RPPM: Rapid Performance Prediction of Multithreaded Applications on Multicore Hardware

Sander De Pestel, Sam Van den Steen, Shoaib Akram, and Lieven Eeckhout

Abstract—This paper proposes RPPM which, based on a microarchitecture-independent profile of a multithreaded application, predicts its performance on a previously unseen multicore platform. RPPM breaks up multithreaded program execution into epochs based on synchronization primitives, and then predicts per-epoch active execution times for each thread and synchronization overhead to arrive at a prediction for overall application performance. RPPM predicts performance within 12 percent on average (27 percent max error) compared to cycle-level simulation. We present a case study to illustrate that RPPM can be used for making accurate multicore design trade-offs early in the design cycle.

Index Terms—Modeling, micro-architecture, performance, multi-threaded

1 Introduction

Simulation is the predominant methodology for computer architects to evaluate new processor architectures. Unfortunately, simulation is extremely time-consuming, especially when simulating multicore hardware. Analytical performance modeling is an attractive alternative especially at early stages of the design cycle to make high-level design decisions which can then later be refined through cycle-level simulation [6], [7], [9], [12]. The current state-of-the-art in (mechanistic) analytical performance modeling [12] collects microarchitecture-independent characteristics of an application, based on which it predicts performance for a range of previously unseen architectures. This prior work unfortunately is limited to single-core processors.

Straightforward extensions of this prior work towards multithreaded applications running on multicore hardware 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 24 and 21 percent, respectively, and a maximum error above 110 percent. There are two reasons for the poor accuracy: (i) it does not model contention in shared resources and (ii) it does not model synchronization overhead. Prior work in multicore performance prediction does not model these inherent multithreaded workload properties [8] or focuses on predicting application performance under strong scaling [10].

We propose RPPM for predicting multithreaded application performance on multicore hardware. A profiler collects a set of characteristics that capture the 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. A key feature of RPPM is that the profile needs to be collected only once, from which the performance of a range of multicore architectures can be predicted. Although the profile is measured during a particular multithreaded 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 cycle-level simulation for all the OpenMP multi-threaded Rodinia benchmarks. RPPM predicts performance within 12 percent on average (27 percent max error) for a quad-core processor. We demonstrate the usefulness of RPPM to quickly identify the optimum among five design points with the same peak performance (in operations per second).

2 Background

In this section, we provide a brief background on microarchitecture-independent analytical performance modeling for single-threaded applications; we refer the reader to [12] for a more elaborate exposition. We next describe naive extensions to this prior work to predict multithreaded application performance.

2.1 Single-Threaded Performance Model

The single-threaded model consists of two steps. In the profiling step, we use a Pin tool to collect an application profile containing only microarchitecture-independent statistics. 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 the following equation:

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

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.

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}) or the time between the branch being dispatched from the front-end pipeline into the back-end (issue queue and reorder buffer), and the branch being executed. Prior work profiles branch behavior in a microarchitecture-independent way using the information theoretic notion of entropy [4], 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 based on reuse distance distributions using StatStack [5].

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 product of the number of last-level cache misses due to load instructions (m_{LLC}) and 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 [11].

- *The authors are with the Ghent University, Gent 9000, Belgium. E-mail: {sander.depestel, sam.vandensteen, shoaib.akram, lieven.eeckhout}@ugent.be.*

Manuscript received 4 Apr. 2018; revised 29 May 2018; accepted 19 June 2018. Date of publication 1 July 2018; date of current version 9 Aug. 2018.
(Corresponding author: Sander De Pestel.)
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/LCA.2018.2849983

1556-6056 © 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

2.2 Naive Extensions for Multithreaded Applications

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

MAIN. In this 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 a prediction for the overall execution time of the multithreaded 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 of 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 multithreaded 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. RPPM models both synchronization and shared resource interference, as we describe next.

3 RPPM

RPPM predicts multithreaded application performance using two key components: (1) a profiler that collects microarchitecture-independent statistics including per-thread characteristics, shared memory access behavior and synchronization events, and (2) a rapid prediction tool that takes these statistics as input and predicts multithreaded execution time on a particular multicore processor architecture. Note 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 range of multicore architectures while varying clock frequency, pipeline width and depth, window and buffer sizes, cache sizes, etc.

3.1 Microarchitecture-Independent Profiling

Profiling is done using a Pin tool that collects a range of microarchitecture-independent statistics. Some of these are the same as in the single-threaded model, e.g., statistics that relate to an individual thread's execution such as branch behavior and ILP. To be able to model multithreaded execution performance, we in addition need to profile synchronization behavior as well as memory system behavior.

Synchronization. We track all synchronization events (barriers, critical sections, etc.) by tracking specific library function calls. More specifically, OpenMP lets the programmer mark a `for` loop with a `#pragma` telling the OpenMP runtime to execute the loop in parallel. The compiler will insert a function call (e.g., `gomp_team_barrier_wait`) to mark a barrier. We capture these function calls in the profiler and log the location of the call in the application's synchronization profile. To be able to distinguish different synchronization events, we track the function arguments. For example, the function `gomp_team_barrier_wait` will pass 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.

Multithreaded StatStack. In this work we use a multithreaded extension of StatStack [1] to estimate cache miss rates using a multi-threaded microarchitecture-independent reuse distance profile. StatStack collects a per-thread distribution of the reuse distance between two references (by any thread) to the same memory location. The extension to multithreaded applications enables predicting both positive and negative interference in shared caches as well as cache coherence effects. StatStack keeps track of the data accessed by all threads to create a profile about the memory behavior for each thread and how it impacts the memory behavior of other threads through the shared cache and the coherence protocol.

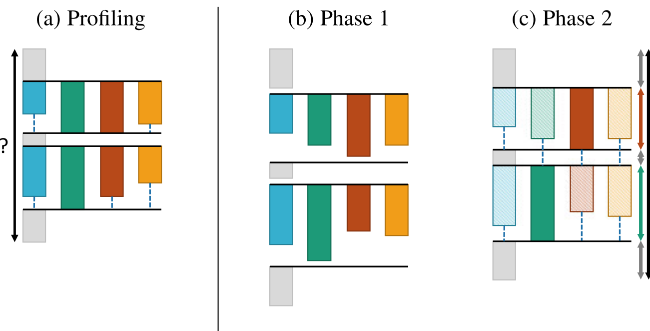

Fig. 1. RPPM predicts multithreaded execution time in three steps: (a) We profile an application's synchronization behavior and per-epoch statistics for each thread. We then predict an application's execution time (b) by predicting per-epoch active execution times for each active thread, and (c) by estimating the impact of synchronization on overall application performance.

Putting it Together. Fig. 1a illustrates how profiling is done. Synchronization events (barriers in this example) delineate different epochs. We collect a separate profile per epoch for each thread. This profile then serves as input to the prediction model, which we describe next.

3.2 Multithreaded Performance Prediction

The multithreaded performance model itself operates in two phases. The first phase (Fig. 1b) predicts the active execution time for each thread in-between synchronization events. The second phase (Fig. 1c) accounts for synchronization events and introduces predicted synchronization overhead to predict overall execution time.

Per-Epoch Active Execution Time. We use the microarchitecture-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 as interference may have a positive or negative impact on overall performance.

As mentioned before, we leverage a multithreaded extension of StatStack [1] to model shared caches and their impact on performance. In particular, for estimating the number of cache misses to a private L1 or L2 cache, StatStack checks whether the memory locations accessed by one thread are written by any other thread in-between the two accesses. If so, write invalidation is detected, which results in a cache miss for the second access. Because the access time to a private cache is relatively small, the model assumes this latency can be hidden to some extent through out-of-order execution, affecting the estimated effective dispatch rate (D_{eff}) in the base component in Equation (1).

To estimate the number of cache misses in the shared last-level cache (m_{LLC}), all memory accesses by all threads are taken into account. More specifically, StatStack considers the distribution of all reuse distances by all threads as input to predict the cache miss rate in the LLC. This accounts for both positive interference (one thread bringing in data for another thread, shortening average memory access time) and negative interference (one thread evicting data brought in by another thread, increasing average memory access time).

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

TABLE 1
Rodinia Benchmarks and Their Inputs

Benchmark	Input
Backprop	8388608
BFS	graph1MW_6
CFD	fvcorr.domn.010K
Heartwall	test.avi 10
Hotspot	16384 5
Kmeans	kdd_cup
LavaMD	10
Leukocyte	testfile.avi 5

Benchmark	Input
LUD	2048.dat
NN	4096k
NW	16k x 16k
Particlefilter	128 x 128 x 10
Pathfinder	1M x 1k
SRAD	2048
Streamcluster	256k

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.

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.

Algorithm 1. 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
```

Estimating synchronization overhead is done using Algorithm 1. 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. 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. We account for the following synchronization events:

- Thread creation: The main thread is created at application start-up time; 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 one 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 place in the code where all threads need to wait for each other to finish the execution of their respective code segment. When a thread arrives at a barrier it checks whether the conditions of the barrier are met. When the conditions are not met, 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.
- Thread joining: The behavior of a join is similar to a barrier with two threads, 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 thread.

TABLE 2
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				

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 this approach will be suitable for unlisted events like semaphores or even indirect synchronization.

This is further illustrated in Fig. 1c. 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.

4 Experimental Methodology

Benchmarks. We consider all the benchmarks from the Rodinia benchmark suite v3.1 [3]. We use the OpenMP implementations and predict the execution time of the parallel region of interest (ROI), which starts after initialization and ends before finalization by the main thread; multiple threads co-execute in the ROI.

Data Inputs. We select input data sets for all benchmarks that lead to reasonable simulation times while executing a sufficient number of instructions in the ROI, see Table 1. 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.

Simulator. We evaluate RPPM's accuracy as follows. We first simulate the benchmarks using the Sniper multicore simulator [2], which is a state-of-the-art, parallel and hardware-validated multicore simulator. We simulate the Base multicore configuration as specified in Table 2, unless mentioned otherwise. These simulated execution times results serve as the golden reference.

Profiling. We also profile the benchmarks and subsequently predict execution time for our benchmarks using RPPM for the exact same multicore architecture that we simulated using Sniper. We then compute the error between the simulated and predicted execution times. Profiling is done using the same number of threads on an Intel Xeon Sandy Bridge (E5-2420).

5 Evaluation

We compare RPPM against two naive extensions of the previously proposed single-threaded performance model, MAIN and CRIT, see Fig. 2. For MAIN, the execution time of the main thread is predicted and used as a prediction for overall application performance. This leads to an average absolute prediction error of 24 percent with several outliers above 40 percent. Predicting the execution time for all threads and then taking the execution time of the slowest thread (critical thread) as a prediction for overall application performance, as done by CRIT, brings the error down to 21 percent on average. CRIT improves prediction accuracy significantly for particlefilter

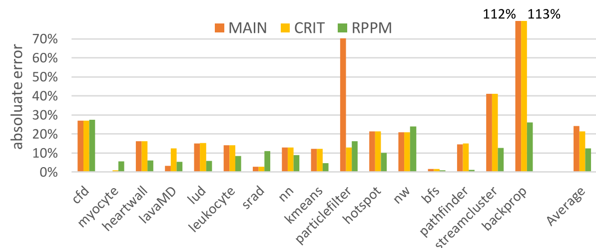

Fig. 2. Prediction error for MAIN, CRIT and RPPM.

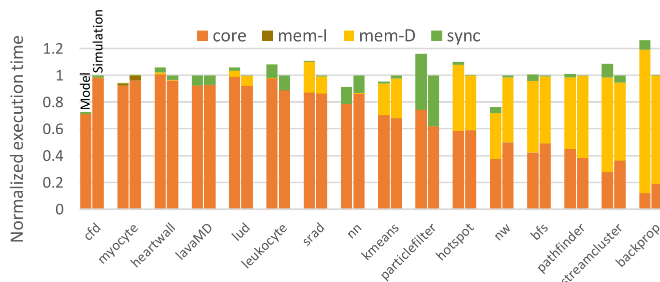

Fig. 3. Cycles stacks by RPPM (left) normalized to simulation (right).

which is highly imbalanced with the main thread being active for only 25 percent of the total execution time.

RPPM clearly outperforms MAIN and CRIT with an average absolute error of 12 percent and a maximum error of 27 percent. RPPM accurately predicts which thread is the most critical thread between synchronization events which leads to an overall more accurate performance prediction than MAIN and CRIT.

To help understand where the remaining error is coming from, Fig. 3 illustrates the average per-thread cycle stacks normalized to simulation. The remaining 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.

6 Case Study

We now consider the following case study to illustrate RPPM's usefulness. We profile each of the benchmarks once and predict performance for five different configurations as listed in Table 2. We change processor width from 2 to 6 (and scale ROB and issue queue resources accordingly) and change clock frequency from 5 to 1.66:GHz across these design points, while keeping the maximum number of operations that can be executed per second constant.

We use RPPM to identify the design points that are within a bound of x% of the predicted optimum, see Table 3. If the bound is set to 0 percent, only the best design point is identified by RPPM. If the bound is larger then 0 percent, all design points within the bound are identified by RPPM and simulation will select the best one. The average deficiency (performance difference) versus the real optimum is 1.95 percent (see bottom row) and up to 19.1 percent for streamcluster. Setting a higher bound of 5 percent 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 percent for pathfinder.

7 Conclusions

In this paper, we proposed RPPM which takes microarchitecture-independent characteristics as input to predict performance of multithreaded applications on a previously unseen multicore platform. RPPM extends prior work by modeling per-epoch active execution times per thread (including the impact of shared resource interference and cache coherence on per-thread performance) and synchronization overhead due to barriers and critical sections. RPPM predicts performance within 12 percent on average (27 percent max). A case study illustrates RPPM's usefulness to evaluate multicore microarchitecture trade-offs.

TABLE 3
Case Study: Predicting the Optimum Design Point

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%	

Acknowledgments

Sander De Pestel is supported through a doctoral fellowship by the Agency for Innovation by Science and Technology in Flanders (IWT). Additional support is provided through the European Research Council (ERC) Advanced Grant agreement no. 741097.

References

[1] G. Åhlman, "Microarchitecture-independent data locality analysis of multithreaded applications on multicore processors," Master's thesis, Division Comput. Syst., Uppsala Univ., Uppsala, Sweden, 2016.

[2] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Trans. Archit. Code Optimization*, vol. 11, no. 3, pp. 28:1–28:25, Aug. 2014. [Online]. Available: http://doi.acm.org/10.1145/2629677

[3] 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 *Proc. IEEE Int. Symp. Workload Characterization*, Oct. 2009, pp. 44–54.

[4] S. De Pestel, S. Eyerman, and L. Eeckhout, "Micro-architecture independent branch prediction modeling," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, Mar. 2015, pp. 135–144.

[5] D. Eklöv and E. Hagersten, "StatStack: Efficient modeling of LRU caches," in *Proc. Int. Symp. Perform. Anal. Syst. Softw.*, Mar. 2010, pp. 55–65.

[6] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A mechanistic performance model for superscalar out-of-order processors," *ACM Trans. Comput. Syst.*, vol. 27, no. 2, pp. 42–53, May 2009.

[7] E. Ipek, S. A. McKee, B. R. de Supinski, M. Schulz, and R. Caruana, "Efficiently exploring architectural design spaces via predictive modeling," in *Proc. 12th Int. Conf. Archit. Support Program. Languages Operating Syst.*, Oct. 2006, pp. 195–206.

[8] R. Jongerius, A. Anghel, G. Dittmann, G. Mariani, E. Vermij, and H. Corporaal, "Analytic multi-core processor model for fast design-space exploration," *IEEE Trans. Comput.*, vol. 67, no. 6, pp. 755–770, Jun. 2018.

[9] B. Lee and D. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *Proc. 12th Int. Conf. Archit. Support Program. Languages Operating Syst.*, Oct. 2006, pp. 185–194.

[10] M. Popov, C. Akel, F. Conti, W. Jalby, and P. D. O. Castro, "PCERE: Fine-grained parallel benchmark decomposition for scalability prediction," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, May 2015, pp. 1151–1160.

[11] S. Van den Steen and L. Eeckhout, "Modeling superscalar processor memory-level parallelism," *Comput. Archit. Lett.*, vol. 1, no. 2, pp. 10–13, Jun. 2018.

[12] 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 Trans. Comput.*, vol. 65, no. 12, pp. 3537–3551, Dec. 2016.