RPPM: Rapid Performance Prediction of Multithreaded Workloads on Multicore Processors

Sander De Pestel[†*] Sam Van den Steen[†*] Shoaib Akram[‡] Lieven Eeckhout[‡]

[†]Intel, Belgium [‡]Ghent University, Belgium

***Abstract*—Analytical performance modeling is a useful complement to detailed cycle-level simulation to quickly explore the design space in an early design stage. Mechanistic analytical modeling is particularly interesting as it provides deep insight and does not require expensive offline profiling as empirical modeling. Previous work in mechanistic analytical modeling, unfortunately, is limited to single-threaded applications running on single-core processors.**

This work proposes RPPM, a mechanistic analytical performance model for multi-threaded applications on multicore hardware. RPPM collects microarchitecture-independent characteristics of a multi-threaded workload to predict performance on a previously unseen multicore architecture. The profile needs to be collected only once to predict a range of processor architectures. We evaluate RPPM's accuracy against simulation and report a performance prediction error of 11.2% on average (23% max). We demonstrate RPPM's usefulness for conducting design space exploration experiments as well as for analyzing parallel application performance.

I. Introduction

Simulation is the predominant methodology for computer architects to evaluate new processor architectures. Unfortunately, simulation is extremely time-consuming and tedious, especially when simulating multi-threaded workloads on multicore hardware. Analytical performance modeling is an attractive complement to detailed cycle-level simulation to quickly explore large design spaces at early stages of the design process. Several research groups have proposed analytical performance models for superscalar processors. These techniques can be broadly classified into three main categories: (1) empirical models, (2) mechanistic models and (3) hybrid models. Empirical models use training data obtained through simulation to create black-box models using machine learning and regression techniques, see for example [18], [23], [24], [25], [26], [30]. Mechanistic models are white-box models that capture the first-order mechanics of a processor, see for example [15], [37]. Hybrid models cover the middle ground through parameter fitting of a parameterized semi-mechanistic model, see for example [8], [16]. Empirical models are typically very accurate but do not provide insight and require extensive offline training. Mechanistic models are challenging to develop but once developed, they provide deep insight and do not require further offline training. This paper seeks to advance the state of the art in mechanistic modeling.

Prior work in mechanistic modeling is limited to single-threaded processors. Interval modeling, developed over a series of research papers by Michaud et al. [28], then Karkhanis and Smith [22] and finally Eyerman et al. [15], models a superscalar processor performance by building up a CPI stack of components that represent useful computation versus lost cycles due to miss events. To collect the number of miss events (cache misses, branch misprediction rates, etc.), interval modeling relies on offline functional cache and branch predictor simulation. More recently, Van den Steen et al. [37] improved upon this prior work by collecting only microarchitecture-independent characteristics of an application. The key advantage of doing so is that it allows for profiling the workload only once after which performance can be predicted for a range of previously unseen architectures. This prior work unfortunately is limited to single-core processors.

This paper extends this prior work for predicting multi-threaded application performance on multicore hardware. Mechanistic modeling of multi-threaded application performance is fundamentally more difficult than predicting single-thread performance. Not only do we need to accurately model per-thread performance, we also need to accurately model synchronization, resource interference and cache coherence effects. Moreover, as demonstrated in this paper, multi-threaded application performance prediction is further complicated by the fact that small prediction inaccuracies of the execution time in-between synchronization events, lead to an accumulation of errors across the entire program execution because application progress is determined by the slowest (most critical) thread.

Straightforward extensions of prior work towards multi-threaded workloads further motivates this work. Predicting multi-threaded application performance based on only the main thread or only the critical thread leads to an average performance prediction error compared to detailed simulation of 45% and 28%, respectively, and a maximum error above 110%. There are three reasons for the poor accuracy: (1) it does not model contention in shared resources, (2) it does not model cache coherence effects, and (3) it does not model synchronization overhead.

Some prior work focused on multicore performance prediction. Jongerius et al. [21] propose a multicore performance model for multiprogram workloads. Hence they only focus on resource contention (i.e., negative interference) and do not model positive interference, cache coherence nor synchronization. Popov et al. [32] predict multi-threaded application performance using Amdahl's Law supplemented with the simulation of snippets of representative parallel code regions.

[*]This work was done while at Ghent University.

We propose RPPM, a mechanistic analytical model for predicting multi-threaded application performance on multi-core hardware [11]. A profiler collects a set of characteristics that captures a workload's behavior in a microarchitecture-independent way. The profile contains per-thread characteristics, as for the single-threaded model, as well as characteristics that affect inter-thread interactions, including shared memory access behavior and synchronization. The profile is then used to predict performance on a previously unseen multicore architecture. We conjecture that the number of threads considered during profiling equals the number of cores of the target architecture. A key feature of RPPM is that the profile needs to be collected only once, using which the performance for a range of multicore architectures can be predicted. Although the profile is measured during a particular multi-threaded execution, and therefore it may be subject to a particular inter-thread interleaving, we find it to enable accurate performance prediction across architectures.

We evaluate the accuracy of RPPM against detailed cycle-level simulation for all the OpenMP multi-threaded Rodinia benchmarks and a subset of the pthread-based Parsec benchmarks. RPPM predicts performance within 11.2% on average (23% max error) for a quad-core processor. We demonstrate the usefulness and applicability of RPPM for design space exploration studies and application performance analysis. In particular, we use RPPM to quickly identify the optimum among five design points with varying characteristics in terms of pipeline width and clock frequency while delivering the same peak performance (in operations per second). We use RPPM to construct bottlegraphs to analyze an application's parallel execution (im)balance.

II. Motivation and Background

Modeling multi-threaded workload performance is challenging for at least three reasons. (1) One needs to accurately model per-thread performance. (2) One needs to accurately model inter-thread synchronization and interaction, including resource interference and cache coherence effects. (3) Because threads synchronize in a multi-threaded workload, there is an effect of accumulating errors. The latter is probably less well-known, hence we describe it next.

A. Accumulating Errors

In contrast to single-threaded performance modeling where performance prediction errors over relatively small execution regions are averaged out across the entire program execution, this is not the case for multi-threaded applications. Modeling multi-threaded performance is complicated by the fact that accurate predictions are needed in-between synchronization events, i.e., inaccurately predicting performance for the critical thread between synchronization events leads to an accumulation of error when predicting overall application performance.

We substantiate this claim through the following discussion. Without loss of generality, consider a barrier-synchronized multi-threaded application. (Other synchronization mechanisms such as critical sections and producer-consumer interactions face similar issues.) Predicting the execution time for each thread in an inter-barrier region may lead to over- and under-estimations for different threads. On average, we assume (expect) the per-thread execution time to be predicted accurately for each inter-barrier region, i.e., the execution time may be over-estimated for some threads and under-estimated for others. However, even though the execution time predictions are accurate on average across all threads, this is not enough for multi-threaded workloads because the execution time of the inter-barrier region is determined by the slowest thread. Over-estimations of the execution time of inter-barrier regions lead to an accumulation of errors.

TABLE I: Accumulating prediction errors in barrier-synchronized applications: overall prediction error as a function of thread count and inter-barrier prediction error. *Prediction errors accumulate because of synchronization.*

#Threads	Inter-barrier error		
	1%	5%	10%
1	0.00%	0.00%	0.00%
2	0.33%	1.67%	3.34%
4	0.60%	3.00%	6.01%
8	0.78%	3.89%	7.79%
16	0.88%	4.41%	8.83%

We illustrate this further using a micro-benchmark consisting of a loop of one million iterations for which each iteration takes the same amount of time. Assume now that the analytical model is 100% accurate on average but introduces some (random) over- or under-estimations within a given bound. The loop is parallelized such that n iterations run in parallel, with n the number of threads. All threads synchronize at a barrier after they have each executed one iteration. We run this micro-benchmark with different number of threads and different assumed inter-barrier prediction errors. The results are shown in Table I. When only a single thread is running, the over- and underestimations balance each other out and the resulting prediction error equals zero, i.e., we perfectly predict the average inter-barrier execution time, as expected. However, when running multiple threads, the execution time of an inter-barrier region is determined by the slowest thread reaching the barrier. As a result, the prediction error accumulates across barriers, leading to significant inaccuracies for predicting overall application execution time. We note that the error increases with increasing thread count.

B. Single-Threaded Performance Model

With this mind, we now provide a brief background on microarchitecture-independent analytical performance modeling for single-threaded applications, which we build upon to model per-thread performance in RPPM; we refer the reader to [37] for a more elaborate exposition of the single-threaded performance model. We next describe naive extensions to this prior work to predict multi-threaded application performance, which, as we will show in the evaluation, are inaccurate.

$$C = \overbrace{\frac{N}{D_{eff}}}^{\textbf{Base}} + \underbrace{m_{bpred} \times (c_{res} + c_{fr})}_{\textbf{Branch}} + \overbrace{\sum_{level=i} m_{ILi} \times c_{Li+1}}^{\textbf{I-cache}} + \underbrace{\frac{m_{LLC} \times c_{mem}}{MLP}}_{\textbf{D-cache}} \tag{1}$$

The single-threaded performance model consists of two steps. In the profiling step, we use a Pin tool [27] to collect an application profile containing only microarchitecture-independent statistics, i.e., these statistics are inherent to the workload and are independent of the underlying microarchitecture. In the prediction step, these statistics are used as input to the analytical model to predict the execution time on a particular processor configuration. Execution time for a single thread running on an out-of-order processor is predicted using Equation 1.
We distinguish four components in the model:

Instruction-level parallelism: The base component is obtained by dividing the number of micro-ops (N) by the effective dispatch rate (D_{eff}). The effective dispatch rate is a function of the width of the front-end pipeline, the available ILP in the application, and the amount of contention in the functional units. To accurately model ILP, we need fine-grained profile information, i.e., we collect statistics regarding instruction mix and inter-instruction dependences at the granularity of a thousand instructions, which we call a micro-trace. In order not to slow down profiling too much, we profile a micro-trace of a thousand instructions once every one million instructions. This allows us to characterize time-varying behavior in ILP at a moderate profiling cost.

Branch misprediction: The branch component quantifies the lost cycles due to branch mispredictions and is computed as the number of mispredictions (m_{bpred}) times the branch resolution time (c_{res}) (this is the time between the branch being dispatched into the issue queue and reorder buffer and its execution) plus the front-end pipeline refill time (c_{fr}). Prior work profiles branch behavior in a microarchitecture-independent way using the information theoretic notion of entropy [10], and uses this entropy profile to predict the branch misprediction rate for a particular branch predictor.

Instruction cache: The I-cache component quantifies the impact of instruction cache misses and is computed as the product of the cache miss rate at each level (m_{ILi}) and the respective miss latency (c_{Li+1}). The cache miss rates are predicted using micro-architecture-independent reuse distance distributions using StatStack [14].

Long-latency loads: The D-cache component quantifies the time the core stalls waiting for main memory requests to resolve as a result of long-latency load misses. This component is computed as the number of last-level cache misses due to load instructions (m_{LLC}) times the average memory access latency (c_{mem}), divided by the amount of memory-level parallelism (MLP) or the average number of outstanding long-latency load misses if at least one is outstanding. MLP is computed using a microarchitecture-independent model as described in [36].

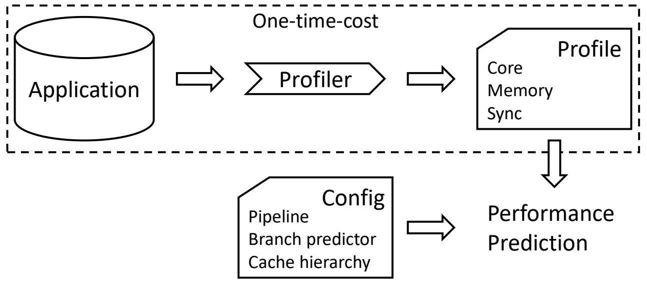

Fig. 1: Schematic overview of the RPPM tool. *A multi-threaded application is profiled once to capture its microarchitecture-independent characteristics; the profile is then used as input to the RPPM model to predict performance for a specific multicore processor.*

C. Naive Multi-Threaded Extensions

We now consider two naive extensions of this prior work to predict the execution time of multi-threaded applications running on a multi-core processor. In the evaluation, we will compare RPPM's accuracy against these approaches.

MAIN: In the first approach, we only profile the main thread. We define the main thread as the thread that gets initiated upon program execution; this thread completes the initialization phase before creating the other worker threads, and finalizes the execution once the worker threads have finished their execution. We apply the single-threaded model as described above to predict the execution time of the main thread. The predicted execution time for the main thread is then used as a prediction for the overall execution time of the multi-threaded application.

CRIT: The second approach profiles all application threads separately instead of only the main thread. After using the model to predict the execution time for every thread, the thread with the longest execution time will be marked as the critical thread. We then use the predicted execution time of the critical thread as a prediction for the overall execution time of the multi-threaded application.

Both these naive extensions do not properly take synchronization into account. Nor do they account for interference in shared resources and cache coherence effects.

III. RPPM

RPPM predicts multithreaded application performance using two key components, see also Figure 1:

1) A profiler that collects microarchitecture-independent statistics including per-thread characteristics, shared memory access behavior and synchronization events.
2) A prediction tool that takes these statistics as input and predicts multi-threaded execution time on a particular multi-core processor architecture.

A key property of RPPM is that the profile contains a collection of characteristics that are independent of the underlying microarchitecture. Hence, we need to collect it only once and we can then predict performance for a range of multicore architectures. We note though that RPPM assumes the same number of threads during profiling as there are cores in the processor architecture for which we make the prediction. However, a single profile can be used to predict performance for a wide range of multicore architectures while varying clock frequency, pipeline width and depth, window and buffer sizes, cache sizes, cache hierarchies, branch predictor, etc. The fact that the profile is independent of the underlying microarchitecture speeds up design space exploration studies substantially as the profiling cost is amortized across many predictions. Future work will explore extending RPPM towards systems with more threads than cores — this will require modeling various multi-tasking effects including thread-to-core mapping, caching, scheduling quanta, etc. [33].

A. *Profiling*

Profiling is done using a Pin tool [27], a dynamic binary instrumentation tool. Some of the characteristics that we collect are the same as for the single-threaded model by Van den Steen et al. [37], i.e., statistics that relate to an individual thread's execution such as instruction mix, inter-instruction dependences and branch behavior. To model multi-threaded execution behavior, we in addition need to profile synchronization as well as memory system behavior.

Synchronization: We track all synchronization events (barriers, critical sections, etc.) by tracking specific library function calls. RPPM provides support for both the pthread and OpenMP parallel execution models.

When using the pthread library, the programmer typically uses the available function calls to mark synchronization events. Defining a critical section for example is done by calling `pthread_mutex_lock` and `pthread_mutex_unlock` at the start and end of a critical section, respectively. For OpenMP, the programmer marks a `for` loop with a `#pragma` telling the compiler to insert the necessary function calls for the runtime environment to execute the loop using parallel threads. For example, the function call to `gomp_team_barrier_wait` marks a barrier. We capture these function calls in the profiler and log the location of the calls in the application's synchronization profile.

Complex parallel applications use multiple barriers and/or multiple critical sections. To be able to distinguish different synchronization events, we also track function arguments. For example, the function `gomp_team_barrier_wait` passes the barrier (`gomp_barrier_t`) as a pointer, and by tracking these function arguments we keep track of which specific barrier a thread is waiting for.

If a programmer were to implement user-level synchronization, one cannot rely on tracking specific function calls. Instead, one would need to manually annotate the source code to mark the various synchronization events.

Condition Variables: Condition variables are a widely used synchronization primitive but require special support beyond what we described in the previous section. Condition variables are frequently used to implement barriers, for which the variable is used to count the number of threads that have reached the barrier, see also Algorithm 1. A thread increments the condition variable as it grabs the lock and finishes its work, and checks whether the variable equals the number of worker threads. If not, the thread calls the `pthread_cond_wait` function, pauses its execution and releases the lock. If the condition is met, the thread calls the `pthread_cond_broadcast` function to tell all waiting threads that the condition is satisfied and all threads can continue their execution.

Algorithm 1 Barrier using condition variables.

```
1: pthread_mutex_lock(&mutex);
2: n++;
3: if n < threads then
4:     pthread_cond_wait(&cond, &mutex);
5: n = 0;
6: pthread_cond_broadcast(&cond);
7: pthread_mutex_unlock(&mutex);
```

As mentioned in the previous section, we profile all pthread library calls to characterize an application's synchronization behavior. Unfortunately, this is not sufficient for condition variables because the `pthread_cond_wait` function is not always called. In particular, the last thread arriving at the barrier does not actually call this function, see Algorithm 1. The problem here is that which thread arrives at the barrier the latest, or in other words, which thread does not call the `pthread_cond_wait` function, depends on the micro-architecture on which the application is executed, and may be different between the profiling run and the run for which the model predicts performance. To be able to adequately model condition variables, we need to know when there is a 'possibility' for a thread to wait — not only when the thread effectively waits during the profiling run. We therefore introduce a marker between lines 2 and 3 in Algorithm 1 to notify the profiler that all threads can potentially call the `pthread_cond_wait` function. This allows RPPM to capture the condition variable for all threads.

Condition variables are used not only for barrier synchronization, but also for other synchronization constructs including producer-consumer synchronization, semaphores, and other variations. All of these will only under certain conditions call the wait, broadcast and signal functions. We solve this problem by adding markers in the source code and by catching them during profiling. While this involves manual changes to the source code, it is not that cumbersome in practice: searching the respective condition variable function calls and adding markers is fairly straightforward. For our set of benchmarks from Rodinia and Parsec, we have five benchmarks that use condition variables. For four benchmarks (bodytrack, raytrace, streamcluster and vips), we had to add one marker for the `pthread_cond_wait` function. For facesim, we added a marker for the `pthread_cond_wait`

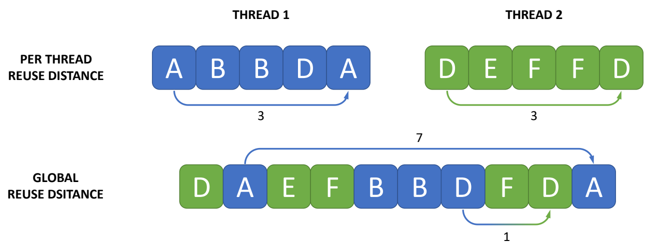

Fig. 2: StatStack characterizes cache sharing and coherence in a microarchitecture-independent way using per-thread and global reuse distance distributions.

function and a marker for the `pthread_cond_broadcast` function. In total, we thus had to add six markers.

Memory Behavior: In this work, we make a distinction between cold misses versus capacity and conflict misses. We model cold misses by simply counting the number of unique memory locations accessed by a thread and across all threads. We model capacity and conflict misses using a microarchitecture-independent reuse distance profile. StatStack [14] collects a per-thread distribution of the reuse distance between two references (by any thread) to the same memory location. The StatStack extension for multi-threaded applications by Ahlman [1] enables predicting both positive and negative interference in shared caches as well as cache coherence effects.

We first briefly describe how StatStack operates for single-threaded applications. StatStack records the reuse distance between memory accesses during the profiling phase. Reuse distance is defined as the number of accesses between two accesses to the same memory location (cache line). Collecting reuse distances is relatively cheap, i.e., it suffices to just increment a counter per memory location. Collecting stack distances which quantifies the number of *unique* memory accesses between two accesses to the same memory location incurs much higher overhead because one needs to maintain and search a stack of unique references. However, it is straightforward to estimate the miss rate for a fully associative LRU cache using the stack distance distribution [17]. StatStack therefore converts the reuse distance distribution, which is easy to collect, into a stack distance distribution to predict cache miss rate.

Extending to multi-threaded applications requires that we somehow interleave the memory access streams of the different threads to characterize the impact of cache sharing and coherence. To this end, we use a global memory access counter per memory location during profiling to calculate the reuse distance across all threads. This is similar to the single-threaded case except that we now consider a counter that counts the number of memory accesses between two accesses to the same memory location across *all* threads, rather than per thread. This captures the impact different threads have upon each other as depicted in Figure 2. In the per-thread access stream for thread 1, the second access to address A has a reuse distance of 3 and a stack distance of 2; however, because of interleaving of the different threads, the reuse distance increases to 7 with a stack distance of 4. An increase in stack distance typically implies negative interference, i.e., a cache line brought in by a thread may be evicted by another thread. Note that interference does not need to be negative. In fact, in case of data sharing, a thread may bring in data that another thread may also need. For example, the reuse distance of the second memory access to address D equals 3 for the second thread when considered in isolation, but it decreases to 1 when interleaved with the first thread.

We characterize memory hierarchy behavior by collecting two reuse distance distributions per thread. The first one uses per-thread counters per memory location to capture a thread's local reuse distance distribution, and is used to predict the cache miss rates of the private L1 and L2 caches. The second uses a global (i.e., shared among all threads) counter per memory location to capture a thread's reuse distance distribution when interleaved with the other concurrently executing threads. This global reuse distance distribution is used to predict the cache miss rate in the shared L3 cache. Write invalidation due to cache coherence is captured by verifying whether a memory location accessed twice by a given thread was written by any other thread in-between the two accesses. If so, write invalidation is detected and an infinite reuse distance is recorded in the per-thread reuse distance distribution, indicating a cache miss for the second access.

It is worth noting that although the reuse distance distributions used by StatStack are measured during a particular profiling run on a particular machine — the distributions may therefore be subject to a particular inter-thread interleaving — StatStack models these inter-thread interactions in a statistical way making the specific ordering during profiling less critical. Moreover, we find that different distributions collected during different profiling runs lead to very similar performance predictions.

Putting It Together: Figure 3(a) illustrates how profiling is done while taking into account synchronization. The grey parts denote sequential execution by the main thread; colored parts denote parallel execution by the main thread and the worker threads. Synchronization events (barriers in this example shown as black horizontal lines) delineate different epochs. We collect a separate profile per inter-synchronization epoch for each thread. This profile contains the per-thread characteristics. This includes the instruction mix, instruction dependence distribution, branch behavior, reuse distance distributions, etc. The inter-synchronization epoch profiles then serve as input to the performance prediction model, which we describe next.

B. Prediction

The multi-threaded performance model itself operates in two phases. The first phase (Figure 3(b)) predicts the active execution time for each thread in-between synchronization events. The second phase (Figure 3(c)) accounts for synchronization events and introduces predicted synchronization

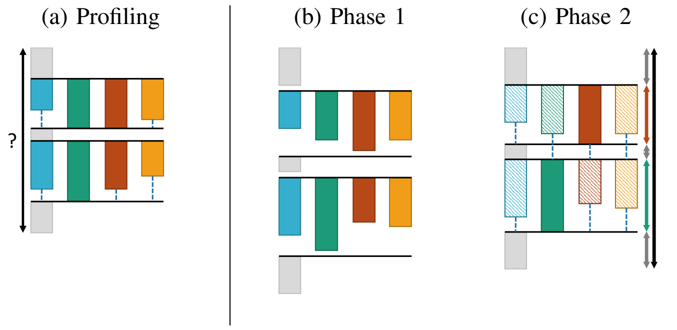

Fig. 3: RPPM predicts multithreaded execution time in three steps: (a) We profile an application's synchronization behavior and per-epoch statistics for each thread. We then predict an application's execution time (b) by predicting per-epoch active execution times for each active thread, and (c) by estimating the impact of synchronization on overall application performance.

Algorithm 2 Estimating synchronization overhead

```
1: while not finished do
2:     for Thread T in sorted(Threads, shortestTimeFirst()) do
3:         if not isBlocked(T) then
4:             Proceed T to next synchronization event
```

overhead (idle time because of synchronization) to predict overall execution time. We now describe these two phases.

Per-epoch active execution time: We use the micro-architecture-independent profile to predict per-epoch active execution times for each thread. To do so, we use Equation 1 from the single-threaded model. Although we use the same equation, some of the numbers that serve as input to the model need to be computed differently. In particular, we need to account for the impact shared resources and cache coherence may have on per-thread performance, for which we leverage the multi-threaded extension of StatStack as mentioned before. In particular, we use the per-thread reuse distance distributions to estimate the miss rates in the private L1 and L2 caches, and we use the global reuse distance distributions to estimate the per-thread miss rates in the shared LLC.

Synchronization overhead: The overall execution time of a multithreaded application is predicted by combining the predicted per-epoch active execution times for each of the threads with the predicted synchronization overhead.

Estimating synchronization overhead, or estimating per-thread idle time because of synchronization, is done using Algorithm 2. We identify the thread with the shortest total execution time (active and idle time) thus far that is not blocked by the next synchronization event and symbolically proceed it to this next event. This process iteratively progresses the threads from the fastest to the slowest running thread to the next synchronization event. The slowest thread eventually determines the timing of the synchronization event. The faster running threads experience idle time while the slower threads have still not reached the synchronization event. By doing so, we emulate the execution behavior at each synchronization event and we repeat this process until all threads reach the end of execution and the application finishes. At the end of the symbolic execution, the critical path through the execution determines the application's execution time.

During the symbolic execution while emulating a synchronization event, we calculate the number of cycles a thread spends waiting for other threads, not making forward progress. This is illustrated in Figure 3(c) for barrier synchronization. Active execution time is depicted by a box; waiting time is depicted by a dashed line; overall execution time is determined by the slowest thread in-between synchronization events. In particular, the execution time of the first inter-barrier epoch is determined by the third thread; the execution time of the second inter-barrier epoch is determined by the second thread; overall execution time is predicted by summing up the predicted inter-barrier execution times and the main thread's execution times when it is running alone.

We model the synchronization events as follows:

- Thread creation: The main thread is created at application start-up; all other threads are therefore initially marked as 'blocked'. When the main thread creates a new thread, the thread is 'unblocked' and its start time is set accordingly.
- Critical sections: A critical section is a code segment that has to be executed atomically, by a single thread at a time. We mark accessing and leaving a critical section as a synchronization event. Before a thread is allowed to enter a critical section, the symbolic execution verifies that no other thread is currently executing that same critical section. If so, the thread blocks waiting for the critical section to be released. Once released, the thread is allowed to proceed and enter the critical section. The waiting time and the actual execution time of the critical section determines overall execution time.
- Barriers: A barrier is a location in the code where all threads need to wait for each other to finish the execution of their respective code segment. A thread can only continue when all threads have reached the barrier. When a thread arrives at a barrier it checks whether all other threads already reached the barrier. If not, the thread blocks itself and waits. The last thread arriving at the barrier releases the barrier and determines the execution time of the inter-barrier epoch.
- Condition variables: As mentioned in Section III-A, we add markers to catch condition variables during profiling. We use these markers to verify the intended behavior of the condition variable. If the condition variable is used to implement barrier synchronization — easily recognized if all but one of the threads need to wait at the condition variable and any of the threads can release the barrier — we model the condition variable as a barrier, as just described. A producer-consumer relationship is recognized if a thread or set of threads waits at the synchronization event — these are the consumer threads — and another thread or set of threads calls the broadcast function to release the waiting

TABLE II: Rodinia benchmarks and their inputs.

Benchmark	Input	Benchmark	Input
Backprop	4,194,304	LUD	2048.dat
BFS	graph8M	NN	4096k
CFD	fvcorr.domn.010K	NW	16k x 16k
Heartwall	test.avi 10	Particlefilter	128 x 128 x 10
Hotspot	16384 5	Pathfinder	1M x 1k
Kmeans	kdd_cup	SRAD	2048
LavaMD	10	Streamcluster	256k
Leukocyte	testfile.avi 5		

thread(s) — these are the producer threads. Waiting at the synchronization event may be conditional, i.e., threads only have to wait if there are no items to process. The broadcast operation may be conditional as well, i.e., the producer may only broadcast new items if at least one consumer is waiting for a new item. The producer-consumer relationship is modeled by keeping track of the number of broadcast markers, i.e., the number of created items. If the number of created items equals zero at the time a consumer reaches the synchronization event, the consumer threads is stalled. As soon as the number of items is larger than one, the consumer thread resumes its execution.

- Thread joining: A join occurs when waiting for a thread to terminate. Its behavior is similar to a barrier, i.e., the execution time of the longest running thread determines when the join happens. The difference in execution time is added as idle time to the shortest running thread.

We note that this is not a complete list of all possible synchronization events, but a list of all events encountered in our benchmark suite. Nevertheless, we are convinced that unlisted events like semaphores or even indirect synchronization can be modeled in a similar way.

IV. Experimental Methodology

Benchmarks: We consider all the benchmarks from the Rodinia benchmark suite v3.1 [7] as well as a subset of benchmarks from Parsec v3.0 [3]. We use the OpenMP implementations of the Rodinia benchmarks and the pthread versions of the Parsec suite to evaluate RPPM for different parallel execution models. We set the number of cores to 4 in our experiments[1]. All of the Rodinia benchmarks create a pool of 3 worker threads, which along with the main thread, leads to 4 parallel threads. The Parsec benchmarks spawn more threads, but we limit the amount of parallelism to 4, meaning that only 4 threads will execute simultaneously. The reason for doing so is because RPPM does not model simultaneous multi-threading (SMT), i.e., it assumes one thread per core. This is also the reason why we exclude dedup, ferret and x264 as they create as many as 16 worker threads. For the other benchmarks, we set the number of threads such that at most 4 threads have an impact on execution time that is larger that 1% and we have no more than 8 threads (twice the number of cores).

[1]Simulation times prevented us from experimenting with larger core counts.

TABLE III: Synchronization events in the Parsec benchmarks.

Benchmark	Critical Sections	Barriers	Cond. var.
Blackscholes	–	–	–
Bodytrack	6,700	98	25
Canneal	4	64	–
Facesim	10,472	–	1,232
Fluidanimate	2,140,206	50	–
Freqmine	–	–	–
Raytrace	47	–	15
Streamcluster	68	13,003	34
Swaptions	–	–	–
Vips	8,973	–	1,433

We predict the execution time of the parallel region of interest (ROI), which starts after initialization and ends before finalization by the main thread; multiple parallel threads co-execute in the ROI.

Data inputs: We use the medium inputs for the Parsec benchmarks, but since there are no predefined inputs for the Rodina suite, we select input data sets that lead to reasonable simulation times while executing a sufficient number of instructions in the ROI, see Table II. Our benchmarks execute between 50 million to 50 billion instructions in the ROI, with LLC MPKI values ranging up to 40, and MLP ranging up to 5.3 for backprop.

Synchronization: Table III reports the number of dynamic synchronization events for the Parsec benchmarks. We make a distinction between critical sections, barriers and condition variables. Some benchmarks are dominated by critical sections (e.g., fluidanimate), others by barriers (e.g., streamcluster), others by condition variables (e.g., facesim and vips), or combination thereof. Some benchmarks, e.g., blackscholes, freqmine and swaptions, do not use any of these synchronization types; this is because these benchmarks are synchronized at the end of the execution through a join operation, not reported in Table III. For completion, we note that the Rodinia benchmarks only feature barrier synchronization.

Simulator: We evaluate RPPM's accuracy by comparing its performance predictions against simulation. We simulate the benchmarks using Sniper, a state-of-the-art, parallel multicore simulator, while considering its most accurate cycle-level hardware-validated core model [5][2]. We simulate the base multicore configuration as specified in Table IV, unless mentioned otherwise. These simulated execution times serve as our golden reference.

Profiling: We profile the benchmarks on an Intel Xeon Gold 6140 (Skylake). We assume the same number of threads during profiling as for prediction.

V. Evaluation

We evaluate RPPM's accuracy against cycle-level simulation and compare against two naive extensions of the previously proposed single-threaded performance model, MAIN and CRIT, as previously described in Section II-C. The results

[2]Hardware validation reports an 11% average prediction error for the most detailed core model, which we use in this work.

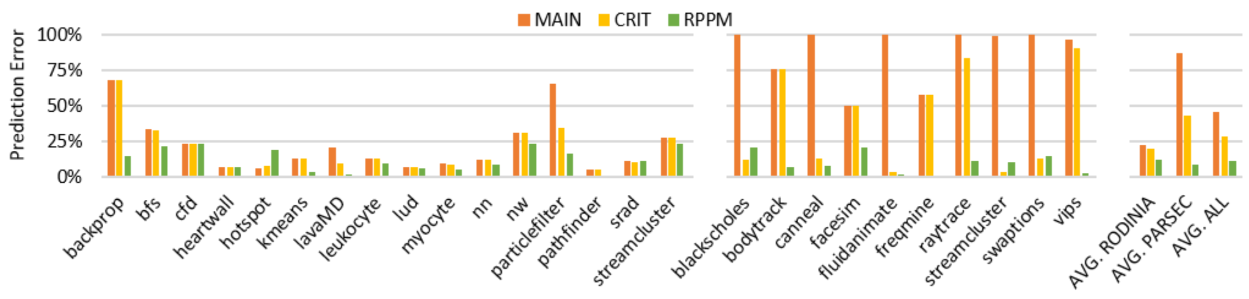

Fig. 4: Prediction error for MAIN, CRIT and RPPM compared to cycle-level simulation for the Rodinia and Parsec benchmarks. *RPPM achieves an average prediction error of 11.2% and significantly outperforms the MAIN and CRIT approaches.*

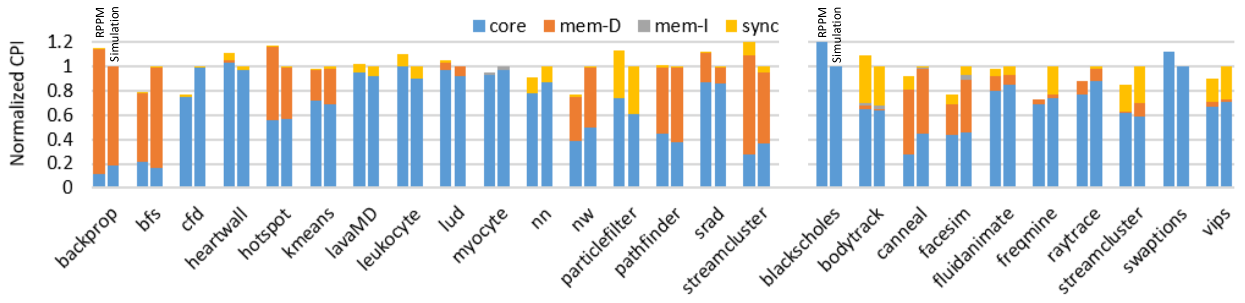

Fig. 5: CPI stacks by RPPM (left bar) and through simulation (right right), normalized to simulation. *The prediction error is primarily attributed to inaccurate predictions of the base and data memory CPI components.*

TABLE IV: Simulated architecture configurations.

	Smallest	Small	**Base**	Big	Biggest
frequency [GHz]	5.00	3.33	2.50	2.00	1.66
dispatch width	2	3	4	5	6
ROB size	32	72	128	200	288
issue queue size	16	36	64	100	144
branch predictor	4 KB, tournament				
L1-I cache	32 KB, 4-way, private				
L1-D cache	32 KB, 4-way, private				
L2 cache	256 KB, 8-way, private				
LLC	8 MB, 16-way, shared				

are shown in Figure 4, with the Rodinia benchmarks on the left-hand side and the Parsec benchmarks on the right-hand side; averages are reported on the far right.

MAIN predicts the execution time of the main thread to predict overall application performance. This leads to an average absolute prediction error of 45% with several outliers up to 100%. The outliers are more common for the Parsec benchmarks because the main thread is just doing some bookkeeping and not performing any actual work. This leads to a significant underestimation of the application's overall execution time.

CRIT predicts the execution time for all threads and then takes the execution time of the slowest thread (critical thread) as a prediction for the application's execution time. CRIT reduces the prediction error to 28% on average. CRIT is more accurate than MAIN, particularly for the Parsec benchmarks, because CRIT models the worker threads as opposed to just modeling the main thread as done by MAIN.

RPPM clearly outperforms MAIN and CRIT with an average absolute error of 11.2% and a maximum error of 23%. RPPM accurately predicts which thread is the most critical thread in-between synchronization events which leads to an overall more accurate prediction than MAIN and CRIT.

To help understand where the remaining error is coming from, Figure 5 illustrates the average per-thread normalized CPI stacks. We measure average per-thread CPI by computing the respective CPI stacks for each thread separately and then compute the average. RPPM's modeling error is due to inaccurate predictions for the base component (e.g., cfd), the mem-D component (e.g., backprop) or both (e.g., nw). These inaccuracies originate from the single-threaded prediction model and/or the extended memory hierarchy model, which indirectly leads to incorrect predictions for the synchronization component.

VI. Case Studies

Having evaluated the accuracy of RPPM, we now consider two case studies to illustrate the usefulness and applicability of RPPM.

A. Design Space Exploration

Our first case study considers a design space exploration problem in which we profile the Rodinia benchmarks and predict their performance for the five configurations listed in Table IV. We vary processor width from 2 to 6 (and scale ROB and issue queue resources accordingly) and vary clock

TABLE V: Case study: Predicting the optimum design point in a design space. *RPPM accurately predicts the (near) optimum design points in a complex design space.*

Bound	**0%**		**< 1%**		**< 3%**		**< 5%**	
backprop	0.00%	1	0.00%	1	0.00%	1	0.00%	1
bfs	0.00%	1	0.00%	1	0.00%	1	0.00%	2
cfd	0.00%	1	0.00%	1	0.00%	1	0.00%	1
heartwall	0.00%	1	0.00%	1	0.00%	1	0.00%	1
hotspot	0.00%	1	0.00%	1	0.00%	1	0.00%	1
kmeans	0.00%	1	0.00%	1	0.00%	1	0.00%	2
lavaMD	0.00%	1	0.00%	1	0.00%	1	0.00%	1
leukocyte	0.00%	1	0.00%	1	0.00%	1	0.00%	1
lud	0.00%	1	0.00%	1	0.00%	1	0.00%	1
myocyte	0.00%	1	0.00%	1	0.00%	1	0.00%	1
nn	0.00%	1	0.00%	1	0.00%	1	0.00%	1
nw	10.15%	1	10.15%	1	10.15%	1	0.00%	2
particlefilter	0.00%	1	0.00%	1	0.00%	1	0.00%	1
pathfinder	1.97%	1	1.97%	1	1.97%	1	1.97%	2
srad	0.00%	1	0.00%	1	0.00%	1	0.00%	1
streamcluster	19.11%	1	0.00%	2	0.00%	2	0.00%	2
average	1.95%		0.76%		0.76%		0.12%	

frequency from 5 to 1.66 GHz across these design points. Note that the maximum number of operations that can be executed per second is constant across the five design points, i.e., all five configurations can execute at most 10 billion instructions per second.

We use RPPM to identify the design points that are within a bound of $x\%$ of the predicted optimum, see Table V which compares simulated performance of the predicted optimum versus the true optimum (obtained through exhaustive simulation of all design points). If the bound is set to 0%, RPPM identifies only a single design point which it predicts to be optimal. The predicted optimum is the effective optimum for all but three of the benchmarks, namely nw, pathfinder and streamcluster. The average deficiency (performance difference) compared to the real optimum is 1.95% (see bottom row) and up to 19.1% for streamcluster. Relaxing the bound allows for identifying (potentially) several near-optimum design points, out of which simulation can then be used to evaluate these designs and identify the optimum. For a 1% bound, we find that we identify two near-optimum design points for streamcluster out of which simulation can determine the true optimum. This eliminates the deficiency for this benchmark and brings down the average deficiency to 0.76%. Setting a higher bound of 5% increases the number of predicted optimum design points (up to 2 for some benchmarks, see rightmost column) but brings down the deficiency of the identified design points to the true optimum to at most 1.97% for pathfinder.

B. Bottlegraphs

Our second case study considers bottlegraphs [13] to visualize multi-threaded program performance. Each thread is represented as a box, with its height equal to the thread's share in the total program execution time. The box' width is equal to the thread's parallelism (number of threads running in parallel if that thread is active). The boxes of all threads are stacked upon each other with the widest box appearing at the bottom, leading to a stack with a height equal to execution time. The tallest box appears at the top of the stacked representation, illustrating the bottleneck of the multi-threaded application. Ideally, assuming a four-threaded application running on a quad-core processor, all threads have a box with a height that is one fourth of the total execution time, and the width equals four; this suggests that all four threads run concurrently in a balanced way. In cases of imbalanced execution, e.g., one thread incurs sequential work, some threads will have a taller and narrower box compared to the other threads.

Figure 6 reports the bottlegraphs for all the Parsec benchmarks. (We do not report bottlegraphs for Rodinia because these benchmarks are very well balanced, yielding almost perfect bottlegraphs.) We report two bottlegraphs per benchmark, on either side of the vertical axis. The right-hand side reports the bottlegraph obtained through simulation; the left-hand side reports the bottlegraph through RPPM.

The key take-away is that RPPM accurately predicts an application's bottlegraph obtained through simulation. We categorize the Parsec benchmarks in three groups based on their bottlegraphs. (1) The top five benchmarks are well balanced, i.e., the main thread divides the work among four well-balanced worker threads — the main thread is not doing any actual work. (2) The bottom left benchmarks are less well-balanced: the main thread spawns three worker threads and both the main thread and the worker threads perform actual work. For facesim, we observe that the execution is fairly well balanced although the main thread needs to do slightly more work than the worker threads. For freqmine, the main thread is clearly the bottleneck although it performs quite a bit of parallel work with its worker threads. (3) The bottom right benchmarks are highly imbalanced. The main thread spawns three worker threads but performs little to no additional work. Hence the parallelism of the worker threads is limited to three and the parallelism of the main thread equals one.

VII. Related Work

We divide related work on methodologies to predict multi-threaded application performance into four groups: (1) modeling, (2) execution-driven simulation, (3) trace-driven simulation, analysis, and replay, and (4) online performance prediction.

Analytical multicore performance modeling: As mentioned in the introduction, there exists some prior work on analytical multicore performance prediction. Jongerius et al. [21] propose a multicore performance model that is limited to multi-program workloads, i.e., it does not model synchronization, coherence nor positive cache interference. Popov et al. [32] combines simulation results of representative code snippets with Amdahl's Law to provide performance predictions — RPPM on the other hand is a pure analytical model that relies on profiling, not simulation.

Execution-driven simulation: Simulating multi-threaded programs is tedious and time-consuming. Prior work proposes techniques to speed up the simulation of multi-threaded programs. Carlson et al. [4] propose a sampling methodology

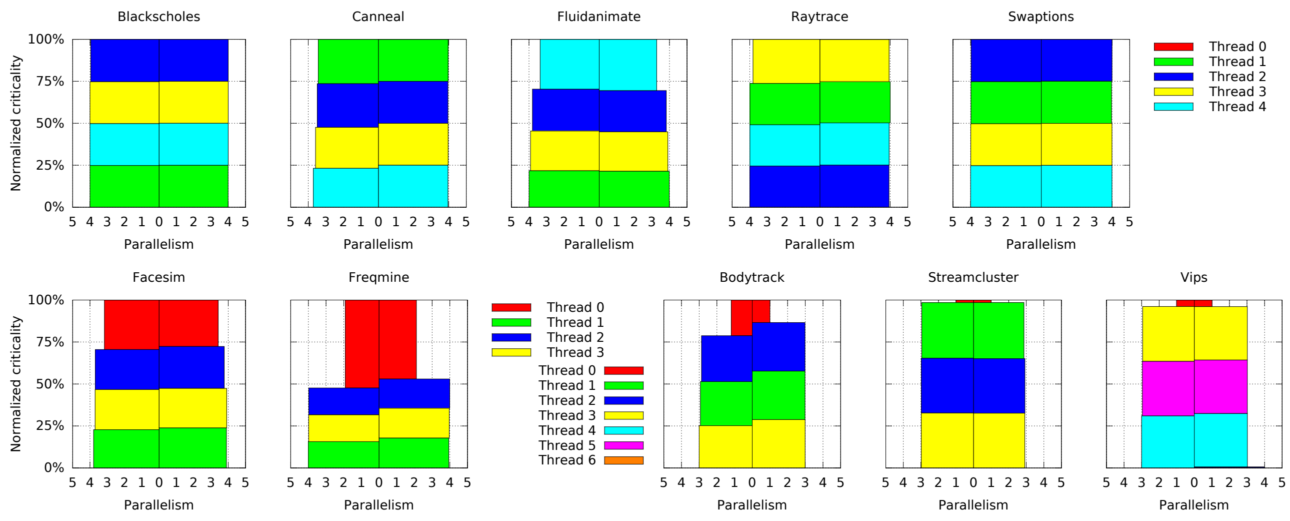

Fig. 6: Bottlegraphs for the PARSEC benchmarks: right-hand side for each graph reports the bottlegraph obtained through simulation; the left-hand side reports the bottlegraph obtained using RPPM. *RPPM accurately models the amount of parallelim and a thread's impact on overall application performance.*

to speed up the simulation of multi-threaded programs. Their approach uses pre-simulation analysis to guide sampling. They accurately model synchronization during fast-forwarding. In subsequent work, they propose a fast simulation methodology specifically targeting barrier-synchronized applications [6]. RPPM collects microarchitecture-independent metrics during a profiling run, which is at least one order of magnitude faster than simulation.

Trace-driven simulation, analysis and replay: Some prior work investigates trace-driven simulation and analysis methodologies for multi-threaded programs. Nilakantan et al. [29] propose a trace collection methodology for multi-threaded programs that is architecture-independent and retains synchronization behavior in the traces. Despite being faster than execution-driven simulation, their simulation approach is slower than our profile-driven mechanistic approach. PinPlay [31] is a tool for the deterministic recording and replay of multi-threaded program executions.

Online performance prediction: DEP+BURST [2] estimates performance at different frequency settings to steer DVFS by dividing execution time into epochs delineated by synchronization events. This prior work is limited to DVFS and does not enable making performance predictions across microarchitectures. Other prior work uses analytical models for understanding critical sections and scheduling heterogeneous multicores [9], [12], [13], [19], [20], [34], [35].

VIII. Conclusions

We proposed RPPM which takes microarchitecture-independent characteristics as input to predict performance of multi-threaded applications on a previously unseen multicore processor. RPPM extends prior work by modeling per-epoch active execution times per thread (including the impact of shared resource interference and cache coherence) and synchronization overhead due to barriers, critical sections and condition variables. RPPM predicts performance within 11.2% on average (23% max). Case studies illustrate RPPM's usefulness to evaluate multicore microarchitecture trade-offs and conduct application performance analysis.

Acknowledgements

We thank the reviewers for their constructive and insightful feedback. Sander De Pestel and Sam Van den Steen are supported through a doctoral fellowship by the Agency for Innovation by Science and Technology in Flanders (IWT). Additional support is provided by the European Research Council (ERC) Advanced Grant agreement No. 741097, and the FWO projects G.0434.16N and G.0144.17N.

References

[1] G. Åhlman. Microarchitecture-independent data locality analysis of multi-threaded applications on multicore processors. Master's thesis, Uppsala University, Division of Computer Systems, 2016.

[2] S. Akram, J. B. Sartor, and L. Eeckhout. DVFS performance prediction for managed multithreaded applications. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 12–23, Apr. 2016.

[3] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

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

[5] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout. An evaluation of high-level mechanistic core models. *ACM Trans. Archit. Code Optim.*, 11(3):28:1–28:25, Aug. 2014.

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

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, Oct. 2009.

[8] X. E. Chen and T. M. Aamodt. Hybrid analytical modeling of pending cache hits, data prefetching, and MSHRs. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 59–70, Dec. 2008.

[9] M. Curtis-Maury, A. Shah, F. Blagojevic, D. S. Nikolopoulos, B. R. de Supinski, and M. Schulz. Prediction models for multi-dimensional power-performance optimization on many cores. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 250–259, Oct. 2008.

[10] S. De Pestel, S. Eyerman, and L. Eeckhout. Micro-architecture independent branch prediction modeling. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 135–144, Mar. 2015.

[11] S. De Pestel, S. Van den Steen, S. Akram, and L. Eeckhout. RPPM: Rapid performance prediction of multithreaded applications on multicore hardware. *IEEE Computer Architecture Letters*, 17(2):183–186, 2018.

[12] K. Du Bois, S. Eyerman, J. B. Sartor, and L. Eeckhout. Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 511–522, June 2013.

[13] K. Du Bois, J. B. Sartor, S. Eyerman, and L. Eeckhout. Bottle graphs: Visualizing scalability bottlenecks in multi-threaded applications. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 355–372, Oct. 2013.

[14] D. Eklöv and E. Hagersten. StatStack: Efficient modeling of LRU caches. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 55–65, Mar. 2010.

[15] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A Mechanistic Performance Model for Superscalar Out-of-Order Processors. *ACM Transactions on Computer Systems (TOCS)*, 27(2):42–53, May 2009.

[16] A. Hartstein and T. R. Puzak. The optimal pipeline depth for a microprocessor. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, pages 7–13, May 2002.

[17] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, Dec. 1989.

[18] E. Ipek, S. A. McKee, B. R. de Supinski, M. Schulz, and R. Caruana. Efficiently exploring architectural design spaces via predictive modeling. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 195–206, Oct. 2006.

[19] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt. Bottleneck identification and scheduling in multithreaded applications. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 223–234, Mar. 2012.

[20] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt. Utility-based acceleration of multithreaded applications on asymmetric CMPs. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, pages 154–165, June 2013.

[21] R. Jongerius, A. Anghel, G. Dittmann, G. Mariani, E. Vermij, and H. Corporaal. Analytic multi-core processor model for fast design-space exploration. *IEEE Transactions on Computers*, 67(6):755–770, June 2018.

[22] T. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, pages 338–349, June 2004.

[23] B. Lee and D. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 185–194, Oct. 2006.

[24] B. Lee, D. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 249–258, Mar. 207.

[25] B. Lee, J. Collins, H. Wang, and D. Brooks. CPR: Composable performance regression for scalable multiprocessor models. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 270–281, Nov. 2008.

[26] B. C. Lee and D. M. Brooks. Illustrative design space studies with microarchitectural regression models. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 340–351, Feb. 2007.

[27] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, pages 190–200, June 2005.

[28] P. Michaud, A. Seznec, and S. Jourdan. Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 2–10, Oct. 1999.

[29] S. Nilakantan, K. Sangaiah, A. More, G. Salvadory, B. Taskin, and M. Hempstead. Synchrotrace: Synchronization-aware architecture-agnostic traces for light-weight multicore simulation. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 278–287, Mar. 2015.

[30] E. Ould-Ahmed-Vall, J. Woodlee, C. Yount, K. A. Doshi, and S. Abraham. Using model trees for computer architecture performance analysis of software applications. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 116–125, Apr. 2007.

[31] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–11, Apr. 2010.

[32] M. Popov, C. Akel, F. Conti, W. Jalby, and P. d. O. Castro. PCERE: Fine-grained parallel benchmark decomposition for scalability prediction. In *2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1151–1160, May 2015.

[33] K. K. Pusukuri, R. Gupta, and L. N. Bhuyan. Thread reinforcer: Dynamically determining number of threads via OS level monitoring. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 116–125, Nov. 2011.

[34] B. Su, J. Gu, L. Shen, W. Huang, J. Greathouse, and Z. Wang. PPEP: Online performance, power, and energy prediction framework and DVFS space exploration. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 445–457, Dec. 2014.

[35] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 253–264, Mar. 2009.

[36] S. Van den Steen and L. Eeckhout. Modeling superscalar processor memory-level parallelism. *Computer Architecture Letters*, 1(2):10–13, June 2018.

[37] S. Van den Steen, S. Eyerman, S. D. Pestel, M. Mechri, T. E. Carlson, D. Black-Schaffer, E. Hagersten, and L. Eeckhout. Analytical processor performance and power modeling using micro-architecture independent characteristics. *IEEE Transactions on Computers*, 65(12):3537–3551, Dec. 2016.