RpStacks-MT: A High-throughput Design Evaluation Methodology for Multi-core Processors

Hanhwi Jang*, Jae-Eon Jo*, Jaewon Lee† and Jangwoo Kim†
*Department of Computer Science and Engineering, POSTECH
†Department of Electrical and Computer Engineering, Seoul National University

***Abstract*—Computer architects put significant efforts on the *design space exploration* of a new processor, as it determines the overall characteristics (e.g., performance, power, cost) of the final product. To thoroughly explore the space and achieve the best results, they need *high design evaluation throughput* – the ability to quickly assess a large number of designs with minimal costs. Unfortunately, the existing simulators and performance models are either *too slow* or *too inaccurate* to meet this demand. As a result, architects often sacrifice the design space coverage to end up with a sub-optimal product.**

To address this challenge, we propose *RpStacks-MT*, a methodology to evaluate multi-core processor designs with high throughput. First, we propose a *graph-based multi-core performance model*, which overcomes the limitations of the existing models to accurately describe a multi-core processor's key performance behaviors. Second, we propose a *reuse distance-based memory system model* and a *dynamic scheduling reconstruction method*, which help our graph model to quickly track the performance changes from processor design changes. Lastly, we combine these models with a state of the art design exploration idea to evaluate multiple processor designs in an efficient way. Our evaluations show that RpStacks-MT achieves extremely high design evaluation throughput – 88× higher versus a conventional cycle-level simulator and 18× higher versus an accelerated simulator (on average, for evaluating 10,000 designs) – while maintaining simulator-level accuracy.

***Index Terms*—Design space exploration, Performance analysis, Simulation**

I. Introduction

Design space exploration is the most critical step in the development of a processor, because it shapes the overall characteristics of the end product. A key to successful exploration is to examine as *many* designs as possible, investigate the tradeoffs, and select the optimal candidates.

Unfortunately, the exploration is becoming increasingly challenging due to 1) the *growing complexity* of design space and 2) the *slowing down* of design evaluation.

First, the design space of modern multi-core processors is becoming extremely large and complex. Fig. 1 is a simplified example of the design space. To select the best design candidates, we must *thoroughly* explore the space and investigate the tradeoffs. Unfortunately, modern processor cores consist of complex modules (e.g., functional units, buffers, caches) which architects must explore with many different parameter combinations. The total number of combinations for a core easily reaches thousands. Moreover, as the cores get diverse *within* a single processor (e.g., heterogeneous multi-core processors), the number of designs grows even further.

Module	L1I	L1D	L2	MEM	DIV	MUL	FPADD	FPMUL
Optimization cost	9	9	40	40	9	3	2.5	3.5

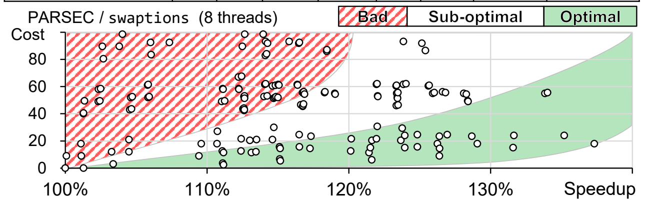


Fig. 1: Design space of a modern processor. For simplicity, we apply 1∼4 optimizations from the table with different combinations (e.g., L1D, L2, Div). We must carefully explore the space and pick optimal candidates for a successful development. Note that maximizing the number of optimal designs is also important as they may have different performance characteristics for other applications.

Second, the design evaluation of multi-core processors, while already prohibitively slow, is getting even slower. As the number of cores per processor increases with each core becoming more complex, the simulation load significantly increases to model the increasing inter-core communications and microarchitecture modules. As a result, modern detailed multi-core simulators are multiple orders of magnitude slower than real hardware.

To address these challenges and achieve high *design evaluation throughput* (i.e., the number of designs evaluated per time and machine resources), architects have proposed various optimization techniques. The first group of studies accelerate simulations with binary translation and parallelization [1]–[3], sampling and load reduction [4]–[11], and hardware acceleration [12]–[15]. While they successfully increase the evaluation speed and efficiency, the large design space eventually *overwhelms* their speedup benefits (Section II-A).

Another group of studies *predict* a design's performance with analytic performance models to avoid expensive simulations [16]–[26]. These models provide a rich set of information regarding the current processor design's performance (e.g., executed operations and their latencies) so that architects can understand the performance bottlenecks and instantly predict the new performance upon alleviating the bottlenecks (i.e., design change). However, we discover that a large portion of these estimations are *inaccurate* because the predictions based on the current design's behaviors often become invalid for the other designs (i.e., different designs have different performance-determining behaviors). A recent work [24] significantly improves the accuracy by simultaneously considering multiple performance-determining behaviors, but its applicability is limited to only single-core processor designs and operation latency adjustments (Section II-B).

In this paper, we propose *RpStacks-MT*, a methodology to evaluate a large number of multi-core processor designs with *high-throughput* and *high-accuracy*, using representative execution paths.

We first analyze the limitations of the existing simulators and analytic models to clarify the requirements for a high-throughput evaluation. The results show that it is necessary to minimize the *per-design evaluation overhead*, and we therefore utilize an analytic model-based prediction to achieve this goal.

Further investigations on recent analytic model studies show that *none* of the existing proposals properly handle performance-critical multi-core behaviors such as resource sharing and dynamic scheduling, and this is the main obstacle preventing analytic model-based multi-core evaluation. To address this problem, we propose 1) a graph-based performance model for multi-core processors, 2) a reuse distance-based memory system model, and 3) a dynamic scheduling reconstruction method.

Specifically, we first represent a multi-core processor's behaviors with a stream of *graph blocks*, where each block represents a code snippet (or a thread) running on a core. Inter-core/thread interactions such as synchronizations are modeled as the dependencies between these graph blocks. To perform *high-throughput* design evaluations with this model, we start by estimating each block's execution performance for *multiple* processor designs using a state of the art design evaluation method, which exploits *representative execution paths* (*RpStacks*, Section II-B3) [24]. To account for key multi-core performance behaviors and maintain *high accuracy*, we develop 1) the memory system model which describes shared resource behaviors and estimates the contention effects and 2) the scheduling reconstruction method which (re-)generates thread (i.e., graph block) execution scenarios for new processor designs.

The evaluation using PARSEC benchmark suite [27] shows that RpStacks-MT achieves 88× higher throughput compared to a conventional cycle-level simulator [28] and 18× higher throughput compared to an accelerated simulator (evaluating 10,000 designs), while providing highly accurate results. We also emphasize that the nature of our method allows highly parallel analysis *scaling beyond* the number of cores of the target design (e.g., 16× analysis speed for an 8-core target design); the users therefore may trade-off the evaluation throughput for faster individual design analysis.

The contributions of this paper are as follows:

- **Graph-based multi-core performance model.** We propose a novel graph-based performance model which can accurately describe the performance-critical factors (e.g., shared resource, scheduling) of multi-core processors. To the best of our knowledge, this is the *first attempt* to analytically model multi-core processors in a detailed microarchitectural level (i.e., pipeline stages) with graphs.
- **Efficient reuse distance-based memory system model.** We propose an efficient reuse distance-based memory system model to quickly estimate the shared resource (i.e., LLC, interconnect) performance for multiple designs. The model greatly enhances the capability and accuracy of our design evaluation without incurring significant overheads.
- **Dynamic scheduling reconstruction method.** We propose a method to reconstruct dynamic scheduling behaviors from our graph-based model. The method allows us to accurately track the performance of different multi-core designs with different scheduling behaviors.
- **High-throughput design evaluation method.** We combine the models and methods introduced above to construct a holistic high-throughput multi-core design evaluation method. The method assesses a large number of designs while maintaining simulator-level accuracy, to allow the users to efficiently investigate tradeoffs and select optimal design candidates.

In Section II, we discuss the simulators and performance models to identify their limitations and motivate our approach. In Section III, we perform further analysis on the state of the art analytic models and design space exploration methods to clarify the design goals. We explain the key ideas of RpStacks-MT in Section IV and describe the implementation details in Section V. Section VI shows the accuracy and throughput evaluation results, and Section VII provides related works. We discuss how to improve RpStacks-MT in Section VIII and conclude the paper in Section IX.

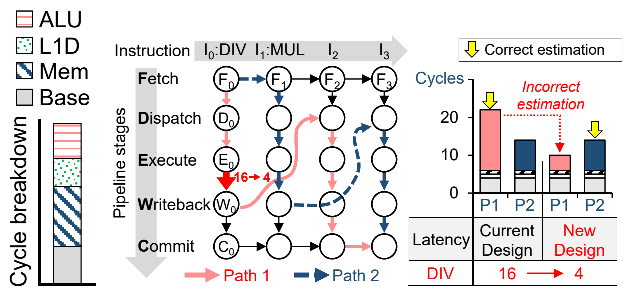
II. Background and Limitations

A. Detailed Cycle-level Simulators

Simulators are by far the most popular tool to model and evaluate a processor design. We now introduce various simulation methods as well as their limitations.

1) Single-threaded cycle-level simulators: The single-threaded simulators such as Flexus [29], gem5 [30], MARSSx86 [28], and Multi2Sim [31] accurately model a multi-core processor by calculating its microarchitectural state for every cycle. While they provide the best accuracy, they are also the *slowest* due to the complexity of the model. In addition, they show (at least) linear slowdown for multi-core designs because a single thread simulates all the cores of the target processor design. Accordingly, they are suitable for detailed performance debugging (e.g., analyzing pipeline bubbles) rather than wide design space exploration.

2) Binary translation / parallel simulators: Recent multi-core simulators (e.g., ZSim [1], Sniper [2], Graphite [3])



(a) Example result

(b) A graph-based processor performance model

Fig. 2: Analytic performance models.

achieve extremely high speed from dynamic binary translation and parallelization. Specifically, they adopt faster functional behavior models (i.e., native execution and instrumentation [32] instead of emulation) and parallelize the simulation with coarser-grain inter-core synchronizations.

While they successfully boost the speed of a single multi-core simulation, they become less effective in terms of *evaluation throughput*. First, the speedup benefits from parallelization disappear if the number of designs dominates the total simulation load. For example, to maximize the evaluation throughput with an 8-core simulation host (i.e., limited resource), it is better to run eight single-threaded simulations to evaluate eight different designs rather than an 8-thread simulation covering a single design, because the speedups from parallelizations are often sub-linear. The speedup from native execution remains valid, however, is often insufficient to cover large design spaces with reasonable costs (Section II-C).

3) Hardware-accelerated simulators: Hardware accelerated simulators [12]–[15] utilize hardware prototyping platforms (e.g., FPGAs) to achieve near-native speed. However, they generally have high design modification overheads and therefore are not suitable for evaluating a large number of designs. We thus do not discuss them further in this paper.

B. Analytic Performance Models

Analytic performance models provide a rich collection of information regarding the current processor design's behavior. Fig. 2a presents a typical outcome of an analytic model. The stacked-bar shows the number of operations performed along with the amount of time spent on each operation. Architects can understand the bottlenecks and *predict the new performance* upon alleviating the bottlenecks (via design changes) by *scaling the stack components*.

Since the performance prediction requires only a simple scaling operation, analytic models can estimate the performance of multiple processor designs with *negligible per-design overheads*. In other words, with one-time analysis overhead, the models can evaluate multiple designs with extremely high throughput. We now discuss the limitations of the models to illustrate why they *cannot* be used for multi-core designs despite their throughput advantages.

1) Graph-based models: Graph-based analytic models [20]–[22], [26], [33], [34] describe a workload's instruction flow and microarchitectural behaviors as a graph's edges and nodes (Fig. 2b). The nodes represent the pipeline stages of instructions and the edges connecting the nodes show the dependencies between the pipeline stages; the edge weights indicate the operation latency. For example, if two nodes are connected with an edge with weight N, the operation in the source node takes N cycles to finish and it triggers the destination node's operation to start. By finding *the longest path* from the first stage of the first instruction to the last stage of the last instruction (i.e., critical path), we can derive the cycles it takes to execute all the instructions (i.e., performance) and identify where the cycles are spent (i.e., bottlenecks).

In this graph model, a new processor design is represented as a graph with new edge weights (i.e., new operation latencies). To calculate the performance of this new design, we can either 1) scale the components in the current design's performance stack (i.e., apply the new latencies) assuming the critical path would not change, or 2) find the new critical path from the new graph. The first option estimates the performance with negligible overheads, however, may *produce inaccurate results* if the critical path changes upon weight changes. Our example (Fig. 2b) shows such a case where we miss the new (formerly 2nd) critical path and overestimate the performance. The second method accurately derives the new performance but requires a full re-traversal of the graph model to find the new critical path. Considering typical graph and simulation size (i.e., millions of nodes representing millions of instructions), this would incur simulation-level overheads and therefore cannot be a scalable solution. To summarize, the existing graph-based models are not suitable for accurate and efficient design evaluation.

2) Counter-based models: Counter-based analytic models [35]–[37] use *hardware performance counters* to analyze the performance of the current design. They provide a stack similar to Fig. 2a to describe the results. Thanks to the low overhead, these models are widely employed to assist real-time performance optimization schemes [19].

However, similar to the case of the graph models, counter-based analyses also utilize only the current critical path. We may estimate the performance of new designs but should expect inaccurate results. We therefore conclude this model is not acceptable for accurate design evaluation.

3) State of the art design space exploration: The conventional graph-/counter-based analytic models make inaccurate predictions because they estimate the performance using *only* the current critical execution path.

To address this problem, a recent proposal [24] suggests to remember *multiple execution paths* from a graph-based model (e.g., Fig. 2b), and utilize them to accurately predict the performance of new designs. Since the total number of execution paths is extremely large, the authors propose various techniques to identify the *representative* paths which are likely to become the critical path.

Unfortunately, we notice that this approach is limited to

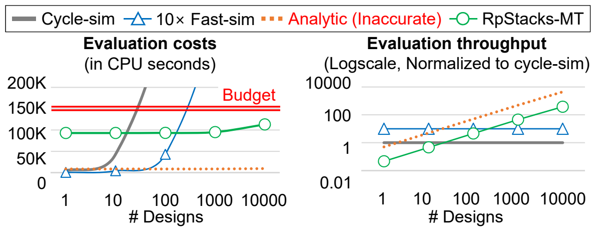


Fig. 3: Design evaluation costs and throughput of the discussed methods, for various number of designs. Each design requires a 1 billion-instruction simulation.

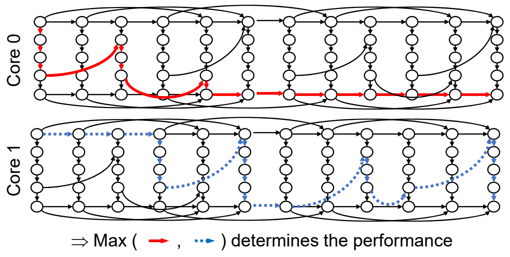


Fig. 4: Naive expansion of a single-core graph-based performance model to multi-core processors.

single-core processor designs. It does not discuss how to handle multiple cores and threads showing dynamic scheduling behaviors. In fact, to the best of our knowledge, *none of the existing studies* discuss graph-based performance models for multi-core processors. Furthermore, we discover that naively expanding the existing single-core models results in highly inaccurate results (Section III). We therefore aim to define an accurate graph-based multi-core performance modeling method and utilize it to effectively explore the large design space.

C. *Lessons from the Analyses*

Fig. 3 summarizes the design evaluation cost and throughput for the methods discussed in this section.

First, simulators have *fixed evaluation throughput* as each design requires a simulation. Accordingly, they should pay more simulation costs to explore larger number of designs. As a result, even accelerated simulations (e.g., 10× faster) cannot thoroughly cover the large design space of modern multi-core processors (1000+ designs) with reasonable budgets. For a realistic example, a single-day simulation using 10,000 vCPU cores (i.e., 10,000 simulation instances) costs approx. 10K USD on Amazon EC2 (0.0425 USD/vCPU-hour [38]); we need a game-changing solution to reduce the costs to a manageable level (e.g., 88× of RpStacks-MT making it ≃100 USD/day). To handle this ever-growing design space, we aim to develop a method which is *less sensitive* to the number of designs.

Fortunately, we confirm that the analytic model-based estimation has negligible per-design evaluation overheads. Therefore, despite it pays high initial analysis overheads, it has *increasing evaluation throughput* for larger number of designs (Fig. 3). Since we identify that its major problems are 1) the low accuracy from the lack of multiple performance-critical path considerations and/or 2) the absence of proper multi-core processor model, we decide to develop a new performance model for multi-cores and improve the multiple-path consideration method [24] accordingly to propose RpStacks-MT – a high-throughput multi-core design evaluation method.

III. Problem Analysis & Motivation

As we decide to evaluate designs with analytic performance models, we further investigate the problems of the state of the art models in evaluating multi-core processors. We then use the insights to set the design goals for RpStacks-MT.

A. *State of the Art Model: Naïve Expansion to Multi-cores*

We first introduce how the state of the art analytic model and design evaluation method can be *naively* applied to multi-core designs. As illustrated in Fig. 4, we describe a multi-core processor as a collection of cores and apply an existing graph-based single-core model and analysis methods to individual cores. We find the performance-determining execution path for each core, and use the longest one to determine the overall performance (i.e., the slowest core determines the speed).

B. *Limitations of the Naïve Expansion*

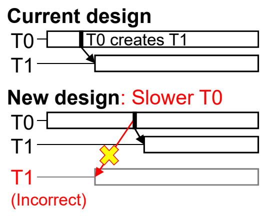
We now discuss why simply expanding the single-core model fails to accurately estimate multi-core performance.

1) Lack of shared resource considerations: The existing graph-based single-core processor models *cannot* accurately consider resource sharing behaviors (e.g., shared cache and interconnects) which have a major impact on a multi-core design's performance. As shown in Fig. 4, there is no mechanism to inform the other cores of the changes in the shared resource status.

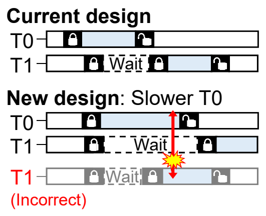
The fundamental limitation is that these shared resource behaviors are *hard-coded in the graphs*. As an instance, memory instructions are marked with the outcomes from the baseline design (e.g., LLC hit, bus latency 10 cycles), whereas they may change for new designs (e.g., shrinking the cache makes the LLC hit to become a miss). We therefore need to quickly reconstruct such shared resource behaviors for new designs and accurately apply the results to the graph-based models.

2) Lack of dynamic scheduling considerations: The existing analytic models lack the description of scheduling behaviors which often determine the first-order performance of multi-core processors. We categorize the scheduling behaviors and explain how each can cause a problem.

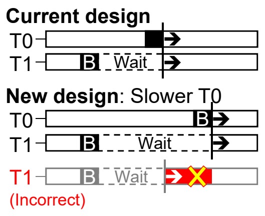
Thread creation. Multi-core processors often run multi-threaded applications which dynamically create new threads. Fig. 5a illustrates a scenario where a thread (`T0`) spawns another thread (`T1`). If a design change slows down `T0`'s progress, `T1`'s start time should be adjusted accordingly. However, as the naive model has `T1`'s start time statically embedded, it cannot account for the delay from the design



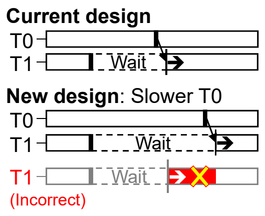
(a) Thread creation



(b) Lock



(c) Barrier



(d) Conditional variable

Fig. 5: Naively expanded analytic model: Inaccurate modeling of performance-critical thread interactions.

change. As a result, we estimate the performance from an incorrect scenario where `T1` runs ahead of its creation time.

Lock. Locks form critical sections where only one thread may enter and proceed; the other threads must wait until the currently occupying thread leaves the section.

Fig. 5b shows an example where changing the processor design alters the lock related behaviors. In this case, the new design slows down `T0`, and `T1` should wait longer for the lock release. However, the naive model has statically marked *wait* behavior for `T1` which results in overlapping critical sections. Again, we fail to correctly model the performance behaviors.

Barrier. Barriers synchronize multiple threads' progress at a certain point in an application. A thread that reaches a barrier waits for the other threads to arrive; when all the threads arrive at the barrier, they resume the execution.

Fig. 5c shows a case where the barrier waiting duration changes upon a design change. The new design slows down `T0`, making `T1` wait longer. The naive model cannot reproduce this behavior, as the barrier waiting delay is hard-coded in the graph model.

Conditional variable. Conditional variables allow threads to pause and resume if certain requirements are met. For example, in a producer-consumer pattern of data processing, the consumer thread halts if there is no data to process and wakes up if the producer generates the data.

Fig. 5d illustrates an example where a design change slows down the producer thread (`T0`). Accordingly, the consumer thread (`T1`) should wait longer than before. The naive model incorrectly wakes up the consumer based on the wakeup time measured from the current (i.e., baseline) architecture.

C. *Design Goals of RpStacks-MT*

The fundamental problem of the naively expanded model is that it does not consider / re-generate important multi-core behaviors such as resource sharing and dynamic scheduling. We observe their performance impacts vary significantly depending on the performance (i.e., thread progress) of a new design.

To accurately model multi-core processor performance, we therefore aim to develop 1) a graph-based analytical performance model which efficiently embeds the shared resource and scheduling information and 2) models and methods which use the information to quickly (re-)construct the new performance behaviors upon processor design changes.

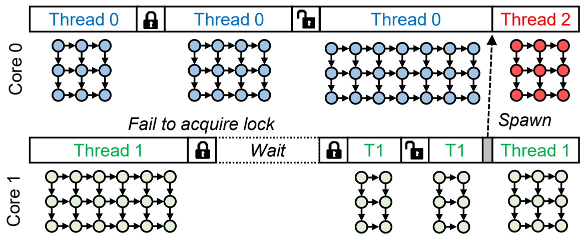


Fig. 6: Graph-based multi-core processor performance model.

IV. RpStacks-MT: Key Ideas

This section introduces the key ideas of RpStacks-MT, which overcome the challenges discussed in Section III and enable accurate high-throughput multi-core design evaluation.

A. *Graph-based Multi-core Model*

We first present a novel graph-based performance model for multi-core processors (Fig. 6).

First, instead of generating a large single graph per core (Fig. 4), we create a *graph block* per code snippet (of a thread) which does not have any scheduling or synchronization events. The scheduling and synchronization events are separately logged as the *dependency* between these blocks, so that we can later reschedule the graph blocks (i.e., code snippets of threads) while preserving the original synchronization semantics.

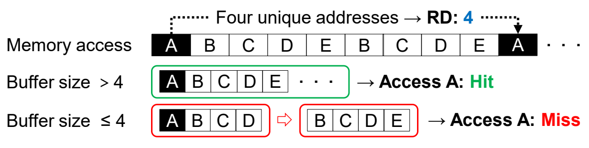
Second, for each graph block, we keep the track of *reuse distances* [39]–[42] to efficiently log memory system (i.e., shared resource) behaviors. Compared to full memory access traces, reuse distance enables quick performance estimation for various memory system designs and resource sharing scenarios.

Lastly, we improve the pipeline details of the graph model to better describe multi-core specific performance events. The details are provided in Section V-C.

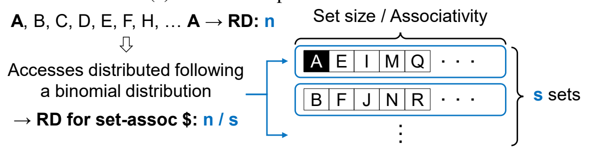
B. *Reuse Distance-based Memory System Model*

To accurately estimate the performance implications of resource sharing behaviors, we develop an efficient *reuse distance-based memory system model* (Fig. 7).

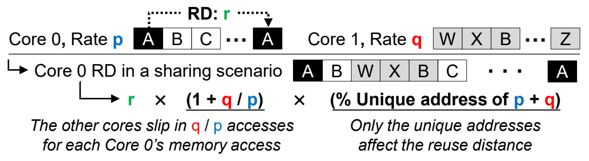
Reuse distance basics. Reuse distance is the number of *unique* addresses between two references to an identical address (Fig. 7a) [39]. Assuming a cache-like buffer with the LRU replacement policy, the second access to the address will hit on



(a) Basic concept of reuse distance

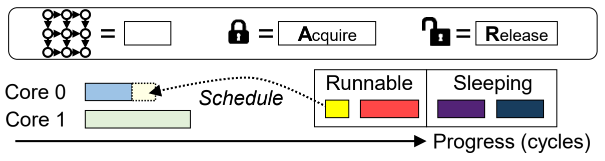

A, B, C, D, E, F, H, ... A → RD: n
Accesses distributed following a binomial distribution
→ RD for set-assoc $: n / s
Set size / Associativity
A E I M Q · · ·
B F J N R · · ·
s sets

(b) Modeling a set-associative cache

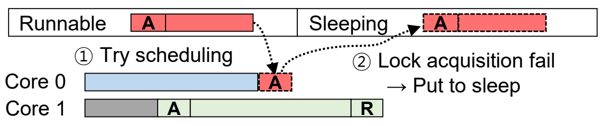

RD: r
Core 0, Rate p
A B C ··· A
Core 1, Rate q
W X B ··· Z
Core 0 RD in a sharing scenario
A B W X B C · · · A
r × (1 + q / p) × (% Unique address of p + q)
The other cores slip in q / p accesses for each Core 0's memory access
Only the unique addresses affect the reuse distance

(c) Modeling a shared cache

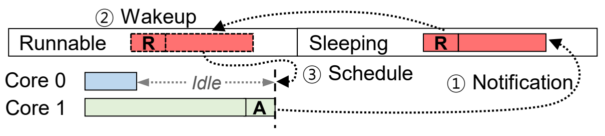
Fig. 7: Reuse distance-based memory system model.



(a) Basic scheduling logic



(b) Runnable block going to sleep



(c) Sleeping block becoming runnable

Fig. 8: Dynamic scheduling reconstruction.

the buffer *only if* the reuse distance is smaller than the buffer size. With this technique, we can quickly identify the hit/miss of accesses for various buffer sizes without simulations.

Modeling a set-associative cache. We model a set-associative cache with reuse distance as follows (Fig. 7b). First, the set's size (i.e., associativity) determines the buffer size because multiple addresses contend for this space. Second, the cache accesses are distributed across multiple sets following a multinomial distribution; we utilize the analysis in [40] to construct this model[1]. As a result, we accurately estimate the hit/miss of a memory access stream on a set associative cache.

Modeling a shared cache. A shared cache has multiple access streams contending for a single set (Fig. 7c). In general, the reuse distance of a stream *increases* as the accesses from the other streams would slip in between two accesses [42]. The amount is determined by 1) the (relative) memory access rates of the streams and 2) the address overlaps between the streams [41] (i.e., accesses to the same addresses would not increase the reuse distance; only the *unique* addresses matter). With this model, we accurately adjust the reuse distance of each stream to account for the contention effects.

Modeling a shared interconnect. The interconnect (i.e., bus) of a memory system is another critical shared resource of multi-core processors. We therefore develop a *queue-based* interconnect model to accurately consider resource contentions. Specifically, we model a bus as an *M/D/1 queue* [43], [44][2], where the requests arrive following a Poisson process (with rate λ) and a single server (i.e., bus) handles them with a fixed rate (μ). The *average wait time* of the requests is then expressed as:

[1]Multinomial distribution implies uniform accesses to the cache sets. To better account for workload-specific patterns (e.g., sequential or stride), we may profile the cache set access frequencies for workloads and use the corresponding distributions to express the non-uniformity.

[2]To model network-on-chip (NoC) based interconnects, we may replace our M/D/1 queue with more complex models suggested in [45]–[47].

$$Average\ wait\ time = \lambda\ /\ 2\mu(\mu - \lambda).$$

In other words, the wait time increases as the request rate (λ) approaches the handling rate (μ). The handling rate is derived for each workload using the baseline design (i.e., workload's nature) and the request rate is calculated for each graph block (i.e., # memory instructions / # total instructions). We confirm that this setting precisely estimates how the bus latency changes for new designs.

Estimating the performance impact. Typical reuse distance and queueing models focus on accurately estimating the hit/miss rates and delays of the whole system. In our case, we are more interested in how they affect *individual execution paths'* performance. We now introduce how we translate these statistics to the performance.

For caches, we measure how the number of accesses to each hierarchy (i.e., L1, L2, MEM) changes, for each execution path (of a graph block). We then re-impose the penalties according to the new access counts. For example, if an L1 hit changes to an L2 hit in a new design, we take out one L1 hit latency and add one L2 hit latency. Using this method, we can accurately adjust the memory access latencies for new cache behaviors.

For shared interconnect, we first calculate the average wait time per *graph block* because all the paths in a block experience the same contention and thus the same latency. We then impose interconnect latencies per path; the total latency is defined as *# accesses to interconnect* $\times$ *average wait time*. This approach accurately derives the latencies for individual paths.

C. Dynamic Scheduling Reconstruction Method

The performance impact of synchronization and scheduling (e.g., load-balancing) is dynamically determined at runtime according to the performance of the processor (Section III-B2). To accurately model this non-deterministic effect and find the critical execution path spanning across multiple cores, we

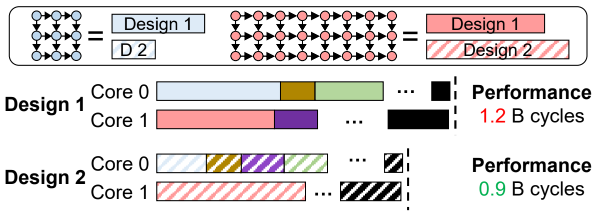


Fig. 9: High-throughput Design Evaluation

develop a *dynamic scheduling reconstruction* method which generates realistic thread execution scenarios (Fig. 8).

1) Basic idea: Fig. 8a describes our basic scheduling logic. We schedule the graph blocks to processor cores to model multiple threads running on a multi-core processor. The graph blocks to be scheduled are either in a *runnable* or a *sleeping* state. The runnable blocks may immediately execute on a core while the sleeping blocks must wait to be activated. We iterate over the cores, find the core with the slowest progress (i.e., smallest cycle count), and schedule a runnable block to the core; scheduling the block advances the core's progress (cycles). This iterative scheduling ensures even progress across the cores; this is what the majority of modern OSes and schedulers try to achieve when scheduling multiple threads to multiple cores. Other scheduling or load-balancing algorithms can be implemented in this step to reproduce their corresponding scheduling behaviors.

2) Runnable block going to sleep: A runnable block violating the scheduling or synchronization semantics is put to sleep because it cannot be executed immediately. Fig. 8b shows a runnable block with a lock; it cannot execute on Core 0 as the block in Core 1 already acquired the lock. The scheduling failure makes this block to sleep.

As we fail to schedule a block to Core 0, we would look up for another runnable block. If there is no block available for scheduling, we move on to the next slowest progress core and try scheduling. In Fig. 8b's example, we would move onto Core 1 for scheduling; the following section describes what happens next.

3) Sleeping block becoming runnable: Some blocks end with a special synchronization or scheduling event (e.g., lock release, thread spawn) which wakes sleeping blocks up and make them runnable again.

In Fig. 8c, Core 1's last block ends with a lock release and it notifies the block sleeping due to this specific lock. As the block becomes runnable again, we may schedule it on either Core 0 or 1. Note that this block must execute *after* the synchronization or scheduling event because it is what triggers the block to run.

4) Handling complex synchronizations: We now describe how the proposed method accurately handles the complex synchronizations discussed in Section III-B2. We exclude the lock as we already described it (i.e., Figs. 8b and 8c).

Thread creation. By default, all the threads except the very first thread start in a sleeping state. As the initial thread creates a new thread, its block ends with a special event: thread spawn. This event triggers the new thread to become runnable, and the new thread starts to run after the event.

Barrier. When a barrier synchronization variable is initialized, we remember *how many* threads are participating in the barrier. Later when a block (of a thread) reaches the special event: *barrier wait* , we put all the blocks of the corresponding thread to sleep to make them wait for the others to reach the barrier. When the *last* barrier wait operation appears, we wake up all the blocks that were previously put to sleep. This is feasible as we track which blocks/threads are participating in each barrier.

Conditional variable. Conditional variables are relatively straightforward to consider. If a block ends with a special event: signal, we simply wake the corresponding threads up. Note that unlike real systems which simply discard the signal if there is no recipient, we keep the signal until the recipient appears and consumes it, to prevent any deadlocks or similar events (e.g., all threads going to sleep).

D. High-throughput Design Evaluation

Lastly, we introduce how we combine our models with a novel high-throughput design evaluation method (Fig. 9).

First, for each graph block, we extract the *representative execution paths* following the idea of [24] (Section II-B3). The paths provide detailed performance bottleneck information in the form of cycle stacks, so that we can instantly estimate a graph block's execution performance (i.e., the cycle it takes to execute the code snippet represented by the graph block) for multiple designs by scaling the stack components. In Fig. 9, we show a two-design example for brevity. Note that a single graph block translates into two different execution performances depending on the design.

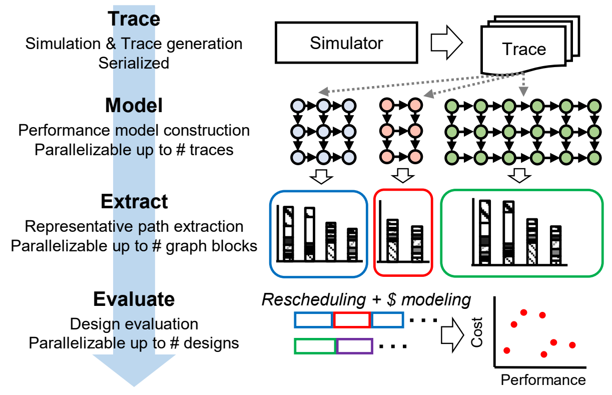
The second step is to re-schedule the graph blocks with the dynamic scheduling reconstruction method. This step is essentially constructing the *multi-core critical execution path* from per-graph block critical paths and the interaction between the blocks (i.e., scheduling constraints). The resulting scheduling scenarios may differ by designs due to different block execution performances (i.e., the critical path may differ by designs). We utilize the memory system model in this step to consider the resource sharing behavior and adjust/finalize the block execution performance.

When the scheduling is done, we can estimate the overall performance of the designs. We emphasize that the graph blocks are large in general (i.e., thousands of instructions) and therefore the scheduling reconstruction incurs negligible overheads. (Refer to Section VI-C for details.)

V. Implementation

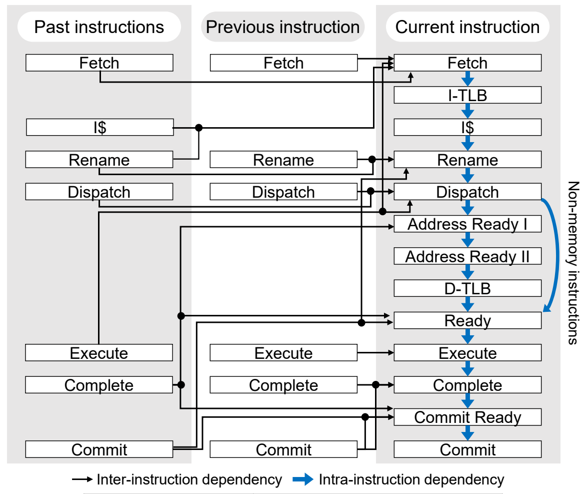
A. Framework Overview

Fig. 10 illustrates the overview of the RpStacks-MT framework. The first step (`Trace`) is to run a cycle-level timing simulation for the current multi-core processor design. The main purpose is to get per-core execution traces, which have a rich set of information regarding the workload execution. Specifically, the trace contains 1) the executed instructions, 2) the timing each instruction went through each of the pipeline



Trace
Simulation & Trace generation
Serialized
Model
Performance model construction
Parallelizable up to # traces
Extract
Representative path extraction
Parallelizable up to # graph blocks
Evaluate
Design evaluation
Parallelizable up to # designs
Simulator
Trace
Rescheduling + $ modeling
Cost
Performance

Fig. 10: Overview of the RpStacks-MT framework



Operation	Dependence edge
Finite fetch bandwidth	I$ → Fetch
Finite fetch buffer size	Rename → Fetch
Control dependency	Instruction → Fetch
Finite ROB	Commit → Rename
Finite rename bandwidth	Rename → Rename
Issue dependency	Execute → Dispatch
Finite dispatch width	Dispatch → Dispatch
Data dependency	Complete → Address Ready 1
Finite physical registers	Commit → Ready
Data dependency	Complete → Ready
Address dependency	Execute → Execute
Cache line sharing	Complete → Complete
Finite commit width	Commit → Commit Ready
uOP dependency	Complete → Commit Ready
Interrupt*	Commit → Fetch
Memory fence*	Commit → Complete

Instruction type (Execute -> Complete)	Classification
ALU	IntAdd, IntDiv, IntMul, Logic
FPALU	FPAdd, FPMul
Control	Branch
Memory*	L1, Remote-L1, L2, Mem (with split bus, port/queue latency)

Fig. 11: Pipeline modeling details of RpStacks-MT's graph model. The Execute→Complete dependence edge is further classified by the instruction type. Newly introduced or augmented components are denoted with an asterisk (*).

stages, 3) the dependencies between the instructions, 4) the performance events each instruction experiences (e.g., cache hit/miss, interrupt, etc), 5) the execution contexts (e.g., thread IDs, synchronization and scheduling events), and 6) the reuse distances (only for memory instructions).

The second step (`Model`) is to construct the multi-core graph-based performance model from the traces. As discussed in Section IV-A, we create a graph block per code snippet.

The third step (`Extract`) is to extract the representative execution paths for each graph block. We adopt the method from [24] to find the paths. As a result, for each block, we get a handful of cycle stacks showing the performances and bottlenecks of potential performance-determining critical paths.

The fourth step (`Evaluate`) is to estimate a new designs' performance. As explained in Section IV-D, we 1) estimate the performance of each graph block by applying the new design specifications (i.e., scaling the cycle stacks accordingly) and 2) perform rescheduling and memory system behavior estimation to derive the overall design performance. Note that we can incorporate other factors such as optimization costs [24] in this step to investigate design tradeoffs.

By repeating the fourth step which has negligible overheads, we explore the design space with high throughput and identify optimal candidates.

B. Achieving High Throughput and Low Latency

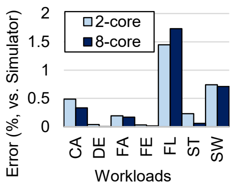
High throughput. To achieve high evaluation throughput, we put huge optimization efforts to the last step (`Evaluate`) which has the overhead growing with the number of designs. We find out that reading graph block performance results (i.e., `Extract` step's results) accounts for the majority of the overhead with a large number of I/Os, because the results are stored as files. We therefore use ramdisk and asynchronous I/O to minimize the overheads.

Low latency. In some development cases, the *absolute design evaluation speed* is an important factor (e.g., exploration results must be available to decide future steps). Fortunately, RpStacks-MT supports parallel analysis to allow the users to tradeoff the evaluation throughput and latency.

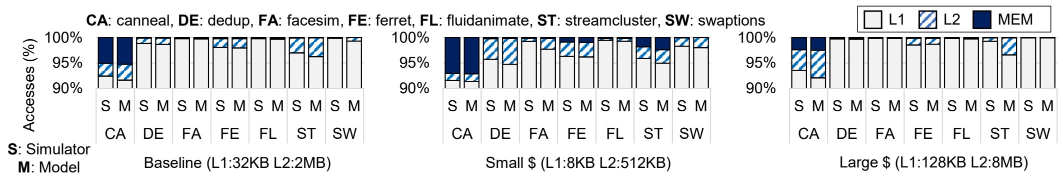
Fig. 10 shows which of the steps can be parallelized. Basically, we allow parallel analysis for *every independent piece of data* (i.e., trace, graph block, or target design). Since we have a large number of traces/blocks/designs, we successfully achieve high speedup *even beyond* the number of target design's core count (which is the typical bound for parallel simulators). To maximize the parallelism, we apply *chunking* to large graph blocks as well. Specifically, we break a large block down into smaller 5K-instruction blocks. Previous work [24] shows that such a segmentation does not degrade the accuracy of the model. Section VI-E shows our parallelization speedup results.

C. Improving the Pipeline Details of the Graph-based Model

To achieve simulation-level performance evaluation accuracy, it is crucial to accurately depict the processor pipeline with a graph-based model. Fig. 11 illustrates the details of our model. It shows the pipeline stages (i.e., vertical nodes of an



(a) Graph model



(b) Memory system model

Fig. 12: Validating the accuracy of the individual models.

instruction) as well as the microarchitectural operations (i.e., dependence edges between the pipeline stages).

We identify the weaknesses of the state of the art models and augment the details in two ways to accurately describe multi-core designs. First, we add new operation types (e.g., interrupt, memory fence) which have a significant performance impact on multi-cores. Second, we further classify the memory instruction's operation latency by modeling split bus and port/queue latency. Our interconnect model (Section IV-B) adjusts this to accurately account for shared resource contentions.

VI. Evaluation

A. Experimental setup

We use MARSSx86 [28] cycle-level simulator as the reference for accuracy and throughput. The `Trace` step (Section V-A) is implemented on this simulator as well (i.e., we get instruction traces, performance events, reuse distance from memory accesses, scheduling events from `pthread` and `__switch_to()` kernel function, etc). Table I shows the parameters of our baseline multi-core processor design.

For the workload, we use seven applications from PARSEC 2.1 benchmark suite [27]. We select the applications with the linkage distance larger than 0.5 based on a similarity analysis [48]. The selection (Table II) covers various synchronization events, parallelization models, and shared resource sensitivities.

B. Accuracy Validation

Multi-core graph model. We first validate the accuracy of our graph-based multi-core performance model, because it should be accurate for the following design performance estimations to be accurate. Specifically, we compare the critical-path length (i.e., performance) of the baseline design from our model against the reference performance from the cycle-level simulator. The results (Fig. 12a) show less than 1.7% errors, confirming the high accuracy of our graph model.

Memory system model. Next, we show the accuracy of our reuse distance-based memory system model. For various designs, we compare the (system-wide) access counts from our model to that from the cycle-level simulator. Fig. 12b shows that our model precisely estimates the accesses to both the private (L1) and the shared (L2, MEM) resources to deliver accurate performance implications.

Design evaluation. Finally, we check whether RpStacks-MT accurately estimates the performance impact of processor design changes (i.e., evaluate new designs). To make a change, we 1) randomly select 1∼3 parameters (denoted with * in Table I) and 2) reduce the values to half or double the values (25% or 4× for the cache capacities). For each application, we perform 500 different changes to evaluate a wide range of designs. To highlight the importance of each model, we compare the results from 1) the naively expanded model (Section III-A), 2) our multi-core graph model + scheduling reconstruction method, 3) our multi-core graph model + memory system model, and 4) RpStacks-MT.

Fig. 13 shows the error distributions of 500 design evaluations with box/violin plots. The boxes and whiskers represent the quartiles, and the violins show the density of the samples (i.e., fatter regions have more samples). First, the naive method (N) shows very large errors because they do not model any critical multi-core performance behaviors. Adding the scheduling method (S) slightly reduces the average error for scheduling-sensitive applications (i.e., `ferret`, `dedup`)^3, but the errors are still too large for accurate performance estimation. The same applies to the memory system model

TABLE I: Baseline multi-core processor design parameters

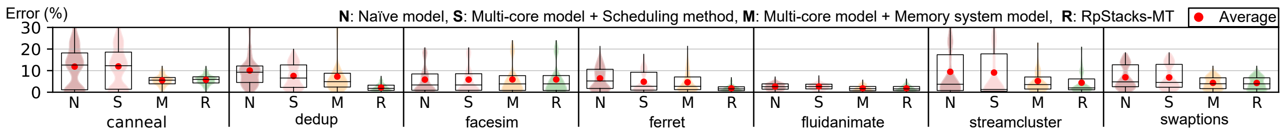
Design parameter	Value
Number of cores	2 or 8
Pipeline Width	4
ROB size	128
LSQ size	48
IQ size	64
*L1 cache (Per-core)	32KB L1D, 32KB L1I, Latency: 4 cycles
*L2 cache (Shared)	2MB, Latency: 15 cycles
*Memory	133 cycles
*Integer Multiplication	8 cycles
*Integer Division	32 cycles
*Float addition/subtract	6 cycles
*Float multiplication	6 cycles
*Integer vector unit	12 cycles
*Float vector unit	12 cycles

*: Target parameters to optimize

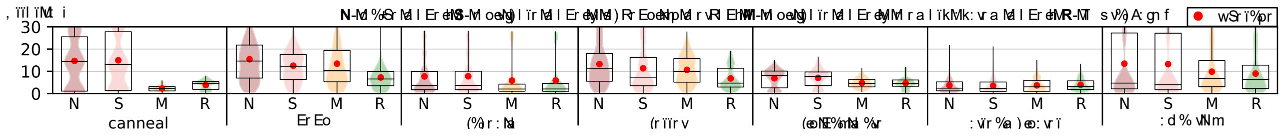
TABLE II: Summary of the PARSEC applications

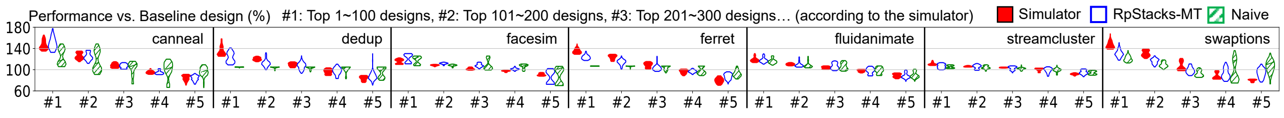
Workload	Parallelization model	*Synchronization methods	Shared resource sensitivity
canneal	Unstructured	L, B	High
dedup	Pipeline	L, C	Low
facesim	Data-parallel	L, C	Low-Medium
ferret	Pipeline	L, C	Medium
fluidanimate	Data-parallel	L, B	Low
streamcluster	Data-parallel	L, B, C	Low-Medium
swaptions	Data-parallel	L	Medium-High

*L: Lock, B: Barrier, C: Condition variable

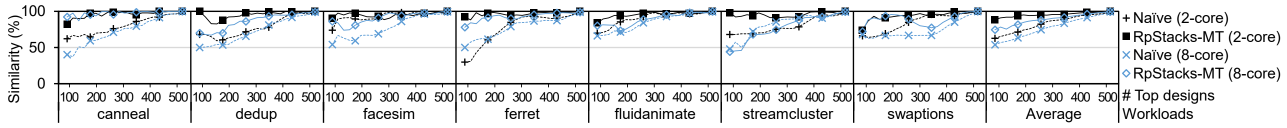


(a) Performance estimation errors: 2-core target processor designs.


(b) Performance estimation errors: 8-core target processor designs.



(c) Performance spectrum of the 500 8-core designs. The designs are sorted based on the simulator results. Fatter regions have more designs.



(d) Similarity between RpStacks-MT's design performance ranking and the simulator's ranking (i.e., ground truth).

Fig. 13: Validating the accuracy for design space exploration.

(M) – while it drastically reduces the errors by accurately considering cache design changes, the scheduling-sensitive applications still show large errors. Lastly, RpStacks-MT (R) correctly accounts for both behaviors and minimizes the errors. Compared to the naive model which has 7.2%/10.0% errors for 2-/8-core designs (average of all applications), our method reduces the errors to 3.0%/4.8% (2-/8-core) to facilitate accurate multi-core processor design evaluation.

Next, we present the performance spectrums of the 500 designs we have (Fig. 13c). The designs have higher (>100) or lower (<100) performance compared to the baseline. We note that our design modifications introduce significant performance changes ($\sim$ 50%), and RpStacks-MT better tracks the distributions compared to the naive model. For example, the naive model fails to detect the increasing performance for `ferret` whereas our method accurately follows the trend.

We also validate RpStacks-MT by comparing its design performance ranking with that of the simulator (Fig. 13d). The similarity value of S indicates that the $S\%$ of the top N designs selected by our method and the simulator are common. Compared to the naive method, RpStacks-MT much more robustly identifies the best performing designs, showing the usefulness of the method in realistic design exploration scenarios.

To better reason about the results, we take a deeper look at individual application behaviors. Fig. 14 shows *where* each application spends its execution cycles (aggregate of all eight cores). First, the applications that hugely benefit from the memory model spend a large portion of their time on memory-related events (e.g., `canneal`). Second, the pipelined applications have large wait cycles, emphasizing the need for accurate dynamic scheduling reconstruction. On the other hand, the data-parallel applications have small wait cycles, indicating that there is only a small room for dynamic behaviors. `streamcluster` is an interesting example; it has large wait cycles but the scheduling pattern is relatively deterministic, also leaving a small room for the dynamics.

C. Overhead Analysis

Before presenting the actual throughput results, we first discuss the design evaluation overhead of RpStacks-MT to illustrate *how* it achieves the high throughput. First, we analyze the overhead of RpStacks-MT's four steps (Fig. 15) – `Trace`,

[3]`ferret` and `dedup` are software-pipelined applications, which show more dynamic and complex scheduling and synchronization behaviors. They have tight producer-consumer communications between the pipeline stages, where the heterogeneity of the stages' characteristics (i.e., instruction mix) and the corresponding speed mismatch incur frequent stalls. As the producer and consumer have vastly different characteristics (e.g., instruction mix, memory intensity), the corresponding mismatch in the speed generates frequent stalls waiting for the data. Furthermore, such producer-consumer dependencies often span over multiple threads to build up long dependency chains.

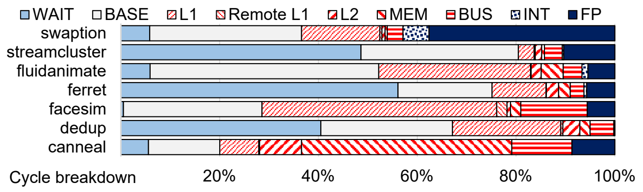


Fig. 14: Cycle breakdown for the 8-core baseline design

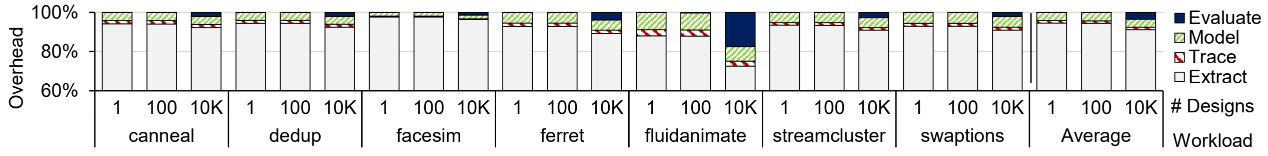


Fig. 15: Latency decomposition of evaluating 1∼10,000 designs (single-threaded RpStacks-MT, 8-core target design).

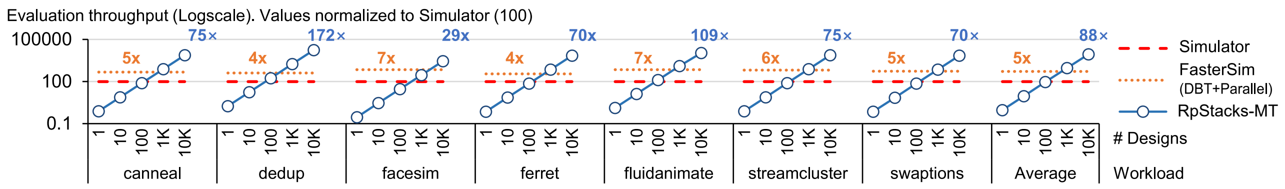


Fig. 16: Comparing the design evaluation throughput (single-threaded RpStacks-MT and simulators, 8-core target design).

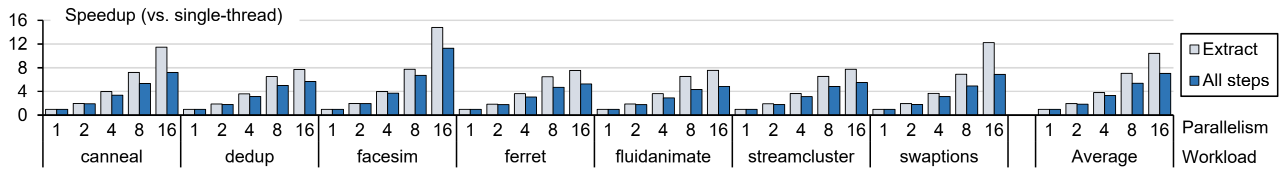


Fig. 17: Speedup from parallelizing RpStacks-MT (10,000 design points, 8-core target design)

`Model`, `Extract`, and `Evaluate`. Note that the overheads are based on the *single-threaded* execution; we discuss the parallelization speedup in Section VI-E.

The overheads of `Trace` and `Model` steps are proportional to the simulation's length. Note that *all* the other simulations and performance models pay the same overhead because a single simulation is always necessary at least for the initial model construction; for further design performance estimations, RpStacks-MT and analytic models might not require a full simulation.

Next, `Extract` accounts for the majority of the overhead, as it is an expensive graph traversal combined with compute-intensive path similarity comparisons. However, this stage is also highly parallelizable and the users may choose to speed up the analysis by using more machine resources (Section VI-E).

The last step's (`Evaluate`) overhead is in fact proportional to the number of designs as we perform scheduling reconstruction and memory system modeling for each design. Nonetheless, the reconstruction overhead is *negligible* compared to any kind of simulations; it consists of finding the graph blocks following simple rules (Section IV-C), and the number of scheduling operations is relatively small thanks to the large block sizes (usually thousands of instructions). Therefore, RpStacks-MT successfully evaluates a large number of designs while maintaining small overheads.

D. Throughput Evaluation

We now discuss the design evaluation throughput of RpStacks-MT (Fig. 16). We provide the throughputs of the baseline simulator and an accelerated simulator (i.e., dynamic binary translation and parallelization, Section II-A2) for comparisons[4]. Since the simulation and RpStacks-MT speeds were stable over different designs, we measured the speeds for 500 designs and projected the throughput for 1K 10K designs. The numbers ($x\times$) show the throughput compared to the baseline simulator assuming 10K design evaluations.

The simulators have fixed throughput regardless of the design space size (i.e., each additional evaluation requires another simulation). RpStacks-MT shows low throughput for a small number of designs because it has high initial model construction and analysis overheads. However, as the design space grows, RpStacks-MT shows much larger throughput because it has a (almost) fixed overhead independent of the number of designs.

On average, RpStacks-MT's throughput exceeds that of the simulators if more than 100∼500 designs are evaluated. For a realistically large number of designs (10,000), RpStacks-MT shows 88× and 18× higher throughput compared to the baseline and the accelerated simulators, respectively. In other words, RpStacks-MT can reduce the design space exploration costs by one to two orders of magnitude.

E. Parallelization Speedups

Lastly, we show how RpStacks-MT can use more simulation resources to accelerate the analysis (Fig. 17). This parallelization is useful when both the design evaluation throughput and *latency* are important.

We provide two speedup results; the speedup of the most time-consuming step (`Extract`) and the whole process.

[4]The baseline cycle-level simulation takes tens of minutes. RpStacks-MT has initial analysis overhead of ten to twenty hours. After the analysis, a single design exploration takes few seconds (i.e., we can explore practically infinite number of designs with negligible overheads after the initial analysis).

`Extract` step scales quite well as we work on many *independent* graph blocks in parallel. The whole framework speedup is slightly less than ideal due to the non-parallelizable steps (e.g., `Trace`). We emphasize that the scalability is still high because the serialized steps account for a relatively small portion in the total overheads (Fig. 15).

The results show that RpStacks-MT allows an easy tradeoff between the design evaluation throughput and latency to meet various demands in a processor development process.

VII. Related Work

A. Models for uArch Analysis & Performance Estimation

Mechanistic models. Mechanistic models [16]–[18] use microarchitectural insights to construct simple pipeline models and efficiently analyze the performance. Interval analysis and its successors [16], [49], [50] are one of the most representative examples. Many other studies improve the mechanical models by combining empirical approaches [51].

Graph-based models. Graph-based models interpret processor components as nodes and operations as edges to represent the behavior. Their rich set of information helps to analyze the current performance bottlenecks as well as predict the performance upon design changes. [20]–[23], [25], [26] find the critical path and slacks (i.e., non-critical operations) from graph models to optimize processor designs. [52] expands the model to a full-system to identify key bottlenecks. [26], [33] change the edge properties (i.e., operation costs) to understand how the instructions interact to form the overall performance. [34] further improves the accuracy of graph models by better modeling store queue and branch misprediction penalty.

Empirical models. The empirical performance models estimate a processor's behavior using machine learning techniques rather than reasoning about the microarchitectural operations. Popular concepts such as linear regression [53] and non-linear spline-based model [54]–[57] have been utilized. [58] suggests a ranking model to perform relative comparisons instead of estimating the absolute performance.

Performance monitoring units. Performance monitoring units (PMUs) notify microarchitectural events with low overhead and facilitate various real-time performance analysis and improvement methods. Initial work [36], [37] simply counts the number of miss events to analyze the bottleneck events. A later proposal [59] uses paired sampling to investigate adjacent instructions as a whole; it provides better results by considering the overlaps between instructions. Frontend Miss Table [35] is a PMU-based mechanistic model to understand overlapped bottleneck in out-of-order core, and [19] further improves it to consider instruction's critical path. [60] identifies the performance criticality of the threads in a multi-core processor to implement efficient acceleration via DVFS.

B. Optimizing Processor Design Evaluation

Trace-driven simulators. Trace-driven simulators are similar to graph-based performance analyses in that they also analyze rich input information (i.e., trace) to quickly derive the performance. Previous approaches [10], [61] successfully reconstruct the performance of multi-threaded applications with low overhead, but are limited to specific parallelization models (e.g., task-based workloads) and share the same limitations as (fast) simulators.

Sampling and simulation load reduction. Sampling is by far the most effective and popular way of accelerating a simulation. A large volume of works [4]–[11], [62] suggest reducing the analysis load to improve the evaluation efficiency. The key ideas of these techniques (e.g., sampling, eliminating duplicates) are *orthogonal* to our method and therefore can be applied in combination to further improve the efficiency of design evaluation.

VIII. Discussion: Improving RpStacks-MT

We now discuss ideas to improve RpStacks-MT's capability.

Modeling unit-level utilization for power and reliability estimation. Unit-level utilization of microarchitectural components (e.g., ALU units) allows us to explore the power and reliability of processors in addition to the performance. Since a unit's activity is the deterministic characteristics of a workload, we can identify the activation counts of the units from workload execution traces. RpStacks-MT then reconstructs the number of cycles it takes to execute a trace (i.e., performance) and estimates the unit activations per cycle (i.e., unit utilization). We may feed this information to other modeling tools [63]–[65] to get the power and reliability numbers of a processor design.

Modeling heterogeneous cores. To model heterogeneous core processors, we need to construct a graph model per each core type. A code snippet now translates into multiple types of graph blocks according to the core types. The scheduler should put the graph blocks matching the core types when constructing the performance behaviors.

Exploiting redundancies to increase the throughput. We can further boost the throughput of RpStacks-MT by exploiting the redundancies in the traces. First, RpStacks-MT is well suited for sampling techniques. The previous graph-based model [24] also utilizes SimPoints. Second, the loops inside graph blocks can be exploited. For example, [11] suggests analyzing only the unique basic blocks to reduce the simulation load; we may adopt similar ideas to accelerate RpStacks-MT.

IX. Conclusion

In this paper, we presented RpStacks-MT, which evaluates multi-core processor designs with high throughput to successfully obtain optimal candidates. The key idea is to use a graph-based multi-core performance model to quickly cover multiple designs, while maintaining good accuracy with reuse distance-based memory system model and dynamic scheduling reconstruction method. Our evaluation using PARSEC workload suite showed that RpStacks-MT achieves 88× higher design evaluation throughput compared to a conventional multi-core simulator (18× vs. an accelerated simulator) for evaluating 10,000 designs, while maintaining low errors.

Acknowledgment

This work was partly supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (MSIT) (NRF-2015M3C4A7065647, NRF-2017R1A2B3011038), Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. R0190-15-2012, No. 1711073912), and Creative Pioneering Researchers Program through Seoul National University. We also appreciate the support from Automation and Systems Research Institute (ASRI), Inter-university Semiconductor Research Center (ISRC), and Neural Processing Research Center (NPRC) at Seoul National University.

References

[1] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[2] T. E. Carlson, W. Heirmant, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

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

[4] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[5] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003.

[6] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sampled simulation of multi-threaded applications," in *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, 2013.

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

[8] E. K. Ardestani and J. Renau, "Esesc: A fast multicore simulator using time-based sampling," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, 2013.

[9] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega, "Cotson: infrastructure for full system simulation," *SIGOPS Oper. Syst. Rev.*, vol. 43, 2009.

[10] T. Grass, A. Rico, M. Casas, M. Moreto, and E. Ayguad, "Taskpoint: Sampled simulation of task-based programs," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016.

[11] J. Lee, H. Jang, J. e. Jo, G. h. Lee, and J. Kim, "Stressright: Finding the right stress for accurate in-development system evaluation," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2017.

[12] E. S. Chung, M. K. Papamichael, E. Nurvitadhi, J. C. Hoe, K. Mai, and B. Falsafi, "Protoflex: Towards scalable, full-system multiprocessor simulations using fpgas," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 2, 2009.

[13] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat, "Fpga-accelerated simulation technologies (fast): Fast, full-system, cycle-accurate simulators," in *Proceedings of the 40th Annual IEEE/ACM international Symposium on Microarchitecture*, 2007.

[14] M. Pellauer, M. Adler, M. Kinsy, A. Parashar, and J. Emer, "Hasim: Fpga-based high-detail multicore simulation using time-division multiplexing," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, 2011.

[15] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, and K. Asanovic, "Ramp gold: an fpga-based architecture simulator for multiprocessors," in *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, 2010.

[16] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A mechanistic performance model for superscalar out-of-order processors," *ACM Trans. Comput. Syst.*, 2009.

[17] P. G. Emma, "Understanding some simple processor-performance limits," *IBM J. Res. Dev.*, vol. 41, 1997.

[18] T. S. Karkhanis and J. E. Smith, "A first-order superscalar processor model," in *Proceedings of the 31st annual international symposium on Computer architecture*, 2004.

[19] W. Heirman, T. E. Carlson, S. Che, K. Skadron, and L. Eeckhout, "Using cycle stacks to understand scaling bottlenecks in multi-threaded workloads," in *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, 2011.

[20] B. Fields, S. Rubin, and R. Bodík, "Focusing processor policies via critical-path prediction," in *Proceedings of the 28th annual international symposium on Computer architecture*, 2001.

[21] B. Fields, R. Bodík, and M. D. Hill, "Slack: maximizing performance under technological constraints," in *Proceedings of the 29th annual international symposium on Computer architecture*, 2002.

[22] M. Agarwal, N. Navale, K. Malik, and M. I. Frank, "Fetch-criticality reduction through control independence," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, 2008.

[23] S. Subramaniam, A. Bracy, H. Wang, and G. H. Loh, "Criticality-based optimizations for efficient load processing," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, 2009.

[24] J. Lee, H. Jang, and J. Kim, "Rpstacks: Fast and accurate processor design space exploration using representative stall-event stacks," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[25] E. Tune, D. Liang, D. Tullsen, and B. Calder, "Dynamic prediction of critical path instructions," in *The Seventh International Symposium on High-Performance Computer Architecture*, 2001.

[26] E. Tune, D. M. Tullsen, and B. Calder, "Quantifying instruction criticality," in *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, 2002.

[27] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008.

[28] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSSx86: A Full System Simulator for x86 CPUs," in *Design Automation Conference 2011 (DAC'11)*, 2011.

[29] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, "Simflex: Statistical sampling of computer system simulation," *IEEE Micro*, vol. 26, 2006.

[30] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, 2011.

[31] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2sim: a simulation framework for cpu-gpu computing," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, 2012.

[32] 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 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[33] B. A. Fields, R. Bodik, M. D. Hill, and C. J. Newburn, "Interaction cost and shotgun profiling," *ACM Trans. Archit. Code Optim.*, vol. 1, 2004.

[34] T. Tanimoto, T. Ono, K. Inoue, and H. Sasaki, "Enhanced dependence graph model for critical path analysis on modern out-of-order processors," *IEEE Computer Architecture Letters*, vol. 16, 2017.

[35] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A performance counter architecture for computing accurate cpi components," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, 2006.

[36] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl, "Continuous profiling: where have all the cycles gone?" *ACM Transactions on Computer Systems*, vol. 15, 1997.

[37] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz, "Performance analysis using the mips r10000 performance counters," in *Supercomputing '96:Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, 1996.
[38] A. W. Services, "Amazon EC2 Pricing," accessed 2018-07-04. [Online]. Available: https://aws.amazon.com/ec2/pricing/on-demand/
[39] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, 2003.
[40] M. D. Hill and A. J. Smith, "Evaluating associativity in cpu caches," *IEEE Transactions on Computers*, vol. 38, 1989.
[41] X. Xiang, B. Bao, C. Ding, and Y. Gao, "Linear-time modeling of program working set in shared cache," in *2011 International Conference on Parallel Architectures and Compilation Techniques*, 2011.
[42] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *11th International Symposium on High-Performance Computer Architecture*, 2005.
[43] A. K. Erlang, "The theory of probabilities and telephone conversations," *Nyt Tidsskrift for Matematik B*, vol. 20, 1909.
[44] J. C. Huang, J. H. Lee, H. Kim, and H. H. S. Lee, "Gpumech: Gpu performance modeling technique based on interval analysis," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.
[45] Y. Zhang, W. Zheng, X. Dong, and S. Gan, "A performance analytical approach based on queuing model for network-on-chip," in *2010 3rd International Symposium on Parallel Architectures, Algorithms and Programming*, 2010.
[46] A. E. Kiasari, Z. Lu, and A. Jantsch, "An analytical latency model for networks-on-chip," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, 2013.
[47] E. Fischer and G. P. Fettweis, "An accurate and scalable analytic model for round-robin arbitration in network-on-chip," in *2013 Seventh IEEE/ACM International Symposium on Networks-on-Chip (NoCS)*, 2013.
[48] C. Bienia, S. Kumar, and K. Li, "Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, 2008.
[49] D. Genbrugge, S. Eyerman, and L. Eeckhout, "Interval simulation: Raising the level of abstraction in architectural simulation," in *2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, 2010.
[50] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, 2014.
[51] A. Hartstein and T. R. Puzak, "The optimum pipeline depth for a microprocessor," in *Proceedings of the 29th annual international symposium on Computer architecture*, 2002.
[52] A. G. Saidi, N. L. Binkert, S. K. Reinhardt, and T. Mudge, "Full-system critical path analysis," in *ISPASS 2008 - IEEE International Symposium on Performance Analysis of Systems and software*, 2008.
[53] P. J. Joseph, K. Vaswani, and M. J. Thazhuthaveetil, "Construction and use of linear regression models for processor performance analysis," in *the 12th International Symposium on High-Performance Computer Architecture*, 2006.
[54] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, 2006.
[55] B. C. Lee and D. Brooks, "Efficiency trends and limits from comprehensive microarchitectural adaptivity," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, 2008.
[56] B. C. 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*, 2008.
[57] B. C. Lee and D. M. Brooks, "Illustrative design space studies with microarchitectural regression models," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007.
[58] T. Chen, Q. Guo, K. Tang, O. Temam, Z. Xu, Z.-H. Zhou, and Y. Chen, "Archranker: A ranking approach to design space exploration," in *Proceedings of the 41st annual international symposium on Computer architecture*, 2014.
[59] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos, "Profileme: hardware support for instruction-level profiling on out-of-order processors," in *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, 1997.
[60] 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 40th Annual International Symposium on Computer Architecture*, 2013.
[61] A. Rico, A. Duran, F. Cabarcas, Y. Etsion, A. Ramirez, and M. Valero, "Trace-driven simulation of multithreaded applications," in *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, 2011.
[62] A. Sandberg, N. Nikoleris, T. E. Carlson, E. Hagersten, S. Kaxiras, and D. Black-Schaffer, "Full speed ahead: Detailed architectural simulation at near-native speed," in *2015 IEEE International Symposium on Workload Characterization*, 2015.
[63] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.
[64] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan, "Temperature-aware microarchitecture," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003.
[65] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.