TaskPoint: Sampled Simulation of Task-Based Programs

Thomas Grass[*†], Alejandro Rico[‡], Marc Casas[†], Miquel Moreto[*†], Eduard Ayguadé[*†]
[*]Universitat Politècnica de Catalunya, [†]Barcelona Supercomputing Center, [‡]ARM Inc.

***Abstract*—Sampled simulation is a mature technique for reducing simulation time of single-threaded programs, but it is not directly applicable to simulation of multi-threaded architectures. 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 system noise and 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-forward mechanism for dynamically scheduled programs. We evaluate the proposed technique on a set of 19 task-based parallel benchmarks and two different architectures. Compared to detailed simulation, TaskPoint accelerates architectural simulation with 64 simulated threads by an average factor of 19.1 at an average error of 1.8% and a maximum error of 15.0%.

I. Introduction

Computer architecture research heavily relies on simulation. Increasing design complexity and increasing core counts in modern multi-core processors present new challenges to architectural simulation. First, simulating a more complex design requires more time for a given workload. Second, the more complex a design, the larger the simulated workload needs to be in order to meaningfully stress the design.

One technique to reduce simulation time is sampling. Sampled simulation reduces simulation time by only simulating a fraction of a workload. Sampling is a well-established technique for simulation of single-threaded architectures. The prevalent techniques perform detailed simulation of either only the representative program parts identified in profiling [1] or periodically via time-based sampling [2].

While sampled simulation is a well-established technique for single-threaded architectures, techniques targeting multi-threaded architectures have only been recently proposed. The main challenge in sampling multi-threaded simulations 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 Carlson et al. [3] achieves this by selecting a periodic sampling interval during offline profiling and, during simulation, estimating the rate at which to fast-forward each thread between intervals of detailed simulation. Carlson et al. [4] 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.

Task-based programming models have been proposed to reduce load imbalance and thus increase parallel efficiency of future large-scale multi-core machines [5]. A task-based programming model allows the programmer to specify 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 there are execution threads. The over-decomposition of a parallel program into tasks, together with dynamic scheduling of task instances to threads, dynamically balances the amount of work assigned to each thread. Inter-task dependencies enforce synchronization only when necessary. The lack of global barriers and the dynamically scheduled execution of task-based programs make them unsuitable for existing sampled simulation techniques.

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

In this paper, we make the following contributions:

- We compare the performance variation of task-based programs in native execution and architectural simulation. This motivates the design of our TaskPoint methodology, its sampling policies and its fast-forwarding methodology.
- We present TaskPoint, a sampled simulation technique for multi-core architectures programmed with a dynamically scheduled, task-based programming model. In this context, we introduce two sampling policies, *periodic sampling* and *lazy sampling*. Lazy sampling simulates task instances in detail based on their type while periodic sampling considers their type and distribution over time.
- We propose a mechanism to accurately fast-forward an architectural simulation of a task-based program. During fast-forward, we model the performance of a given task instance based on previous instances of the same task type. We account for different task input sizes across the application execution by factoring in the number of instructions of the given task instance accordingly.

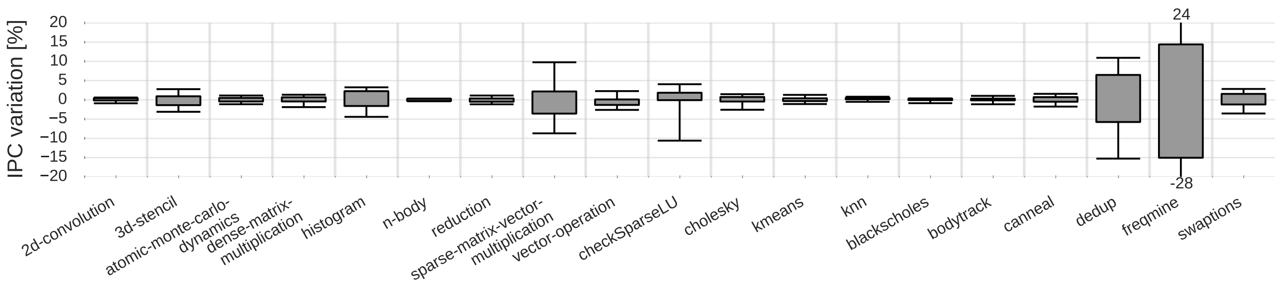


Fig. 1: IPC variation across all task instances for native execution with 8 threads, normalized per task type

- We evaluate TaskPoint simulating 19 task-based parallel benchmarks, including 6 task-based versions of the PARSEC benchmark suite. We evaluate the sensitivity of TaskPoint to different architectures by testing different numbers of simulated threads on two different configurations covering the opposite extremes of the multi-core design space: high-performance and low power.

The remainder of this paper is organized as follows. In Section II, we provide background and motivation of our work. In Section III, we present our TaskPoint methodology. Next, we introduce our experimental setup in Section IV. We evaluate TaskPoint in Section V. Finally, we present related work in Section VI, before we conclude in Section VII.

II. 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 19 task-based parallel benchmarks.

A. 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 environment dynamically schedules task instances to execution threads.

Due to a fine-grained *over-decomposition* of the application, there are ideally more task instances ready for execution than there are threads. This allows the runtime environment to dynamically balance the workload assigned to each thread [5]. Further optimizations are possible if the architecture interfaces directly with the runtime environment [7, 8].

In this work, we differentiate between *task types* and *task instances*. Every execution of a task declaration statement at runtime 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 small, whereas the number of task instances lies in the order of thousands.

B. Performance Variation of Task-Based Programs

In order to motivate TaskPoint, our sampled simulation technique for task-based parallel programs, we analyze performance variation in native execution of 19 benchmarks. The investigated benchmarks are introduced in Section IV.

Different benchmarks, and even different task types of the same benchmark, generally show different average instructions per cycle (IPC). For an easy comparison of performance variation across benchmarks, we normalize the IPC of all task instances to the average IPC of their respective task type. For each benchmark, we use one box plot of these normalized IPC values to visualize performance variation across task instances.

Figure 1 shows IPC variation across task instances observed in a 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. 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 15 out of 19 benchmarks performance variation lies within $\pm 5\%$. We show that performance variation is closely reflected in simulation when we introduce the TaskSim simulator in Section IV.

We motivate TaskPoint based on the insight that performance of task-based programs is generally regular across instances of the same task type. A novelty of TaskPoint is that it leverages the concept of tasks declared by the programmer to identify task instances of the same task type as sampling units of similar performance.

III. Sampled Simulation of Task-Based Programs

In this section, we present our TaskPoint methodology. First, we introduce the prerequisites which need to be fulfilled by an architectural simulator in order to serve as an implementation platform for TaskPoint. Next, we present the different

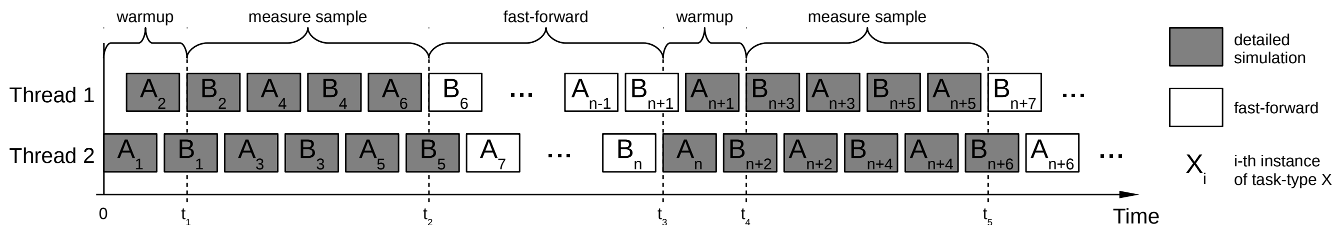


Fig. 2: Initial warmup, sampling, fast-forwarding and resampling in TaskPoint

phases of TaskPoint's sampling mechanism, namely warm-up, sampling and fast-forwarding. Afterwards, we introduce our periodic sampling policy. The separation into sampling mechanism and policy allows for the integration of other sampling policies with low implementation effort.

A. 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. In order to serve as a suitable platform for implementing our methodology, a simulator needs to fulfil the following two requirements:

1) The simulator needs to feature a detailed and a fast simulation mode.
2) The fast mode has to be capable of operating at a user-specified IPC.

Most contemporary architectural simulators feature several levels of detail [9, 10, 11], allowing to trade off speed for accuracy. Thus, we assume the first requirement to be trivially fulfilled. Regarding the second requirement, if a simulator does not support fixed-IPC simulation by default, we consider the implementation of this functionality to be a minor effort.

B. Sampling Mechanism

TaskPoint operates on the level of granularity of task instances. A task instance is simulated either in detailed or in fast mode. Simulation in detailed mode serves for warming architectural state or to measure samples, whereas simulation in fast mode accurately fast-forwards simulation time. Switching between detailed and fast mode only occurs between two consecutive task instances.

Figure 2 illustrates the different phases of TaskPoint. For each task type, 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*. While the former is the sample history we usually use to determine which IPC to use in fast mode, the latter is needed if there are task types that occur infrequently and can not be sampled in a single sampling interval. We refer to these task types as *rare task types*.

In multi-threaded applications, co-existing threads 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 thread interference, we simulate all threads either in detailed mode or in fast 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-forward mode, while others are simulated in detailed mode (see t_2, t_3 and t_5 in Figure 2).

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 [1, 2, 12, 13, 14, 15]. 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. However, we distinguish between warming at simulation start and warming before resampling after a simulation phase in fast mode. 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 V.

After a simulation phase in fast 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 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 mode when one of the following two conditions is fulfilled:

1) The history of valid samples is fully populated.

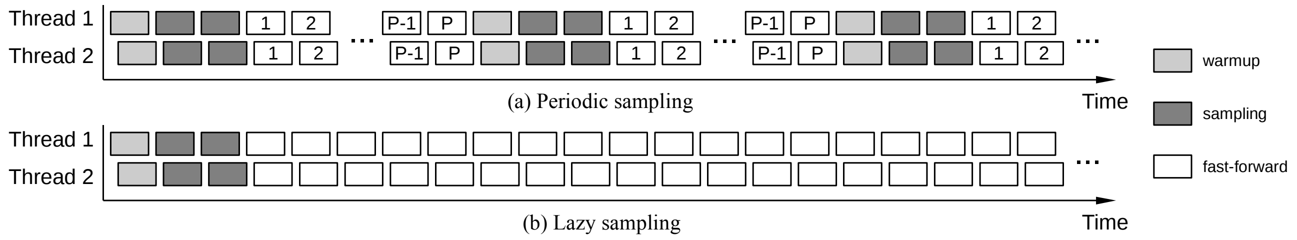


Fig. 3: Illustration of periodic sampling (a) and lazy sampling (b) as a special case of periodic sampling with infinite sampling period P

2) A certain number of task instances has been simulated without encountering any instance of a rare task type whose history of valid samples is not yet fully populated.

The first condition means that all task types are fully sampled. The second condition is needed to avoid spending an excessive amount of time on detailed simulation in the presence of rare task types. In this paper, we cut off sampling when all threads have simulated 5 task instances without encountering an instance of a previously observed rare task type.

Accurate Fast-Forwarding: When the transition to fast mode is triggered, all task instances starting in the future are simulated in fast mode. However, task instances which started in the past are simulated in detailed mode until they complete. Task instances finishing simulation after the transition to fast mode are only added to the history of all samples.

A task instance simulated in fast mode is simulated with the average IPC of the history of valid samples of its task type. If a task instance belongs to a rare task type 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 task type is also empty, we trigger resampling.

Rare task types 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 task types to be negligible.

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 task type of the task instance currently being simulated.

C. Periodic Sampling Policy

A sampling policy decides when to resample a simulation running in fast-forward mode. The *periodic sampling* policy, illustrated in Figure 3a, warms and samples a simulation at simulation start. Afterwards, it switches the simulation to fast-forward mode. When a thread has executed a number P of task instances of any task type in fast-forward 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-forward mode and the process repeats.

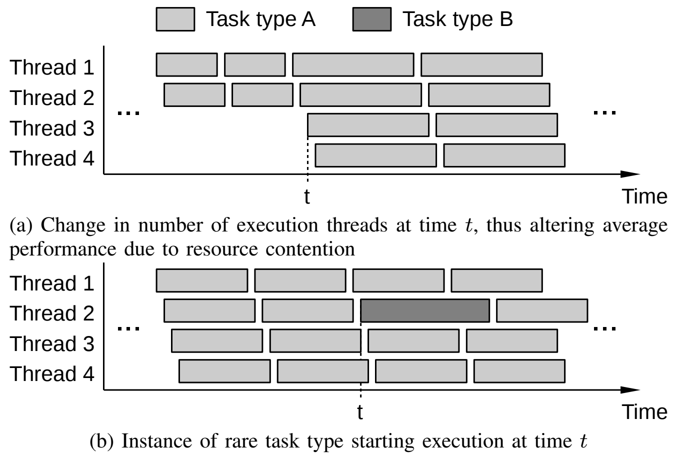

Task type A
Task type B
Thread 1
Thread 2
Thread 3
Thread 4
t
Time
(a) Change in number of execution threads at time t, thus altering average performance due to resource contention
(b) Instance of rare task type starting execution at time t

Fig. 4: Illustration of changing number of execution threads (a) and rare task type (b)

Simulation speedup is determined by the size of the sampling period. The larger the sampling period, the more task instances are simulated in fast 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 3b. 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-forward 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 mode, resampling is also triggered when it is impossible to accurately simulate a task instance in fast mode. This happens in the following two cases.

Figure 4a 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 4b shows a case where the first instance of a new task type is encountered while simulating in fast mode. When encountering an instance of a previously unknown task type, the task type's sample history is empty. Therefore, it is impossible to simulate this task instance in fast 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 is implemented with different task types, the simulation is resampled. The same holds for changes in the available computation resources or the available parallelism.

IV. 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 19 benchmarks and the two architectures we use in our evaluation. Finally, we elaborate on the TaskSim simulator and our implementation of fast simulation at arbitrary IPC and show an analysis of performance variation observed in simulation of task-based programs.

The OmpSs Programming Model: For our evaluations we choose the OmpSs programming model [16]. The OmpSs compiler and runtime environment are available as open source. 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 were included into the specifications of OpenMP 3.0 and 4.0.

Benchmarks: Table I 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*, *freqmine* and *swaptions* are part of the PARSEC benchmark suite [17]. Whenever possible, we generated traces equivalent to 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 I lists the number of task instances and the time required for a detailed simulation of the entire benchmark for 1 and 64 execution threads using the TaskSim simulator. TaskSim is introduced later in this section.

Simulated Architectures: We evaluate the fidelity of our methodology by investigating simulation speedup and execution time error of multi-threaded simulations of two radically different multi-core architectures. One resembles a server-class system, while the other resembles a low-power mobile platform. Table II 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 [18].

The TaskSim Simulator: We evaluate our methodology using the TaskSim simulator [19, 20]. 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 simulation mode. The detailed mode is based on the *Reorder-Buffer Occupancy Analysis* model proposed by Lee et al. [21]. 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 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. In the implementation of our fast-forward mechanism, the duration of a task instance is calculated at the beginning of its execution. Using the mean IPC of the sample history of a task instance i's task type 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 detailed and fast-forward mode at runtime. We also extended its fast simulation mode. Instead of using previously recorded cycle counts from a trace, our implementation of fast mode uses cycle counts predicted by our fast-forward mechanism. To the best of our knowledge, this is the first fast-forward mechanism applying different IPCs to different parts of a program. Our mechanism allows fast-forwarding dynamically scheduled parallel programs in which the per-thread instruction stream is a-priori unknown. Next, we evaluate performance variation of task-based programs observed in simulation with TaskSim.

Figure 5 shows IPC variation across task instances in an architectural simulation with 8 execution threads. The parameters of the simulated architecture match the machine used for native execution, as far as they are publicly available. All benchmarks showing a performance variation of less then $\pm 5\%$ in native execution (see Figure 1) also do so in simulation. Conversely, three out of the four benchmarks showing a variation larger than $\pm 5\%$ in native execution also do so in simulation. The exception is *sparse-matrix-vector-multiplication*, which in native execution exhibits a variation of nearly $\pm 10\%$, compared to less than $\pm 5\%$ in simulation. The three benchmarks with the largest degree of performance variation in native execution, namely *checkSparseLU*, *dedup* and *freqmine*, also show the largest variation in simulation.

Due to modelling inaccuracies in TaskSim's detailed simulation mode, the magnitudes of performance variation in native execution and simulation do not match exactly. However, for 18 out of 19 benchmarks we correctly identify if a benchmark exposes a performance variation of more or less than 5%.

V. Evaluation

In this section, we conduct a sensitivity analysis of TaskPoint's model parameters. Then, we evaluate execution time error and simulation speedup of periodic sampling and lazy

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

Benchmark	**# Task Types**	**# Task Instances**	**Simulation time** $[h : min]$ **1 Thread**	**64 Threads**	**Properties**
2d-convolution	1	16384	31:37	59:34	Kernel: strided memory accesses
3d-stencil	1	16370	9:12	40:51	Kernel: strided memory accesses
atomic-monte-carlo-dynamics	1	16384	8:38	15:16	Kernel: embarrassingly parallel
dense-matrix-multiplication	1	17576	70:14	127:10	Kernel: high data reuse, compute bound
histogram	1	16384	6:02	12:13	Kernel: atomic operations
n-body	2	25000	8:15	12:31	Kernel: irregular memory accesses
reduction	2	16384	1:51	5:15	Kernel: parallelism decreases over time
sparse-matrix-vector-multiplication	1	1024	0:33	1:26	Kernel: load imbalance, memory bound
vector-operation	1	16400	24:25	191:00	Kernel: regular, memory bound
checkSparseLU	11	22058	7:25	17:17	Decomposition of large, sparse matrices
cholesky	4	19600	33:42	59:29	Decomposition of Hermitian positive-definite matrices
kmeans	6	16337	75:21	141:02	Clustering based on Lloyd's algorithm
knn	2	18400	31:28	65:27	Instance-based machine learning algorithm
blackscholes	2	24500	8:42	17:19	Option price calculation
bodytrack	7	21439	15:24	31:28	Human body tracking with multiple cameras
canneal	1	16384	11:13	29:38	Cache-aware simulated annealing
dedup	4	15738	10:08	23:32	Deduplication: combination of global and local compression
freqmine	7	1932	23:52	34:13	Frequent Pattern Growth method for Frequent Item Mining
swaptions	1	16384	29:27	70:25	Monte-Carlo simulation to calculate swaption prices

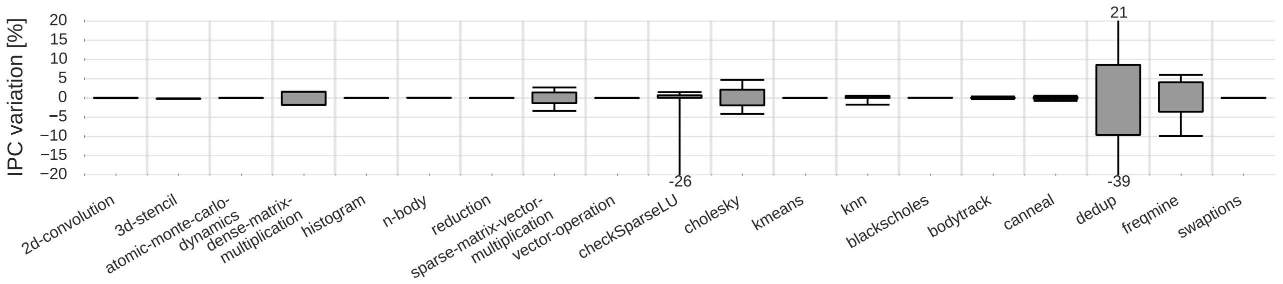


Fig. 5: IPC variation across all task instances for simulation of high-performance architecture with 8 threads, normalized per task type

TABLE II: Architectural parameters of high performance and mobile configurations used for model validation

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

sampling. Finally, we test the robustness of our model by using the same parameters to simulate a low-power architecture.

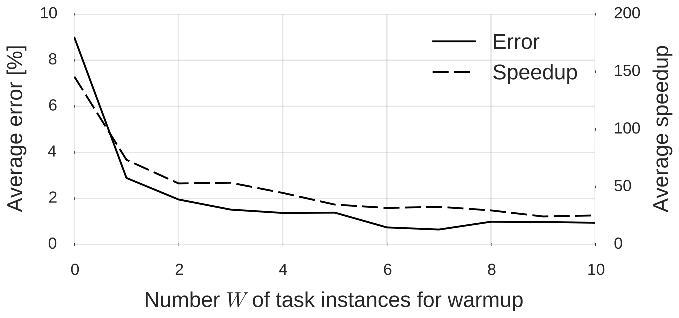
A. Adjusting the Model Parameters

We determine the optimal model parameters following an incremental approach. First, we determine the optimal number W of task instances needed for warmup at simulation start. Afterwards, we consider different numbers of task instances H constituting the sample history. Finally, we explore a range of values for the sampling period P.

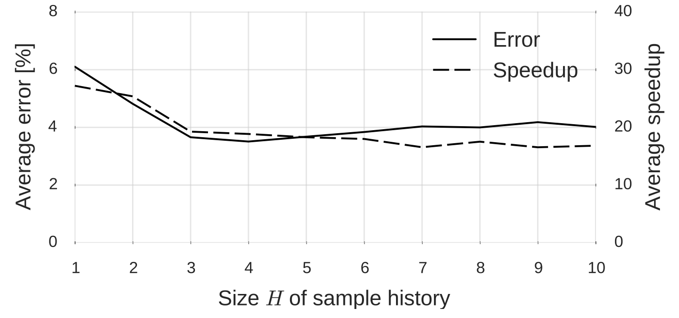
In order to determine the optimal value for W we set $H = 10$ and $P = \infty$ and evaluate different values ranging from $W = 0$ (no warmup) to $W = 10$. Figure 6a shows error and speedup, averaged over simulations with 32 and 64 threads. The reported values are averaged over the benchmarks and kernels with an error $> 5\%$ for at least one value of H, namely *2d-convolution*, *3d-stencil*, *atomic-monte-carlo-dynamics*, *knn* and *blackscholes*. 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 they reduce simulation speedup. 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 we already set $W = 2$. Figure 6b 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 $H = 4$.

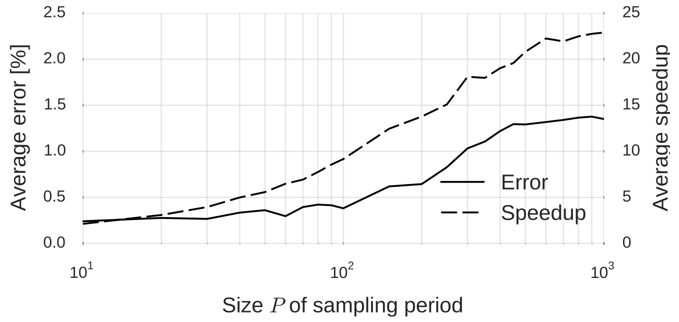
Finally, we explore different sizes of the sampling period P. With $W = 2$ and $H = 4$ already fixed, P is the only remaining parameter. Figure 6c shows the average error for values of P ranging from 10 to $1,000$. We find that average error



(a) Error and speedup for different sizes W of warmup interval, average of 32 and 64 threads



(b) Error and speedup for different sizes H of task instance history, average of 32 and 64 threads



(c) Error and speedup for different sizes P of sampling period, average of 32 and 64 threads

Fig. 6: Error and speedup for different sizes of warmup interval (a), sample history (b) and sampling period (c)

and speedup increase with the size of the sampling period. The larger the value of P, more task instances are simulated in fast 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 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 $P = \infty$).

B. Periodic Sampling

First, we evaluate periodic sampling, simulating the high-performance architecture in Table II, which we also use to find the sampling parameters. Afterwards, we simulate the low-power architecture using the same sampling parameters.

High-Performance Architecture: Figure 7 shows execution time error and simulation speedup for all investigated benchmarks, simulated with the parameters $W = 2$, $H = 4$ and $P = 250$. The average execution time error is less than 2% for 8, 16, 32 and 64 simulated threads. The error for 1, 2 and 4 simulated threads is less than 1% and not shown in the Figure. We observe the largest simulation speedup of 76.2 for *sparse-matrix-vector-multiplication* executed with 8 threads.

We observe the highest error of 8.9% in the simulation of *freqmine* with 8 threads. *Freqmine* consists of 7 different task types, one of which accounts for 93% of the total number of dynamic instructions. The dynamic instruction count of the instances of this task type ranges from 490 to 11,000,000. Inspecting the source code reveals a construct of nested `if`-statements in a task declaration. This causes different instances of the same task type to follow completely unrelated control flow paths. The unbalanced size across task instances makes sampling the simulations with 32 and 64 threads ineffective. Since these configurations are simulated almost entirely in detail, the error is negligible and speedup is close to 1.

From this finding, we derive a recommendation to programmers for improving performance predictability of task-based programs: One should avoid large-scale control flow divergence among instances of the same task type. In practice, this is achieved by declaring code performing unrelated work as different task types.

The second largest error of 7.3% is shown by *dedup* for 64 threads. *Dedup* consists of 4 task types, one of which accounts for 99.9% of the dynamic instruction count. The dynamic instruction count of the instances of this task type ranges from 3,500,000 to 25,100,000. The dominating task type performs de-duplication as well as compression, which are highly input dependent operations. Previous work identified input dependence as a source of performance variation [22]. Performance variation makes it difficult to determine a task type's average performance during sampling.

We recognize that, in certain cases, input dependence can not be avoided. One way to improve the accuracy of sampled simulation of programs showing input dependence is to classify task instances into classes of similar performance. We envision clustering of instances of the same task type based on micro-architecture independent metrics, e.g. instruction count or instruction mix. We leave this for future work.

Next, we evaluate the generalization capability of periodic sampling. We simulate a low-power architecture which is radically different from the high-performance architecture we used to determine the sampling parameters.

Low-Power Architecture: Figure 8 shows execution time error and simulation speedup for simulations of all benchmarks executed on the low-power architecture introduced in Table II with 1, 2, 4 and 8 threads. We notice that, for increasing

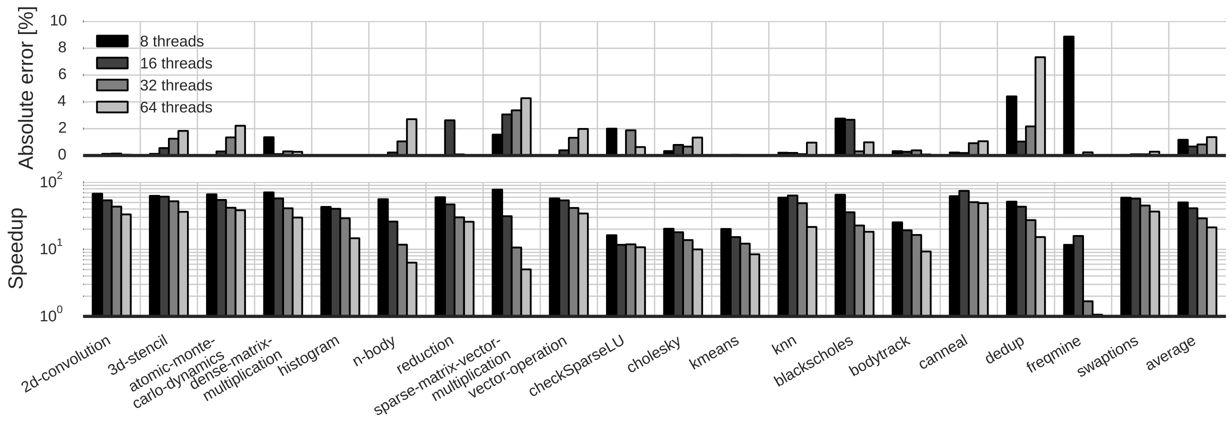


Fig. 7: Error and speedup of periodic sampling; high-performance architecture; $P = 250$

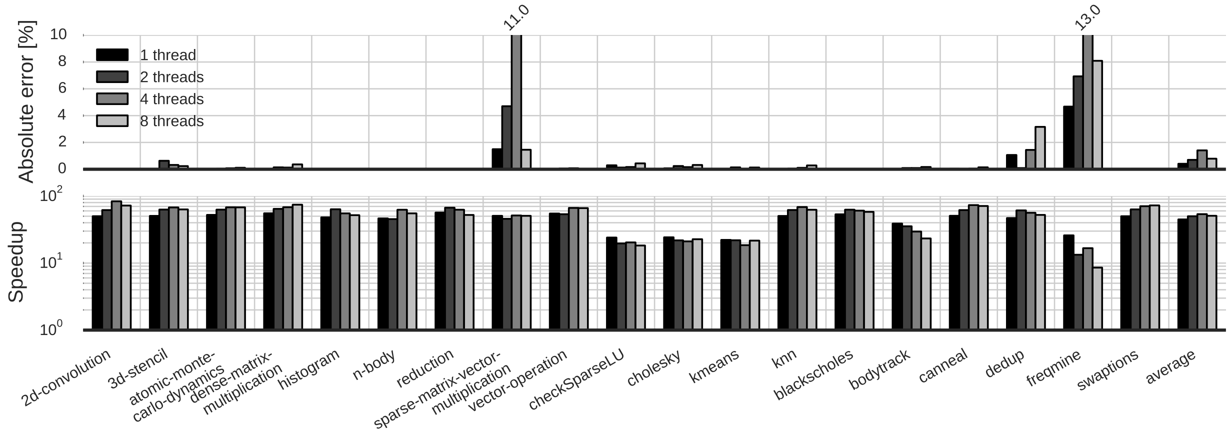


Fig. 8: Error and speedup of periodic sampling; low-power architecture; $P = 250$

thread counts, speedup degrades less than in the case of the high-performance architecture. Since we simulate smaller thread counts, the simulation is resampled more often and the percentage of task instances simulated in fast mode is more similar across different thread counts.

With an error of 13.0% for 4 threads, *freqmine* is the benchmark with the highest error. This is consistent with the simulation of the high-performance architecture. We attribute this error to the same reason as in the case of the high-performance architecture, namely the highly imbalanced size of the instances of the dominant task type.

We observed the second largest error of 8.4% for *sparse-matrix-vector-multiplication* with 8 threads. Depending on the structure of the input matrix, memory accesses are more or less regular [23]. We conclude that, due to the two-level cache hierarchy, the smaller last-level cache and the lower memory bandwidth, this has a higher impact on performance variation than in the high-performance architecture. This is another example of input dependence, similar to the case of *dedup* explained in the previous section.

C. Lazy Sampling

For our evaluation of lazy sampling, we set $W = 2$, $H = 4$ and $P = \infty$. We simulate the benchmarks listed in Table I executing on the high performance architecture and the low-power architecture listed in Table II.

High-Performance Architecture: Figure 9 shows execution time error and simulation speedup of the lazy sampling policy for the investigated benchmarks executed on the high-performance architecture. The average error is less than 2% for all simulated thread counts (including 1, 2, and 4 threads, which are not shown in the Figure).

Dedup and *freqmine* are still the benchmarks showing the highest error. Compared to periodic sampling, the highest observed error of *dedup* increases from 7.3% to 15.0% for the simulation with 64 threads. In the case of *freqmine*, the highest observed error increases from 8.9% to 9.6% for the simulation with 8 threads.

While the average error of lazy sampling is comparable to the error of periodic sampling, we observe a significant increase of average simulation speedup. Compared to periodic sampling, we observe the largest increase from 44.4 to 178.5 for the average speedup of the simulations with 8 threads. The smallest gain in speedup is observed for the simulations with 64 threads, in which speedup increases from 15.8 to 19.1. For 1 thread, which is not shown in the Figure, speedup increases from 43.2 to 1019.

Low-Power Architecture: Figure 10 shows execution time error and simulation speedup for the low-power architecture. We observe a marginal increase of the maximum error of *sparse-matrix-vector-multiplication* and *freqmine*, the bench-

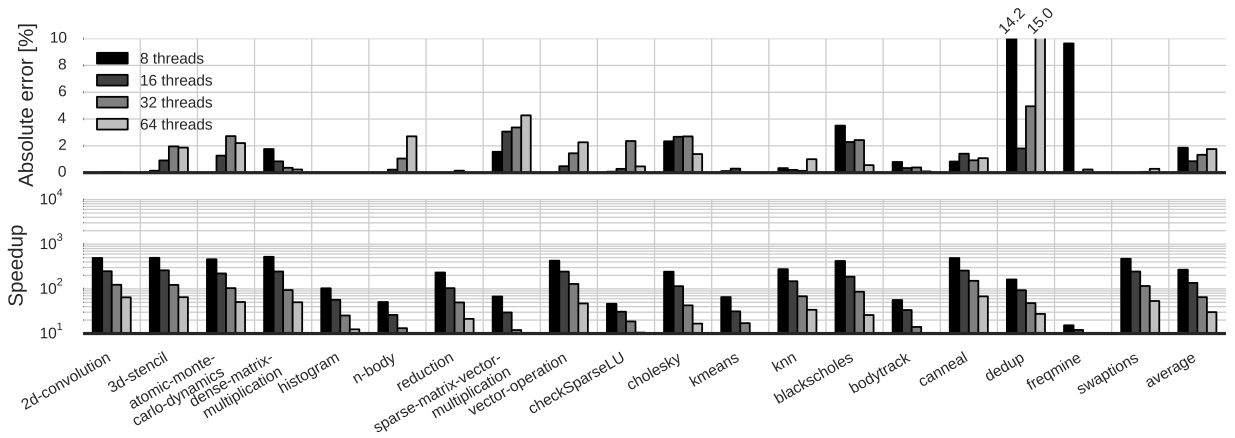


Fig. 9: Error and speed-up of lazy sampling; high-performance architecture

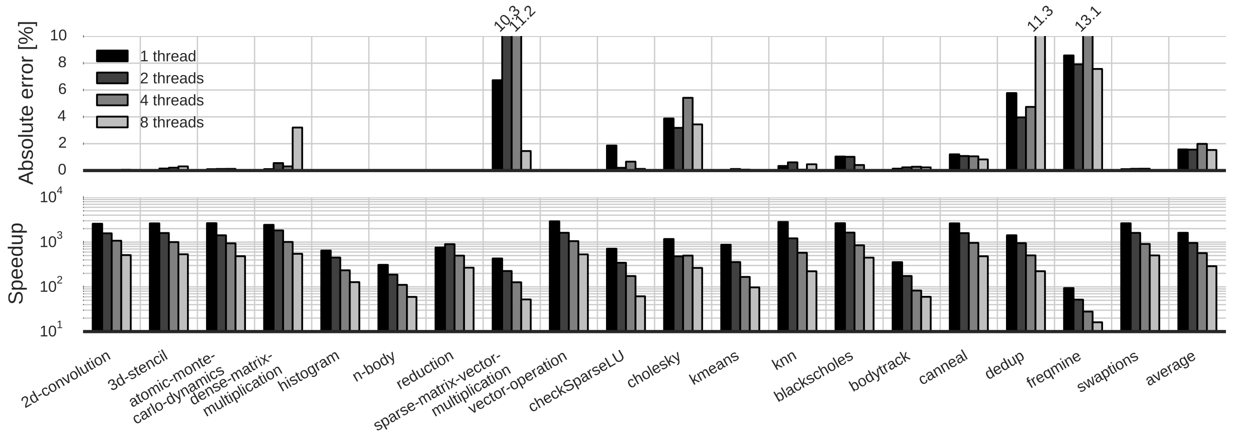


Fig. 10: Error and speed-up of lazy sampling; low-power architecture

marks with the largest errors in the simulations of the low-power architecture employing periodic sampling. However, in the case of *dedup*, the error increases for all simulated thread counts. We observe the highest increase, from 3.2% to 11.3%, for the simulation with 8 threads.

Summary: The results of our evaluation show that Task-Point accurately predicts execution time of task-based programs. For lazy sampling, the average error is 1.8% with a maximum error of 15% and a simulation speedup of 19.1. We show that lazy sampling achieves much greater speedup than periodic sampling at a comparable error. Therefore, we advocate the use of lazy sampling for evaluations requiring a large number of simulations, e.g. during the early phase of design space exploration. We recommend to employ periodic sampling in later phases of design space exploration when the size of the design space has already been significantly reduced.

VI. Related Work

In this section, we first introduce different 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. Finally, we present work on performance analysis of task-based programs.

Multi-Threaded Architectural Simulation: *COTSon* [10] is a full-system simulator decoupling functional and timing simulation. Functional simulation relies on just-in-time compilation of the simulated program. COTSon features several levels of detail and supports sampling.

In addition to performance, ESESC [24] also simulates a future design's power consumption and thermal behaviour. ESESC is the first simulator applying time-based sampling to simulation of multi-threaded applications.

The full-system simulator *gem5* [11] features CPU models at several levels of detail, ranging from a model employing native execution to a detailed model of a superscalar out-of-order CPU. Besides others, gem5 supports the x86 and ARM architectures, which are the most prevalent architectures today.

In contrast to the aforementioned simulators, *Sniper* [25] features a purely analytic CPU model. Instead of modelling micro-architectural structures within the CPU, it employs the mechanistic *Interval Simulation* model [26]. The higher level of abstraction of interval simulation is directly reflected in a higher simulation speed, compared to more detailed models.

Single-Threaded Simulation Sampling: In their *SimPoint* methodology [1], Sherwood et al. use basic block vectors to identify the most representative code sections. The major simulation effort is spent on these sections SimPoints requires a-priori profiling of the application to be simulated in order to identify basic block vectors.

SMARTS [2] and *TurboSMARTS* [27] switch periodically between warmup, detailed simulation and fast-forward. Warmup makes sure that simulated micro-architectural structures are in a representative state. After warmup, the performance metrics of interest are measured in detailed mode. Fast-forward mode only performs functional simulation maintaining the correct architectural state of the simulated program. The durations of the respective intervals are user-specified parameters.

Multi-Threaded Simulation Sampling: There are recent techniques applying sampling to simulations of multi-threaded programs. Carlson et al. [3] apply time based sampling [28] to parallel programs. Short detailed simulation phases take turns with longer fast-forward phases, resulting in an overall reduction of simulation time. The fast-forward 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-forward intervals are determined during profiling using micro-architecture independent metrics.

BarrierPoint [4], also proposed by Carlson et al., first analyzes micro-architecture independent performance metrics of program sections between global barriers. Afterwards, the SimPoint infrastructure [1] 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 24.7 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. [29] identify representative phases (*CPU bursts*) of programs implemented in the Message Passing Interface (MPI) programming model. These 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.

In dynamically scheduled task-based programs, the workload processed by each thread varies across different executions. This makes it impossible to statically determine the representative sections of a program. Therefore, the existing sampling techniques for multi-threaded applications are not directly applicable to task-based programming models.

Warming in Multi-Threaded Simulations: Warming for single-threaded simulations has been extensively studied [1, 2, 12, 13, 14, 30]. The technique used by the BarrierPoint methodology combines two existing methodologies, namely functional warming [13] and checkpointing [14]. 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.

Luo et al. [15] propose *Self-Monitored Adaptive Cache Warm-Up* (SMA), a technique not requiring profiling prior to simulation. Every cache in a simulated system monitors its fraction of used lines over time. When this fraction passes a threshold or remains constant during a certain time, a cache is considered warmed. The authors evaluate SMA for single-threaded simulations. However, there are no fundamental reasons impeding its applicability to multi-threaded simulations.

Performance Variation in Task-Based Programs: Recent work [22] shows that execution time of task-based programs is predictable. Generally, instances of the same task type show similar execution time and performance. If task instances compete for shared resources in multi-threaded executions, their execution time increases. Contention on shared resources also increases performance variation across task instances. Olivier et al. [31] investigate the effect of increasing execution time of task instances in case of resource sharing. They refer to the effect as *work time inflation*.

VII. Conclusions

Previous sampled simulation techniques for parallel programs rely on profiling to identify the parameters of the sampling mechanism. Although those techniques have been proven to be accurate for statically scheduled fork-join based programs, they are not directly applicable to 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-forward 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.

We assessed TaskPoint's generalization capability by using two radically different architectures to select sampling parameters and to run simulations. The evaluation results are satisfactory across a wide range of benchmarks, different numbers of simulated threads and different architecture models. The average simulation error is less than 2% at an average speedup ranging from 19× for 64 threads to 1019× for 1 thread.

Acknowledgments

This work has been supported by the Spanish Government (Severo Ochoa grants SEV2015-0493, SEV-2011-00067), 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), the European HiPEAC Network of Excellence 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] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. *ACM SIGOPS Operating Systems Review*, 36(5):45–57, 2002.

[2] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. Smarts: accelerating microarchitecture simulation via rigorous statistical sampling. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 84–95, 2003.

[3] Trevor E Carlson, Wim Heirman, and Lieven Eeckhout. Sampled simulation of multi-threaded applications. In *Performance Analysis of Systems and Software, 2013 IEEE International Symposium on*, pages 2–12, 2013.

[4] Trevor E Carlson, Wim Heirman, Kenzo Van Craeynest, and Lieven Eeckhout. Barrierpoint: Sampled simulation of multi-threaded applications. In *Performance Analysis of Systems and Software, 2014 IEEE International Symposium on*, pages 2–12, 2014.

[5] Jonathan Lifflander, Sriram Krishnamoorthy, and Laxmikant 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.

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

[7] Marc Casas, Miquel Moretó, Lluc Alvarez, Emilio Castillo, Dimitrios Chasapis, Timothy Hayes, Luc Jaulmes, Oscar Palomar, Osman Unsal, Adrian Cristal, Eduard Ayguade, Jesus Labarta, and Mateo Valero. Runtime-aware architectures. In *Euro-Par*, pages 16–27. 2015.

[8] Mateo Valero, Miquel Moretó, Marc Casas, Eduard Ayguadé, and Jesus Labarta. Runtime-aware architectures: A first approach. *International Journal on Supercomputing Frontiers and Innovations*, 1(1):29–44, June 2014.

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

[10] Eduardo Argollo, Ayose Falcón, Paolo Faraboschi, Matteo Monchiero, and Daniel Ortega. Cotson: infrastructure for full system simulation. *ACM SIGOPS Operating Systems Review*, 43(1):52–61, 2009.

[11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[12] John W Haskins Jr and Kevin Skadron. Memory reference reuse latency: Accelerated warmup for sampled microarchitecture simulation. In *Performance Analysis of Systems and Software, 2003 IEEE International Symposium on*, pages 195–203, 2003.

[13] Thomas M Conte, Mary Ann Hirsch, and Wen Mei W Hwu. Combining trace sampling with single pass methods for efficient cache simulation. *Computers, IEEE Transactions on*, 47(6):714–720, 1998.

[14] Thomas F Wenisch, Roland E Wunderlich, Babak Falsafi, and James C Hoe. Simulation sampling with live-points. In *Performance Analysis of Systems and Software, 2006 IEEE International Symposium on*, pages 2–12, 2006.

[15] Yue Luo, Lizy K John, and Lieven Eeckhout. Self-monitored adaptive cache warm-up for microprocessor simulation. In *Computer Architecture and High Performance Computing, 2004. SBAC-PAD 2004. 16th Symposium on*, pages 10–17, 2004.

[16] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[17] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[18] Nikola Rajovic, Paul M Carpenter, Isaac Gelado, Nikola Puzovic, Alex Ramirez, and Mateo Valero. Supercomputing with commodity cpus: Are mobile socs ready for hpc? In *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*, pages 1–12, 2013.

[19] Alejandro Rico, Alejandro Duran, Felipe Cabarcas, Yoav Etsion, Alex Ramirez, and Mateo Valero. Trace-driven simulation of multithreaded applications. In *Performance Analysis of Systems and Software, 2011 IEEE International Symposium on*, pages 87–96, 2011.

[20] Alejandro Rico, Felipe Cabarcas, Carlos Villavieja, Milan Pavlovic, Augusto Vega, Yoav Etsion, Alex Ramirez, and Mateo 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, 2012.

[21] Kiyeon Lee, S. Evans, and Sangyeun Cho. Accurately approximating superscalar processor performance from traces. In *Performance Analysis of Systems and Software, 2009 IEEE International Symposium on*, pages 238–248, 2009.

[22] Thomas Grass, Alejandro Rico, Marc Casas, Miquel Moreto, and Alex Ramirez. Evaluating execution time predictability of task-based programs on multi-core processors. In *Euro-Par 2014: Parallel Processing Workshops*, pages 218–229, 2014.

[23] Georgios Goumas, Kornilios Kourtis, Nikos Anastopoulos, Vasileios Karakasis, and Nectarios Koziris. Understanding the performance of sparse matrix-vector multiplication. In *Parallel, Distributed and Network-Based Processing, 16th Euromicro Conference on*, pages 283–292, 2008.

[24] Ehsan K Ardestani and Jose Renau. Esesc: A fast multicore simulator using time-based sampling. In *High Performance Computer Architecture, 2013 IEEE 19th International Symposium on*, pages 448–459, 2013.

[25] Trevor E Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *High Performance Computing, Networking, Storage and Analysis, 2011 International Conference for*, pages 1–12, 2011.

[26] Davy Genbrugge, Stijn Eyerman, and Lieven Eeckhout. Interval simulation: Raising the level of abstraction in architectural simulation. In *High Performance Computer Architecture, 2010 IEEE 16th International Symposium on*, pages 1–12, 2010.

[27] Thomas F Wenisch, Roland E Wunderlich, Babak Falsafi, and James C Hoe. Turbosmarts: Accurate microarchitecture simulation sampling in minutes. In *ACM SIGMETRICS Performance Evaluation Review*, volume 33, pages 408–409, 2005.

[28] Marc Casas, Harald Servat, Rosa M. Badia, and Jesús Labarta. Extracting the optimal sampling frequency of applications using spectral analysis. *Concurrency and Computation: Practice and Experience*, 24(3):237–259, 2012.

[29] Juan Gonzalez, Judit Gimenez, Marc Casas, Miquel Moreto, Alex Ramirez, Jesus Labarta, and Mateo Valero. Simulating whole supercomputer applications. *IEEE Micro*, (3):32–45, 2011.

[30] Lieven Eeckhout, Yue Luo, Koen De Bosschere, and Lizy K John. Blrl: Accurate and efficient warmup for sampled processor simulation. *The Computer Journal*, 48(4):451–459, 2005.

[31] Stephen L. Olivier, Bronis R. De Supinski, Martin Schulz, and Jan F. Prins. Characterizing and mitigating work time inflation in task parallel programs. In *High Performance Computing, Networking, Storage and Analysis, 2012 International Conference for*, pages 1–12, 2012.