation of Dynamic Sampling. We configure COTSon to simulate an architecture resembling the low-power architecture in Table 2, running Ubuntu 16.04 Server as its operating system. We are able to run COTSon successfully with up to 4 simulated cores and analyze five applications leading to a similar average error as the full benchmark set. For these applications, we use the same application binaries and input sets as in the simulations with TaskPoint.

We evaluate the parameter space of Dynamic Sampling and confirm the finding of Falcón et al., that the rate of translation cache invalidations is a suitable metric for controlling resampling. We found that an interval length of 100 thousand cycles for the warming, detailed simulation and fast-forwarding intervals in combination with a resampling threshold of 100 leads to a good trade-off of simulation speed and accuracy. Resampling is only triggered if the monitored metric exceeds the threshold, i.e., we allow for an infinite number of consecutive fast-forwarding intervals.

Figure 12 shows simulation error and speedup for simulations of 5 benchmarks with TaskPoint and Dynamic Sampling, simulated with 1 and 4 threads. For 1 thread, both techniques show a high simulation accuracy. We attribute the slightly higher error of Dynamic Sampling to effects introduced by the simulated operating system, which is not simulated in the user-level simulations we perform with TaskPoint. For 4 simulated threads, Dynamic Sampling shows an average simulation error of 2.94%, while the average error of TaskPoint amounts to 0.15%. Again, both techniques show an acceptable level of simulation error. Besides operating system effects, we attribute the larger error of Dynamic Sampling to the fact that the fast-forwarding mechanism does not model the changes in resource contention caused by task instances executing the same static code, but showing input dependent behavior. TaskPoint clusters these task instances into different clusters and fast-forwards them at their specific rate.

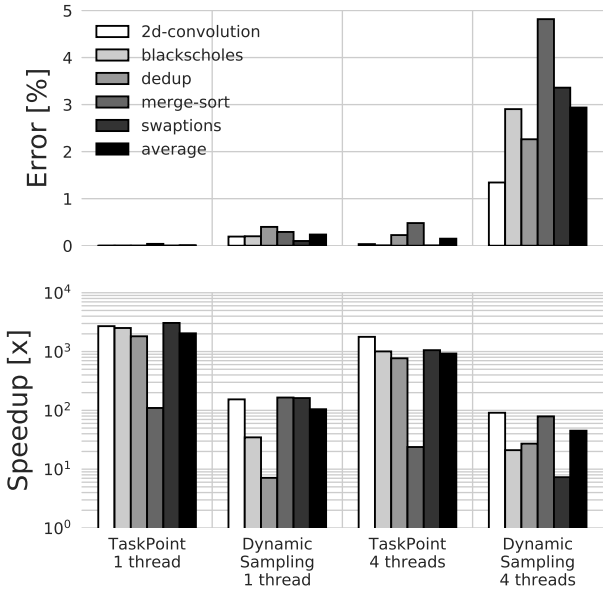


Fig. 12. Error and speedup of TaskPoint and Dynamic Sampling for 1 and 4 simulated threads.

The speedup of simulations with TaskPoint is around one order of magnitude higher than the speedup observed for Dynamic Sampling. Moreover, our fast-forwarding mechanism minimizes the amount of detailed simulation and avoids functional simulation of the fast-forwarded application parts, achieving higher speedups in simulation time as a result.

5.8 Summary

In this section we compare simulation accuracy and speedup for TaskPoint using different clustering techniques, namely clustering based on the task type, k-means and DBSCAN. Our results show that clustering based on the task type gives good accuracy and speedup for applications with regular performance per task type. The technique achieves an average error of 2.95% and a speedup of 197x.

k-means clustering achieves a significantly higher simulation accuracy across all benchmarks, but sacrifices up to 59% of the simulation speedup compared to clustering based on the task type. Overall, k-means shows an average error of 0.21% and an average speedup of 124x for 8 threads.

We showed that DBSCAN clustering alone shows large simulation errors of up to 32.8% in the case of *merge-sort*. Although it yields a low average simulation error of 1.87% and a speedup of 225x for 8 threads, the error variation across different benchmarks is significant, as shown in Figure 6.

The combination of Analytical modeling and DBSCAN reduces the average simulation error for 8 threads to 0.46%. At 220x, the average simulation speedup is only 2.3% lower compared to DBSCAN without analytical modeling. Equally important, the variation of the error across all benchmarks is also significantly improved. Overall, DBSCAN with analytical modeling yields good simulation accuracy at a high simulation speed across all investigated benchmarks.

Finally, we demonstrate that TaskPoint's accuracy is comparable to existing techniques. Compared to an existing technique, Dynamic Sampling, the speedup of TaskPoint is an order of magnitude higher with a lower average error. TaskPoint minimizes the amount of time-consuming detailed simulation. Once obtained, a sample can be used for the remainder of the simulation, ensuring high accuracy by means of an analytical performance model. Furthermore, our fast-forwarding mechanism does not rely on functional simulation of the simulated application to advance to the next detailed region.

6 Related Work

In this section, we first introduce a number of simulators for multi-core systems. Then, we present the prevalent techniques for sampled simulation of single-threaded architectures. Afterwards, we review recent work on sampled simulation of multi-threaded architectures.

Multi-Threaded Architectural Simulation

COTSon [2] is a full-system simulator that decouples functional and timing simulation. Its functional simulation relies on execution of the simulated program in a virtual machine. COTSon features several levels of detail and supports sampling.

In addition to performance, ESESC [1] also simulates a future design's power consumption and thermal behaviour. ESESC applies time-based sampling to simulation of multi-threaded applications.

The full-system simulator *gem5* [5] features CPU models at several levels of detail, ranging from a model employing hardware virtualization without modeling micro-architectural details to a detailed model of a superscalar out-of-order CPU.

In contrast to the aforementioned simulators, *Sniper* [7] features two different analytical CPU models. Instead of modeling micro-architectural structures within the CPU, it employs analytical core models, namely the mechanistic *Interval Simulation* model [19] or the *instruction-window centric* model [9]. Furthermore, Sniper supports time-based sampling [8].

Single-Threaded Simulation Sampling

In their *SimPoint* methodology [33], Sherwood et al. use basic block vectors to identify the most representative code sections to simulate in detail. SimPoint requires a-priori profiling of the application in order to identify basic block vectors.

SMARTS [38] and TurboSMARTS [36] switch periodically between warmup, detailed simulation and fast-forwarding to accurately determine the performance of a single-threaded application. Warmup allows for higher accuracy by putting simulated micro-architectural structures into a representative state. After warmup, the performance metrics of interest are measured in detailed mode. Fast-forwarding mode maintains the correct state of the simulated cache memories with functional warming. The durations of the respective intervals are user-specified parameters.

The single-threaded sampling techniques introduced in this subsection can only be applied to single-threaded simulations of task-based programs, not multi-threaded programs. However, with TaskPoint, we present a technique targeting the more general case with arbitrary thread counts.

Multi-Threaded Simulation Sampling

There are recent techniques applying sampling to simulations of multi-threaded programs. Carlson et al. [8] and Ardestani et al. [1] apply time based sampling [13] to parallel programs. Short detailed simulation phases take turns with longer fast-forwarding phases, resulting in a reduction of simulation time. The fast-forwarding mechanism employs functional simulation, using the average IPC of the previous detailed simulation phase in order to approximate the progress rates of different threads. The lengths of the sampling and fast-forwarding intervals are determined during profiling using micro-architecture independent metrics.

BarrierPoint [10] first analyzes micro-architecture independent performance metrics of program sections between global barriers. Afterwards, the SimPoint infrastructure [33] identifies clusters of those inter-barrier regions with similar performance. Simulation time is reduced by simulating only one representative out of each cluster. BarrierPoint achieves an average simulation speedup of 24.7x with an average execution time error of 0.9%. This shows that leveraging the nature of a parallel programming model can lead to significantly higher simulation speedup.

In their *Multilevel Simulation* technique, Gonzalez et al. [20] identify representative phases (*CPU bursts*) of programs implemented in the Message Passing Interface (MPI) programming model. Representative CPU bursts are identified during profiling prior to simulation and are afterwards simulated in detail. The obtained performance information is then used to extrapolate the overall program performance. Multilevel Simulation targets distributed-memory applications, while with TaskPoint we present a technique for multi-threaded applications on shared-memory systems.

Dynamic Sampling [18] targets full-system simulation based on execution in a virtual machine and switches between warming, detailed simulation and fast-forwarding without cache warming. Resampling results in the execution of a warmup phase followed by a detailed simulation phase and is triggered based on changes of internal statistics of the virtual machine. The lengths of the warmup, detailed, and fast-forwarding phases are user-specified parameters. Resampling is also triggered after a user-specified maximum number of consecutive fast-forwarding phases. In

our evaluation in Section 6 we show the applicability of Dynamic Sampling to task-based programs. For single-threaded simulations TaskPoint achieves a simulation error comparable to the error of Dynamic Sampling. For multi-threaded simulations, TaskPoint reduces the average error by 2.79%. For both single and multi-threaded applications, TaskPoint achieves a simulation speedup one order of magnitude higher than the average simulation speedup of Dynamic Sampling.

Warming in Multi-Threaded Simulations

Warming for single-threaded simulations has been extensively studied [15], [22], [26], [33], [38]. The technique used by the BarrierPoint methodology combines two existing methodologies, namely functional warming [15] and checkpointing [37]. The resulting technique uses dynamic instrumentation to track the most recent memory accesses on a per-cache-line basis. Afterwards, this information is used to restore cache state at the beginning of each detailed simulation interval. In this paper, we simulate a fixed number of task instances for architectural warmup.

7 Conclusions

Sampled simulation is a widely used technique to achieve high performance with low error in architectural studies. Previous sampled simulation techniques have proven to be fast and accurate for statically scheduled fork-join based programs. However, they can lose speed and accuracy in simulations of dynamically scheduled task-based parallel programs.

The proposed methodology enables sampled simulation of task-based parallel programs. Sampling units are identified based on the partitioning into tasks provided by the programmer. Between detailed simulation phases, we employ a novel fast-forwarding mechanism, which correctly reflects the different progress rates of task instances belonging to different task types and adapts to phase changes in the simulated application.

In this paper, we provide a complete methodology for task-based simulations that are resilient to performance variation across instances of the same task type, while taking advantage of inter-task similarity. By combining clustering (BBVs and DBSCAN) with analytical modeling, we create a hybrid simulation methodology, changing the classical trade-off curve of simulation speed and accuracy. Compared to using only DBSCAN clustering, we improve accuracy while maintaining performance for all benchmarks. We also significantly decrease the error variance. Our methodology enables simulations of benchmarks with large input sets, which previously were infeasible.

Acknowledgments

This paper was developed with the support of the HiPEAC network that received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement no. 687698, by the Spanish Government (Severo Ochoa SEV2015-0493), the Spanish Ministry of Science and Innovation (contract TIN2015-65316-P), Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272), the RoMoL ERC Advanced Grant (GA 321253), and the Mont-Blanc project (EU-FP7-610402 and EU-H2020-671697). M. Moreto has been partially supported by the Ministry of Economy and Competitiveness under Juan de la Cierva postdoctoral fellowship JCI-2012-15047. M. Casas is supported by the Ministry of Economy and Knowledge of the Government of Catalonia and the Cofund programme of the Marie Curie Actions of the EU-FP7 (contract 2013BP B 00243). T. Grass has been partially supported by the AGAUR of the Generalitat de Catalunya (grant 2013FI B 0058).

References

[1] E.K. Ardestani and J. Renau, "ESESC: A Fast Multicore Simulator Using Time-Based Sampling," in *Proc. 19th IEEE Int'l Symp. High Performance Computer Architecture*, 2013, pp. 448–459.

[2] E. Argollo *et al.*, "COTSon: Infrastructure for Full System Simulation," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 1, pp. 52–61, 2009.

[3] E. Ayguadé *et al.*, "The Design of OpenMP Tasks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 3, pp. 404–418, 2009.

[4] C. Bienia, "Benchmarking Modern Multiprocessors," Ph.D. dissertation, Princeton University, 2011.

[5] N. Binkert *et al.*, "The gem5 Simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[6] D.R. Butenhof, *Programming with POSIX Threads*. Addison-Wesley Professional, 1997.

[7] T.E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-core Simulation," in *Proc. 24th Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 52:1–52:12.

[8] T.E. Carlson, W. Heirman, and L. Eeckhout, "Sampled Simulation of Multi-Threaded Applications," in *Proc. 2013 IEEE Int'l Symp. on Performance Analysis of Systems and Software*, 2013, pp. 2–12.

[9] T.E. Carlson *et al.*, "An Evaluation of High-Level Mechanistic Core Models," *ACM Transactions on Architecture and Code Optimization*, vol. 11, no. 3, p. Article No. 28, 2014.

[10] T.E. Carlson *et al.*, "BarrierPoint: Sampled Simulation of Multi-Threaded Applications," in *Proc. 2014 IEEE Int'l Symp. on Performance Analysis of Systems and Software*, 2014, pp. 2–12.

[11] M. Casas, R.M. Badia, and J. Labarta, "Automatic Phase Detection and Structure Extraction of MPI Applications," *Int'l J. of High Performance Computing Applications*, vol. 24, no. 3, pp. 335–360, 2010.

[12] M. Casas *et al.*, "Runtime-Aware Architectures," in *Proc. 21st Int'l European Conf. on Parallel and Distributed Computing*, 2015, pp. 16–27.

[13] M. Casas *et al.*, "Extracting the Optimal Sampling Frequency of Applications Using Spectral Analysis," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 3, pp. 237–259, 2012.

[14] D. Chasapis *et al.*, "PARSECSs: Evaluating the Impact of Task Parallelism in the PARSEC Benchmark Suite," *ACM Trans. on Architecture and Code Optimization*, vol. 12, no. 4, pp. 1–22, 2015.

[15] T.M. Conte, M.A. Hirsch, and W.M.W. Hwu, "Combining Trace Sampling with Single Pass Methods for Efficient Cache Simulation," *IEEE Trans. Comput.*, vol. 47, no. 6, pp. 714–720, 1998.

[16] D. Eklov and E. Hagersten, "StatStack: Efficient Modeling of LRU Caches," in *Proc. 2010 IEEE Int'l Symp. on Performance Analysis of Systems and Software*, 2010, pp. 55–65.

[17] M. Ester *et al.*, "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise," in *Proc. 2nd Int'l Conf. on Knowledge Discovery and Data Mining*, 1996, pp. 226–231.

[18] A. Falcón, P. Faraboschi, and D. Ortega, "Combining Simulation and Virtualization Through Dynamic Sampling," in *Proc. 2007 IEEE Int'l Symp. on Performance Analysis of Systems and Software*, 2007, pp. 72–83.

[19] D. Genbrugge, S. Eyerman, and L. Eeckhout, "Interval Simulation: Raising the Level of Abstraction in Architectural Simulation," in *Proc. 16th IEEE Symp. on High Performance Computer Architecture*, 2010, pp. 1–12.

[20] J. Gonzalez *et al.*, "Simulating Whole Supercomputer Applications," *IEEE Micro*, no. 3, pp. 32–45, 2011.

[21] J. Gonzalez, J. Gimenez, and J. Labarta, "Automatic Detection of Parallel Applications Computation Phases," in *Proc. 32nd IEEE Int'l Parallel and Distributed Processing Symp.*, 2009, pp. 1–11.

[22] J.W. Haskins Jr and K. Skadron, "Memory Reference Reuse Latency: Accelerated Warmup for Sampled Microarchitecture Simulation," in *Proc. 2003 IEEE Int'l Symp. on Performance Analysis of Systems and Software*, 2003, pp. 195–203.

[23] T. Karkhanis and J. Smith, "A First-Order Superscalar Processor Model," in *Proc. 31st Ann. Int'l Symp. on Computer Architecture*, 2004, pp. 338–349.

[24] J. Lau *et al.*, "The Strong Correlation Between Code Signatures and Performance," in *Proc. 2005 IEEE Int'l Symp. on Performance Analysis of Systems and Software*, 2005, pp. 236–247.

[25] K. Lee, S. Evans, and S. Cho, "Accurately Approximating Superscalar Processor Performance from Traces," in *Proc. 2009 IEEE Int'l Symp. on Performance Analysis of Systems and Software*, 2009, pp. 238–248.

[26] Y. Luo, L.K. John, and L. Eeckhout, "Self-Monitored Adaptive Cache Warm-Up for Microprocessor Simulation," in *Proc. 16th Symp. on Computer Architecture and High Performance Computing*, 2004, pp. 10–17.

[27] J. MacQueen *et al.*, "Some Methods for Classification and Analysis of Multivariate Observations," in *Proc. 5th Berkeley Symp. on Mathematical Statistics and Probability*, vol. 1, no. 14, 1967, pp. 281–297.

[28] S.D. Pestel, S. Eyerman, and L. Eeckhout, "Micro-Architecture Independent Branch Behavior Characterization," in *Proc. 2015 IEEE Int'l Symp. on Performance Analysis of Systems and Software*, 2015, pp. 135–144.

[29] N. Rajovic *et al.*, "Supercomputing with Commodity CPUs: Are Mobile SoCs Ready for HPC?" in *Proc. 26th Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12.

[30] A. Rico *et al.*, "On the Simulation of Large-Scale Architectures Using Multiple Application Abstraction Levels," *ACM Trans. on Architecture and Code Optimization*, vol. 8, no. 4, pp. 36:1–36:20, 2012.

[31] A. Rico *et al.*, "Trace-Driven Simulation of Multithreaded Applications," in *Proc. 2011 IEEE Int'l Symp. on Performance Analysis of Systems and Software*, 2011, pp. 87–96.

[32] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," in *Proc. 40th Ann. Int'l Symp. on Computer Architecture*, 2013, pp. 475–486.

[33] T. Sherwood *et al.*, "Automatically Characterizing Large Scale Program Behavior," in *Proc. 10th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2002, pp. 45–57.

[34] M. Valero *et al.*, "Runtime-Aware Architectures: A First Approach," *Int'l J. on Supercomputing Frontiers and Innovations*, vol. 1, no. 1, pp. 29–44, Jun. 2014.

[35] S. Van Den Steen *et al.*, "Analytical Processor Performance and Power Modeling Using Micro-Architecture Independent Characteristics," *IEEE Trans. Comput.*, vol. 65, no. 12, pp. 3537–3551, 2016.

[36] T.F. Wenisch *et al.*, "TurboSMARTS: Accurate Microarchitecture Simulation Sampling in Minutes," in *Proc. ACM SIGMETRICS Int'l Conf. on Measurement and Modeling of Computer Systems*, 2005, pp. 408–409.

[37] T.F. Wenisch *et al.*, "Simulation Sampling with Live-Points," in *Proc. 2006 IEEE Int'l Symp. on Performance Analysis of Systems and Software*, 2006, pp. 2–12.

[38] R.E. Wunderlich *et al.*, "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling," in *30th Annual International Symposium on Computer Architecture*, 2003, pp. 84–95.