To appear in the 47th IEEE/ACM International Symposium on Microarchitecture

RpStacks: Fast and Accurate Processor Design Space Exploration Using Representative Stall-Event Stacks

Jaewon Lee*, Hanhwi Jang*, and Jangwoo Kim
Department of Computer Science and Engineering
POSTECH
{*spiegel0, hanhwi, jangwoo*}*@postech.ac.kr*

***Abstract*—CPU architects perform a series of slow timing simulations to explore large processor design space. To minimize the exploration overhead, architects make their best efforts to accelerate each simulation step as well as reduce the number of simulations by predicting the exact performance of designs. However, the existing methods are either too slow to overcome the large number of design points, or inaccurate to safely substitute extra simulation steps with performance predictions.**

In this paper, we propose RpStacks, a fast and accurate processor design space exploration method to 1) identify the current design point's key performance bottlenecks and 2) estimate the exact impacts of latency adjustments without launching an extra step of simulations. The key idea is to selectively collect the information about performance-critical events from a single simulation, construct a small number of event stacks describing the latency of distinctive execution paths, and estimate the overall performance as well as stall-event composition using the stacks. Our proposed method significantly outperforms the existing design space exploration methods in terms of both the latency and the accuracy. For investigating 1,000 design points, RpStacks achieves 26 times speedup on average over a variety of applications while showing high accuracy, when compared to a popular x86 timing simulator.

***Keywords*-Design space exploration; Performance analysis; Simulation;**

I. Introduction

Designing a high-performance out-of-order superscalar processor is an extremely challenging task because many performance-critical events interact and determine the overall performance. To correctly account all such events, architects explore the huge design space by launching a wide spectrum of timing-accurate, but slow simulations over and over. During this process, architects identify key performance bottlenecks from the previous-step simulation results, determine how to adjust the design of corresponding modules, and launch the next-step simulation expecting the new results to match the desired performance.

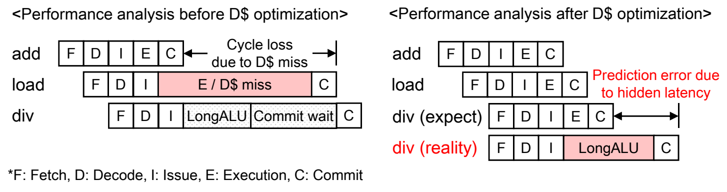
However, the simulation speed has become one of the most critical bottlenecks in developing a high-performance CPU these days. CPU architects in industry put more emphasis on simulation accuracy than simulation latency to avoid fabricating an incorrect product. The corresponding increase in simulation latency can delay the development cycle prohibitively or even lead to a development failure. For example, even a single step of simulations can now take up to days and weeks due to the increasing complexity of the simulation models.

To overcome the simulation latency overhead, architects must reduce the latency of a single simulation, the number of simulations at each step, and the number of simulation steps. First, to reduce the latency of a single simulation, architects propose to simulate only a small set of representative workload regions [1], [2], use hardware support [3]–[5], parallelize a single simulation to launch multiple threads and processes [6], and increase the level of abstraction by introducing analytic models [7]. Even though such methods can significantly reduce the latency of a single simulation, they do not reduce the simulation space nor the number of simulation steps. As a result, the large number of simulations eventually overwhelm their speedup benefits.

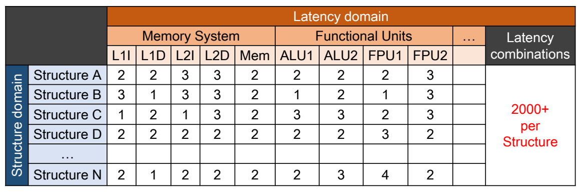
On the other hand, advanced simulation-based performance modeling methods such as pipeline-stall analysis [8], [9] and critical-path analysis [10]–[12] aim to provide more intuition from a single simulation. These methods first simulate the current design point to collect the information about various performance-critical events (e.g., pipeline stalls, branch mispredictions, cache misses). Next, they explain the overall performance with specific pipeline-stall events identified by either an analytic pipeline model (i.e., interval analysis) or the longest path of dependent instructions (i.e., critical path). Although these methods potentially reduce the design space by providing more insights, they fail to accurately predict the performance upon design changes due to the lack of the information about currently unexposed performance-critical events. Consequently, they still require a large number of simulations and thus fail to reduce the simulation latency overhead.

In this paper, we propose RpStacks, a *fast and accurate* processor design space exploration method to 1) identify the current design point's key performance bottlenecks and 2) accurately predict the performance impacts of adjusting the bottleneck events' latencies without an extra simulation step. The key idea is to identify a small number of *representative*

*These authors contributed equally to this work.


<Performance analysis before D$ optimization>
add
F D I E C
Cycle loss due to D$ miss
load
F D I E / D$ miss C
div
F D I LongALU Commit wait C
<Performance analysis after D$ optimization>
add
F D I E C
load
F D I E C
Prediction error due to hidden latency
div (expect)
F D I E C
div (reality)
F D I LongALU C
*F: Fetch, D: Decode, I: Issue, E: Execution, C: Commit

(a) Complexity: penalties hidden in an out-of-order superscalar CPU



		Latency domain										
		Memory System					Functional Units				…	Latency combinations
		L1I	L1D	L2I	L2D	Mem	ALU1	ALU2	FPU1	FPU2		
Structure domain	Structure A	2	2	3	3	2	2	2	2	3		2000+ per Structure
	Structure B	3	1	3	3	2	1	2	1	3		
	Structure C	1	2	1	3	2	3	3	2	3		
	Structure D	2	2	2	2	2	2	2	3	2		
	…											
	Structure N	2	1	2	2	2	2	3	4	2		

(b) Diversity: the number of design points to explore

Figure 1: Challenges of processor design exploration

execution paths and construct a *Representative stall-event Stack (RpStack)* for each path to reveal the performance penalties. The penalty composition of RpStacks changes depending on the bottleneck event latencies of a design. Architects can therefore identify the key performance bottlenecks as well as accurate performance of many design points by simply examining the stacks.

The fundamental challenge in building RpStacks is to efficiently collect the representative paths and their stall events during a single simulation. For example, naively collecting all path information would increase the amount of data exponentially as the number of instructions grows. RpStacks therefore avoids such overhead using the following three intuitions. First, multiple execution paths share major performance-critical events, suggesting to group the similar paths. Second, the number of distinctive path-groups will be minimal due to a program's repeated popular event sequences. Third, the paths with no potential performance-critical events (i.e., too short paths or the paths dominated by the other paths) can be safely discarded.

Following the intuitions, RpStacks successfully extracts a set of performance-critical event stacks and allows the fast and accurate exploration over wide processor design space. Our implementation on top of MARSSx86 simulator [13] for SPEC CPU 2006 workloads [14] shows that the proposed method significantly outperforms the existing design exploration methods in terms of both latency and accuracy.

To the best of our knowledge, RpStacks is the first approach to accurately identify performance bottlenecks and correctly predict performance changes by analyzing multiple sets of performance-critical events based on a single simulation, whereas other methods focus mainly on the currently exposed stall events. Therefore, our work makes the following contributions:

- **Novel approach.** We propose a *novel, fast,* and *accurate* processor design space exploration method, which identifies the key performance bottlenecks and correct predicts performance using multiple stall-event stacks collected from a *single simulation step*.
- **Accurate prediction.** Our method accurately predicts processor performance by simultaneously considering multiple unique execution paths and their stall-event decompositions.
- **Reduced simulations.** Our method reduces the scope of a single simulation step as well as the number of simulation steps.
- **Real-world applicability.** Our method targets a real-world processor development, where the overhead of design space exploration and result regression is prohibitive.
- **Strong results.** Our implementation significantly outperforms the widely-accepted existing methods in both the prediction accuracy and the simulation latency. For investigating 1,000 design points, our method shows *26 times speedup* on average over a variety of applications, while showing high accuracy as well.

In Section 2, we discuss the difficulties of processor design space exploration and limitations of the existing approaches. In Section 3, we describe the design goals and challenges to motivate our idea, and Section 4 discusses our implementation. Section 5 evaluates the performance and overhead of our method. Section 6 summarizes the related work and Section 7 draws conclusions.

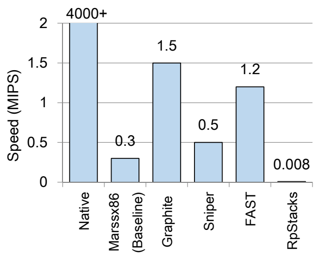
II. Background and Limitations

In this section, we introduce the difficulties of processor design space exploration and the existing approaches addressing the problem. We then discuss their limitations to motivate RpStacks.

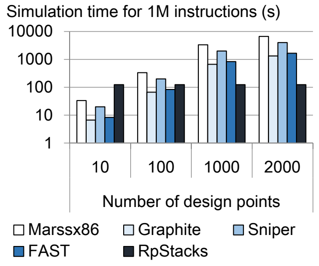
A. Challenges of exploring modern processor designs

The fundamental challenges of exploring processor designs come from the two aspects of modern out-of-order superscalar processors; the *complexity* and the *diversity*.

Complexity. The complex behavior of modern out-of-order processors and the corresponding simulation overhead are the major reasons slowing down design explorations. The three-instruction snippet in Figure 1a illustrates such a complicated behavior of modern processors. Intuitively, optimizing the data cache access in the example on the left should eliminate the cycle loss due to the cache miss. However, as the previously hidden long ALU event now becomes the new bottleneck, the performance improvement is less than expected (as shown in the example on the right.) To correctly address such interactions and accurately estimate the performance, CPU architects often end up relying on timing simulators. As a result, a processor design exploration



(a) Speed comparison of various simulators



(b) Design exploration overhead of various simulators

Figure 2: Addressing the complexity: simulator acceleration

suffers from the long latency of thoroughly simulating complex processor behaviors.

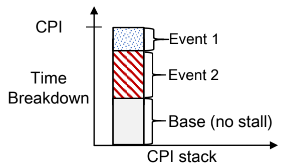
Diversity. A large number of processor design choices combined with slow simulation further exacerbate the design space exploration problem. As shown in Figure 1b, modern processors have a variety of choices in both the *structure (e.g., size, bandwidth, policy)* and *latency (i.e., cycles consumed for an operation)* domain, yielding a considerable number of design points to test. However, as the complexity of processors forces the design evaluation to be simulation-based, exhaustive search to find the best design point becomes infeasible. Architects therefore compromise with a sub-optimal design, even after spending a significant amount of time on simulating and comparing numerous design candidates.

B. Addressing the complexity problem

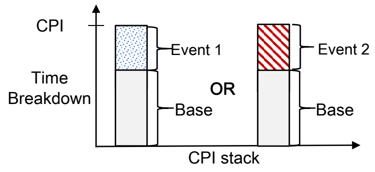
One way to accelerate a processor design exploration is to address the complexity problem and make each simulation step faster. As simulation load reduction methods [1], [2] tradeoff their accuracy with the speedup, we only discuss pure simulation acceleration methods.

Figure 2a shows the simulation speed of native execution, MARSSx86 [13] (baseline timing simulator), Graphite [6] (parallelization and one-IPC model), Sniper [7] (parallelization and abstraction), FAST [3] (hardware acceleration), and RpStacks. We use the best-reported numbers from the literatures to represent the simulation speed of each acceleration method and use the average speedup for our proposed method (refer to Section V-C for details.) As shown in the figure, the acceleration methods gain significant speedup compared to the baseline timing simulator by either sacrificing a small amount of accuracy or using an additional hardware support. RpStacks, in contrast, shows slow simulation speed due to the extra information collection and processing routines added to the baseline simulator.

Limitations. Nonetheless, the high speed of acceleration methods does not necessarily translate into fast design space exploration. As illustrated in Figure 2b, the total design space exploration time of the simulation acceleration methods diverges as the number of designs does. In contrast,



(a) Result of pipeline-stall analysis in CPI stack



(b) Limitation of pipeline-stall analysis: which one to use?

Figure 3: Addressing the diversity: pipeline-stall analysis

RpStacks maintains the same overhead due to its single-simulation nature, and its performance eventually surpasses that of the acceleration methods. This illustrates that although the acceleration methods are useful for evaluating few design points, they lose their effectiveness in case of wide design space exploration, which is an inevitable step in processor development cycle.

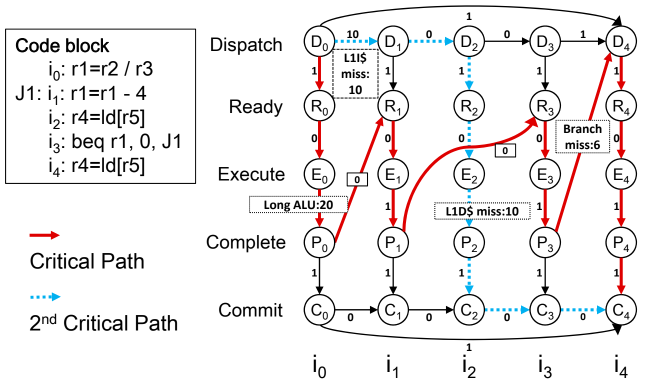
C. Addressing the diversity problem

An alternative way of accelerating the exploration is to reduce the diversity of processor design. Simulation result analysis methods provide architects with additional insights of the current bottleneck events, to reduce the number of design points to explore in the next simulation step. In our study, we discuss the key idea and limitations of two representative methods: pipeline-stall analysis [8], [9] and critical-path analysis [10]–[12].

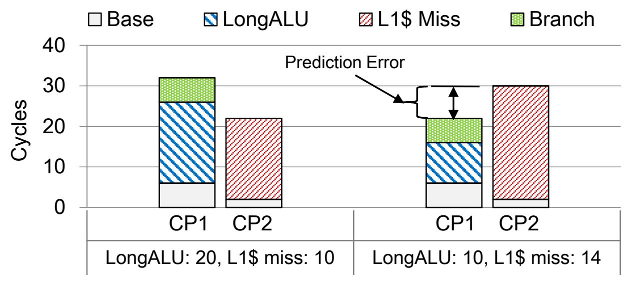
Pipeline-stall analysis. The pipeline-stall analysis first collects the information about various performance stalling events during the region of interest (e.g., data hazards, branch mispredictions, cache misses). Architects can collect such information using either a detailed timing simulator or performance monitoring unit (PMU) available in the real machine, depending on the target accuracy and latency. Once the results become available, they assign the lost cycles to specific pipeline-stall events identified for the moment of cycle losses, using their own analytic pipeline models.

As shown in Figure 3a, the per-bottleneck stall cycles can be expressed as a CPI stack [15], which describes how much each bottleneck event contributes to the overall performance. With CPI stack, architects can identify the performance-critical events of the current design point as well as make predictions on future design points by adjusting the stack.

Limitations of pipeline-stall analysis. The pipeline-stall analysis methods provide neither accurate bottleneck event analysis nor performance predictions due to the following reasons. First, these methods cannot identify the right bottleneck events when multiple events overlap in the pipeline. For example, in a situation similar to the one in Figure 1a, they consider the penalty event to be either one of the two concurrent events as illustrated in Figure 3b. Therefore, the adjustment of the penalties of the two events cannot be accurately modeled using pipeline-stall analysis methods. Moreover, this type of methods cannot handle fine-grain stall



(a) Basic idea of critical-path analysis



(b) Performance misprediction due to hidden execution path

Figure 4: Addressing the diversity: critical path analysis

events because they only log the event when the pipeline is completely stalled. The performance-critical events causing low issue/commit rates (e.g., data dependency) are thus ignored in this method, further decreasing the accuracy of the analysis and prediction.

Critical-path analysis. The critical-path analysis methods [10]–[12] are well-known techniques to identify the longest instruction execution path existing in a dynamic instruction execution trace. Figure 4a illustrates how to identify the critical path existing in a five-instruction example trace. Each instruction is represented as each column advancing from left to right, which consists of five pseudo-pipeline stages: Dispatch (D), Ready for issue (R), Issued for execution (E), Complete the execution (P) and Commit (C). Arrows in the graph indicate all normal pipeline-stage advances (e.g., D→R→E→P→C), and the cycles spent due to specific performance-stalling events (e.g., long-latency operation, L1 cache misses, branch misprediction). Once such a graph becomes available, we can find the critical path by adding up all the marked cycles following all the arrows and finding the longest path from the first instruction to the last instruction.

The critical-path analysis methods can also be used to identify key performance bottlenecks of the current design, and predict the performance impacts of alleviating the bottleneck events. In fact, just as pipeline-stall analysis, a critical path can be directly translated into a CPI stack by adding up all the cycles spent for the same type of event. Compared

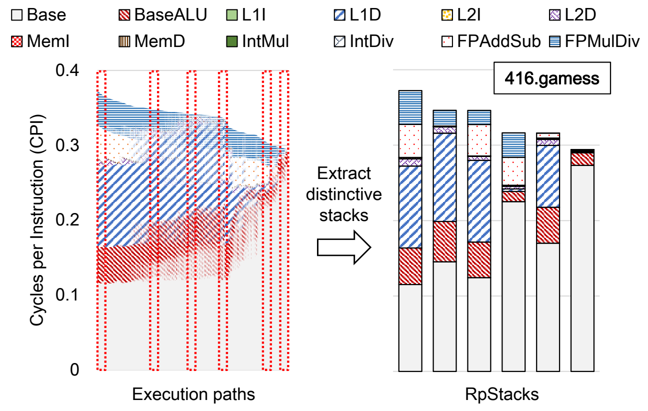


Figure 5: Generating RpStacks from multiple execution paths

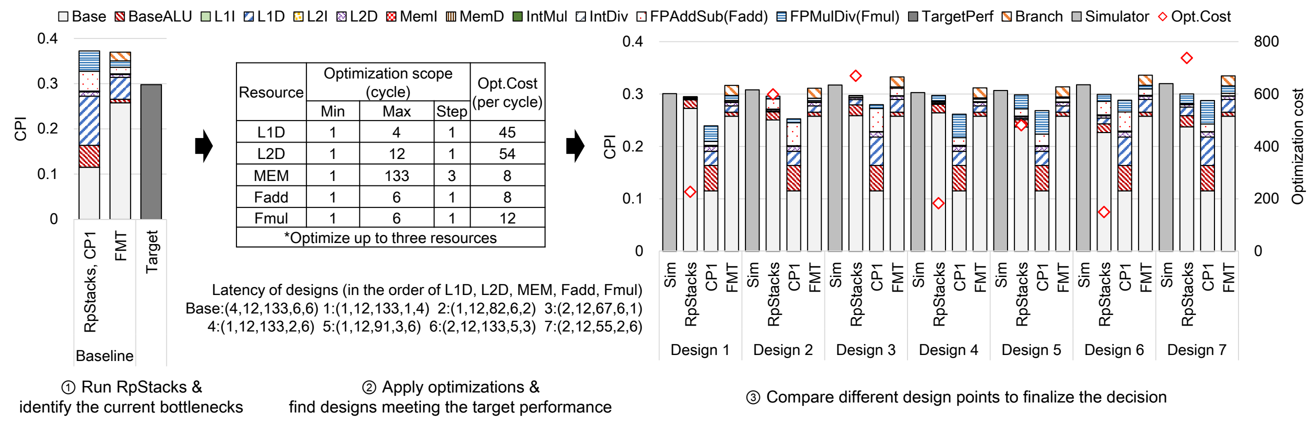
to the pipeline-stall analysis, the critical-path analysis can always assign the lost cycles to the right bottleneck of the right paths and thus shows more accurate analysis result. In addition, as modifying the latency of a specific event adjusts the length of all related paths, the per-path CPI stack can be used to identify key performance bottlenecks for a given path and accurately predict the performance impact even when adjusting multiple latencies together.

Limitations of critical-path analysis. However, the critical-path analysis methods have their own limitations to be used for performance prediction. First, these methods also suffer from inaccurate predictions due to the overlapping penalty events. In such scenarios, the overlapping penalties appear as different paths. The graph in Figure 4a shows that there exists a secondary critical path experiencing two L1 cache miss events and its length is slightly shorter than the main critical path. When such near-critical secondary paths exist, the performance analysis based on the main critical path can incur a significant misprediction due to the possible change of critical path. For example, in Figure 4b, the secondary path becomes the main critical path after applying different amount of event latencies. In such cases, the performance analysis based on the ex-critical path becomes invalid.

In fact, such change of critical path can be tracked by reconstructing the graph and finding the new longest path for every latency reconfiguration [12]. As the graph reconstruction is much faster than timing simulations, this can be an effective approach when the number of design points is small. However, the graph reconstruction method has the same problem as simulation acceleration methods; it has to re-evaluate the graph for each design point. Therefore, the exponentially increasing number of design points will eventually dominate its fixed speedup.

III. Design Goals and Challenges

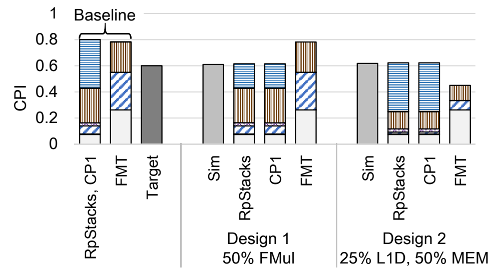
In this section, we describe the key ideas and challenges of RpStacks. We first introduce the intuition which led us to explore multiple sets of performance-critical stall-event



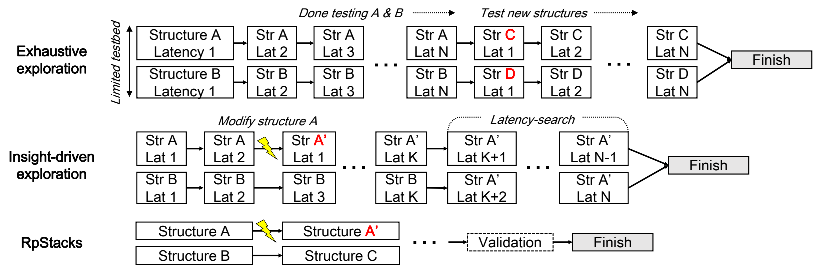
Resource	Optimization scope (cycle)			Opt.Cost (per cycle)
	Min	Max	Step	
L1D	1	4	1	45
L2D	1	12	1	54
MEM	1	133	3	8
Fadd	1	6	1	8
Fmul	1	6	1	12
*Optimize up to three resources				

Latency of designs (in the order of L1D, L2D, MEM, Fadd, Fmul)
Base:(4,12,133,6,6) 1:(1,12,133,1,4) 2:(1,12,82,6,2) 3:(2,12,67,6,1)
4:(1,12,133,2,6) 5:(1,12,91,3,6) 6:(2,12,133,5,3) 7:(2,12,55,2,6)

(a) Using RpStacks to simultaneously compare multiple design points (416.gamess)



(b) Example of FMT showing low accuracy (437.leslie3d)



(c) Design space exploration method comparison

Figure 6: RpStacks: example design space exploration scenarios

stacks, using a real-world application. Next, we illustrate how RpStacks effectively identifies the bottleneck events and predicts the performance with a design space exploration scenario. We finally present the challenges of RpStacks and introduce our solutions to the problems.

A. *The existence of multiple performance-critical paths*

The fundamental idea of RpStacks is to simultaneously investigate stall-event stacks of multiple performance-critical execution paths. The key intuition is that the execution paths share major stall events and therefore can be grouped into a small number of representative paths.

Figure 5 shows the stall-event stacks of the execution paths in `416.gamess` [14] along with the selected RpStacks. We collect the data using the proposed methodology described in Section IV. As illustrated in the figure, the stacks of different execution paths show a similar trend and a small number of distinct stacks can be extracted. The leftmost RpStack shows the penalty decomposition of the current design point, as it has the largest CPI and thus represents the longest execution path. The other stacks can become the longest execution path for different stall-event latency configurations. In other words, by keeping these representative stacks, it is possible to accurately predict the performance even when the event latencies are adjusted in a complex way.

We observe the similar tendencies in other applications of the benchmark suite as well, and use the phenomenon as the key intuition driving our methodology.

B. *Utilizing RpStacks for fast and accurate design space exploration*

We next illustrate how RpStacks efficiently explores a processor design space using real applications (`416.gamess` and `437.leslie3d`) and our baseline architecture (Table II). The performance predictions of popular simulation result analysis methods - *single critical path analysis (CP1)* and a *pipeline-stall based analysis (Frontend Miss Table, FMT)* [8] - are also provided to emphasize the importance of accurate prediction.

Figure 6a shows the exploration process of RpStacks; a single structure/application example is used for brevity. First, architects would run RpStacks with the baseline latency to identify the major bottlenecks of the current design point's execution path as well as the other hidden execution paths. In the example, they would notice three major bottlenecks - L1D cache access (L1D), FP addition/subtraction operation (Fadd), and FP multiplication/division operation (Fmul). FMT shows different event decompositions due to its unique penalty accounting method.

Next, the architects would apply various latency optimizations focusing on the bottlenecks of the current design, to reach the target performance. In this step, RpStacks can incorporate any useful factor that is related to the event latency (e.g., optimization cost for reducing latency) without extra overhead. In our example, RpStacks quickly explores

2500+ latency configurations and obtains more than 200 different points meeting the design goal (i.e., target CPI). The exploration incurs negligible costs because our method simply finds the longest path among a small number of representative execution paths (collected in the first step) to estimate the performance. If traditional simulation methods were used, this step would take significantly longer time while covering much less design points.

As the final step, the architects would compare the selected designs to finalize the decision. As different designs yield different optimization costs as well as performance characteristics, they can choose points which are optimal for multiple workloads while considering the optimization budget.

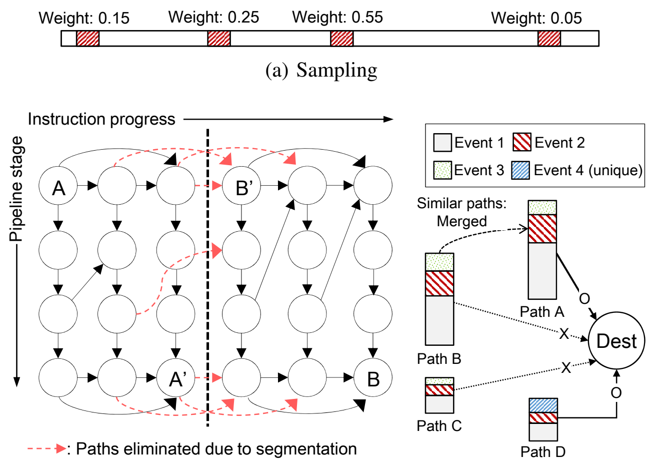
Accuracy validation. For RpStacks (or other methods) to be useful, its performance prediction should be accurate over multiple optimization scenarios and workloads. The example in Figure 6a illustrates that RpStacks generates highly accurate results, while CP1 fails to correctly predict the performance. This is mainly due to its inability to account the overlapping of events; in the example, CP1 fails to catch the base penalty hidden behind the current bottleneck events. On the other hand, FMT produces quite accurate results for this specific example, despite its penalty decompositions are different from RpStacks.

Figure 6b shows another optimization case using a different workload. In this example, FMT fails to catch FP multiplication/division events because it cannot correctly label the lost cycles when multiple events occur simultaneously (as discussed in Section II-C). Because it incorrectly accounts the Fmul event as L1D and base event, its performance prediction on any design point with Fmul or L1D optimization becomes inaccurate. In contrast, RpStacks and CP1 precisely estimate the performance. From CP1's stack composition, we can deduce that the workload has small overlaps between events, at least for the design points analyzed in the example.

We observe that RpStacks shows robust accuracy over various workloads and optimization scenarios, while the others fail to achieve consistent results. Section V-B further provides the accuracy results for the full set of workloads and optimization scenarios.

Comparison with other simulator-based explorations. Figure 6c compares two simulator-based design space exploration methods and RpStacks. First, exhaustive exploration aims to cover all structure-latency combinations. Although it produces the richest information, testing all cases is practically infeasible considering the limited resources and time.

Architects therefore use their knowledge and simulation result analysis to modify both the structure and latency during the exploration. Such an insight-driven approach can eliminate unnecessary simulation steps and reduce the exploration time. However, heuristic selection of design may



(a) Sampling

(b) Segmentation

(c) Reduction

Figure 7: Optimization techniques for RpStack generation

omit important points due to wrong judgments. In addition, detailed latency-domain searching process often consumes a large amount of resource for many steps and prohibits the exploration of other designs.

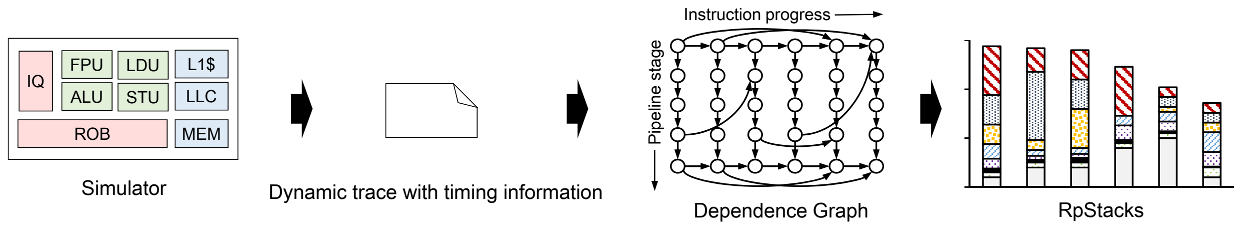
On the other hand, RpStacks covers wider design space than the insight-driven approach and significantly reduces the exploration overhead as well. With RpStacks, architects would explore the structure of processor design just as the insight-driven approach. However, as a single run now covers all possible latency configurations of a structure in much shorter time, they can test more structures in the same amount of time or finish the exploration earlier to save the development costs. After the exploration, the selected design points can be optionally validated using timing simulators to ensure the results are correct.

From the examples illustrated in this section, we conclude that RpStacks efficiently reduces the design space exploration overhead without losing the precision.

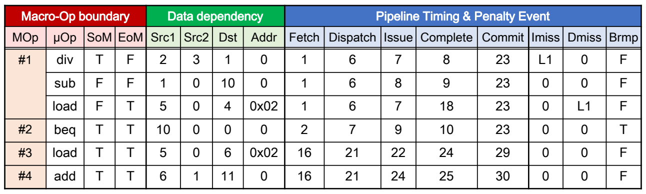
C. Investigating multiple execution paths

The key technique enabling RpStacks is the extraction of performance-critical stall-event stacks from a single simulation. However, investigating all execution paths and events of an application can be extremely time-consuming due to the exponentially increasing number of execution paths. Because RpStacks considers the events between all pipeline stages, each additional instruction can theoretically multiply the number of paths by the number of pipeline stages. To avoid such overhead, our method adopts the three different optimization techniques: *sampling*, *segmentation*, and *reduction*.

Sampling. Sampling is a well-known technique to reduce the simulation or analysis time [1], [2]. To reduce the simulation result analysis time without losing the accuracy, we carefully take weighted samples from our target applications using SimPoint approach [1] (Figure 7a). We generate RpStacks for each 1M SimPoint which represents

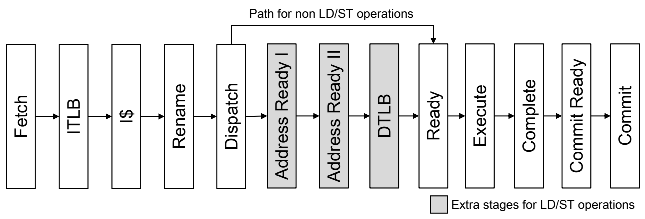


(a) Workflow of RpStacks framework: generating stall-event stacks of distinctive execution paths



Macro-Op boundary				Data dependency				Pipeline Timing & Penalty Event							
MOp	μOp	SoM	EoM	Src1	Src2	Dst	Addr	Fetch	Dispatch	Issue	Complete	Commit	Imiss	Dmiss	Brmp
#1	div	T	F	2	3	1	0	1	6	7	8	23	L1	0	F
	sub	F	F	1	0	10	0	1	6	8	9	23	0	0	F
	load	F	T	5	0	4	0x02	1	6	7	18	23	0	L1	F
#2	beq	T	T	10	0	0	0	2	7	9	10	23	0	0	T
#3	load	T	T	5	0	6	0x02	16	21	22	24	29	0	0	F
#4	add	T	T	6	1	11	0	16	21	24	25	30	0	0	F

(b) An example dynamic trace from simulator



(c) Pipeline stages of our dependence graph model

Figure 8: RpStack framework overview and implementation details

the characteristics of the workloads while minimizing the evaluation overhead. Our method can work simultaneously on multiple independent SimPoints simultaneously to gain extra speedup.

Segmentation. RpStacks uses dependence graphs [11], [16] to track the execution paths of an application. We further optimize the stack generation process by segmenting the dependence graphs obtained from the SimPoints, as shown in Figure 7b. We 1) split a long graph into shorter graphs of a fixed size, 2) generate RpStacks for each graph segment, and 3) add up the stacks of partial segments to form the RpStacks of the original SimPoint. The segmentation significantly reduces the path analysis complexity as the number of paths decreases exponentially with the graph size. It also increases the degree of parallelism by allowing multiple per-segment stack generation instances to run concurrently.

However, the segmentation granularity must be chosen carefully to minimize the errors induced by eliminated paths and extra path traversals. First, dividing a graph into smaller segments eliminates the paths between different segments (dotted arrow lines in Figure 7b) and may degrade the accuracy. Also, handling each segment as an independent graph causes extra path traversals and increases the path lengths. For example, in Figure 7b, the penalty should be measured between node A, the first pipeline stage of the first instruction, and node B, the last pipeline stage of the last instruction. However, RpStacks actually measures the penalties of A-A' and B'-B traversals and uses the sum - which can be larger than the original penalty - to model the original penalty. We thus perform a sensitivity test on the segment size to find the optimal point having tolerable error rate and processing time, and provide the results in Section V-D.

Reduction. As the last step of the optimization, RpStacks performs path reduction at every node in a dependence graph to effectively eliminate non-critical paths during the analysis. Figure 7c illustrates an example where four different paths reach the same destination node. The basic reduction policy is to merge the paths showing the similarity higher than a certain threshold (e.g., path A and B). When merging the paths, we preserve the one with larger overall penalty to keep the paths more critical to performance. Next, when all event components of a path is smaller than that of another path (e.g., path C and A), we simply eliminate the smaller path as it has no chance of becoming the critical path. Finally, we keep every path with unique event combinations to preserve the diversity. For example, path D in Figure 7c is preserved to cope with all possible latency modification scenarios (e.g., increasing Event 4 latency), even though its total penalty is small. The details of calculating the similarity and determining the uniqueness are explained in Section IV-E.

IV. Implementation

A. Overview of the framework

Figure 8a illustrates the high level flow of RpStacks framework. The flow consists of three major steps: 1) dynamic trace generation from timing simulator, 2) dependence graph generation from the trace output, and 3) RpStacks generation via graph analysis. In the following sections, we describe each step in detail.

B. Dynamic trace generation using timing simulator

To generate a dependence graph for analyzing an application and building event stacks, our timing simulator keeps the log of detailed microarchitectural events occurred during a simulation. Figure 8b shows an example dynamic trace from the simulator. The simulator logs three different types of information - *MacroOp boundary*, *data dependency*, and *pipeline timing with penalty event logs* - to correctly reproduce the pipeline dependencies in the later steps.

Constraint	Edge	Description
In-order fetch	$I\$_{i-1} \rightarrow F_i$	
Finite fetch bandwidth	$I\$_{i-fbw} \rightarrow F_i$	where *fbw* is the maximum number of instructions that can be fetched in a cycle
Finite fetch buffer size$^+$	$N_{i-fbs} \rightarrow F_i$	where *fbs* is the fetch buffer size
Control dependency	$I_{i-1} \rightarrow F_i$	inserted if instruction i - 1 is mispredicted branch
ITLB access latency	$F_i \rightarrow ITLB_i$	is 0 in case of ITLB hit
I$ access latency	$ITLB_i \rightarrow I\$_i$	is 0 in case of I$ hit
Rename after I$ access	$I\$_i \rightarrow N_i$	
In-order rename	$N_{i-1} \rightarrow N_i$	
Finite reorder buffer	$C_{i-rbs} \rightarrow N_i$	where *rbs* is the re-order buffer size
Finite rename bandwidth	$N_{i-nbw} \rightarrow N_i$	where *nbw* is the maximum number of instruction that can be processed at the rename stage in a cycle
Dispatch after rename	$N_i \rightarrow D_i$	
In-order dispatch	$D_{i-1} \rightarrow D_i$	
Issue dependency$^+$	$E_j \rightarrow D_i$	prefer to select instruction j waiting for the result of an optimizable long-latency instruction (modeling the issue dynamics)
Finite dispatch width	$D_{i-dbw} \rightarrow D_i$	where *dbw* is the maximum number of instructions which can be dispatched in a cycle
Ready after dispatch$^+$	$D_i \rightarrow AR1_i$	
Data dependency$^+$	$P_j \rightarrow AR1_i$	inserted if a ld/st instruction i depends on previous instruction j's result for address calculation
Address calculation$^+$	$AR1_i \rightarrow AR2_i$	
DTLB access latency$^+$	$AR2_i \rightarrow DTLB_i$	
Ready after dispatch	$D_i \rightarrow R_i$	
Finite physical registers	$C_j \rightarrow R_i$	inserted if instruction j releases the physical register instruction i will use
Data dependency	$P_j \rightarrow R_i$	inserted if instruction i depends on previous instruction j's result
Ready after DTLB access$^+$	$DTLB_i \rightarrow R_i$	inserted if the instruction i is load or store
Execute after ready	$R_i \rightarrow E_i$	
Address dependency$^+$	$E_j \rightarrow E_i$	every load i can be executed after all stores $j < i$ be executed or at the same time
Completion after execute	$E_i \rightarrow P_i$	
Cache line sharing	$P_j \rightarrow P_i$	inserted if load i and load j has the same address
In-order commit	$C_{i-1} \rightarrow RC_i$	
Finite commit width	$C_{i-cbw} \rightarrow RC_i$	
μop dependecy$^+$	$P_j \rightarrow RC_{som}$	inserted for all instruction $j >=$ p *som* where *som* is start of macro op instruction
Commit latency	$RC_i \rightarrow C_i$	

⋆ Node descriptions: F, start of instruction fetch; ITLB, ITLB access done; I$, I-Cache access done; N, register renaming and re-order buffer entry allocation; D, issue queue entry allocation; AR1, all data operands for address calculation ready, except address calculation unit; AR2, address calculation; DTLB, DTLB access done; R, all data operands ready, except functional unit; E, execution; P, execution complete; RC, ready to commit; C, commit.

Table I: Constraints modeled in our dependence graph model

MacroOp boundary. RpStacks targets a x86 ISA microarchitecture and an instruction (macro-op) is decoded into multiple micro-ops and issued into the pipeline. However, the instruction commit should be done with macro-op granularity instead of micro-op to keep the correct order of instruction stream. This issue/commit granularity mismatch introduces a new type of dependency in the commit stage. For accurate performance modeling, our simulator records whether a micro-op is the Start of Macro-op (SoM) or the End of Macro-op (EoM) and uses the information to track the commit dependency of micro-ops.

Data dependency. To track the data dependencies and corresponding penalties, the simulator records physical registers and the memory address usage of each micro-op instruction. The recorded information later forms the dependency edges in our sgraph model.

Pipeline timing and penalty event. The simulator records the pipeline timings of all micro-op instructions to construct the basic pipeline penalty edges in a dependence graph. In case of an exceptional event occurrence (e.g., cache miss, branch misprediction), the simulator also logs the event to the corresponding instruction to note the instruction has undergone a special event.

C. The dependence graph model

The second step is to generate a dependence graph from the dynamic trace of the simulator. Our graph model targets a modern out-of-order superscalar x86 ISA microarchitecture with micro-ops, whereas previous work models simpler RISC architectures [11], [12].

Figure 8c illustrates the pipeline stages of our dependence graph model. We model the architecture pipeline using a 10-node graph, where memory operations go through three extra stages for address generation and DTLB access.

Table I summarizes the constraint events (i.e., dependencies between pipeline stages) our graph model considers. Note that some of the constraints are originally from previous studies [12], [16]. Compared to the existing dependence graph modeling work, our model has a richer collection of new constraints to improve the accuracy of performance modeling and prediction. The newly added entries are denoted with $^+$ in the table.

Modeling the issue dynamics. In fact, out-of-order issue queue dynamically affects the instruction dispatch timing. Our graph model reproduces the instruction issue timing of the baseline latency configuration. As the timing may change for other configurations, we insert a small perturbation in the graph model to cope with the dynamics. Specifically, when creating the issue dependency edge, we prefer to make an instruction (to be dispatched) wait for the instruction which has a data dependency with optimizable events (e.g., data access, long-latency operations). This allows a chain of instructions to dispatch earlier/later depending on the latency of data-generating optimizable events, which RpStacks targets to

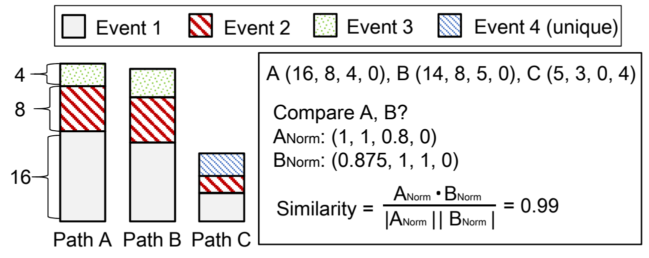


Figure 9: Path comparison using modified cosine similarity

change.

As the result of this step, we obtain a dependence graph where each node represents a specific pipeline stage of an instruction, and each edge indicates a dependency and its penalty between two pipeline stages.

D. RpStacks generation

In the final step, we traverse the dependence graph to find distinctive performance-critical paths and build corresponding stall-event stacks (RpStacks). We start the traversal from the top leftmost node, which represents the first pipeline stage of the first instruction. The process is similar to breadth-first search; we visit the nodes in a topological order. Each pipeline stage (node) contains the stall-event stacks representing the penalties of the paths reaching the node. In other words, we can identify the costs to reach a certain node by investigating these stacks. For each edge traversal, we accumulate the penalty (edge weight) in the stall-event stack to count the cycles required to reach the next pipeline stage. When we visit a node, we apply the reduction method described in Section III-C to dynamically eliminate non-critical paths. Lastly, unlike breadth-first search, a node can be visited multiple times to consider all paths reaching the node.

The traversal ends when we reach the final stage of the last instruction (the bottom rightmost node). The paths remaining in the last node become the representative performance-critical paths, as they survived our reduction and elimination methods removing non-critical execution paths. Accordingly, the stall-event stacks of these paths become RpStacks.

Addressing the branch misprediction penalty. In a dependence graph, the branch misprediction penalty cannot be eliminated by setting the edge weight to zero, as zero-weight edge still implies an ordering dependency between two instructions. Therefore, the branch predictor design falls into the structure category. To test different prediction algorithms, a dependence graph and its corresponding RpStacks should be generated for each predictor design. The resulting sets of RpStacks can then predict the performance of the different predictors.

E. Path reduction using similarity and uniqueness

As introduced in Section III-C, we use *similarity* to merge and eliminate non-critical paths and *uniqueness* to prevent the accidental reduction of performance-critical paths.

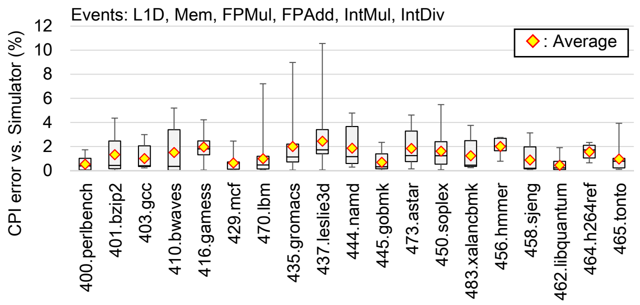


Figure 10: Accuracy of dependence graph model

Similarity. RpStacks uses a variant of *cosine similarity* [17] to calculate the similarity between two different paths. Cosine similarity uses the cosine angle between two vectors to measure the similarity. The maximum similarity is achieved when the vectors are in parallel, and the minimum similarity is obtained when the vectors are orthogonal. The similarity value varies from zero to one; the larger the value, the higher the similarity.

Figure 9 illustrates a path comparison example. We start by changing the event stack of a path to a penalty vector. The penalty of each kind of event becomes a single dimension in the penalty vector. However, because cosine similarity favors the dimension with larger magnitude, we normalize the magnitude of each dimension before comparing two vectors, with the larger magnitude among the two becoming the unit size. By the normalization, we compare the penalty vectors providing the fairness between different events. Finally, the similarity is calculated using the simple equation shown in the figure.

To determine the appropriate similarity threshold value for merging the paths, we perform a sensitivity analysis and show the results in Section V-D.

Uniqueness. RpStacks considers a path unique and excludes it from the merging process, if it has a penalty component that the others do not have. In Figure 9, path C is preserved as it has a unique penalty component (Event 4). Introducing the uniqueness concept greatly improves the prediction accuracy as it preserves the diversity of event stacks, and we show its impact in Section V-D.

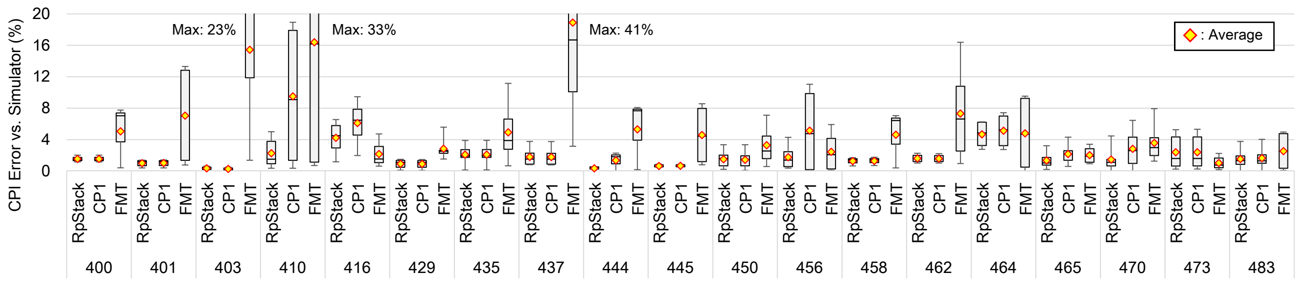
V. Evaluation

A. Methodology

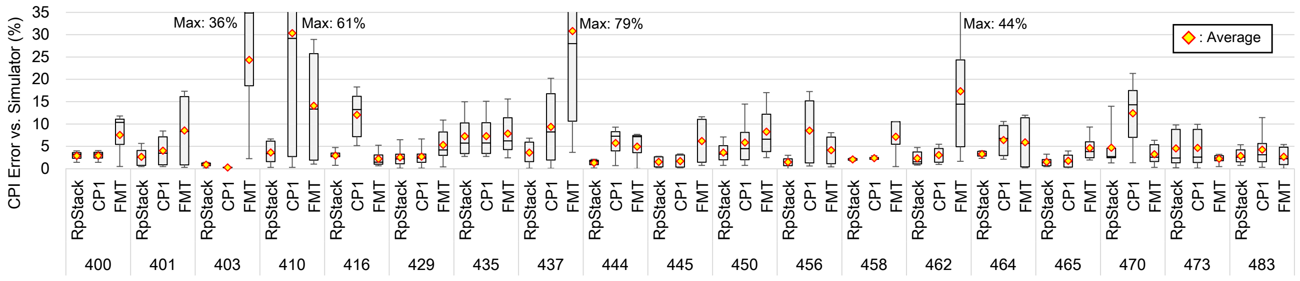
We use MARSSx86 [13], a cycle-accurate full-system simulator as our baseline simulator. Table II summarizes the

ROB / IssueQ / LSQ	128 / 36 / 64
Pipeline width	Fetch / Rename / Dispatch / Issue / Commit: 4
# functional units	LD(2), ST(2), FP(2), BaseALU(4), LongALU(2)
FU Latencies(cycles)	LD(2), IntMul(4), IntDiv(32), FP(6), FPDiv(24)
L1 I-Cache	48KB 4-way set-assoc, 2 cycles
L1 D-Cache	48KB 4-way set-assoc, 4 cycles
L2 Cache	4MB 8-way set-assoc, 12 cycles
Main memory access	133 cycles

Table II: Details of target microarchitecture model

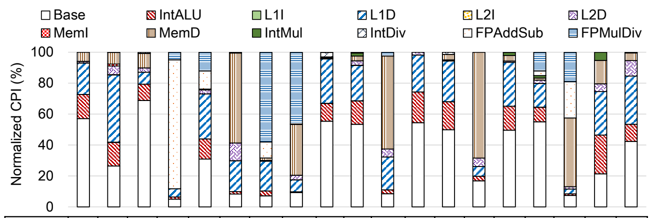


(a) Latency of major bottlenecks reduced to 50%



(b) Latency of major bottlenecks reduced to 10∼25%

Figure 11: Performance prediction accuracy: RpStacks, single critical path (CP1), and pipeline-stall analysis (FMT)



Workload	400	401	403	410	416	429	435	437	444	445	450	456	458	462	464	465	470	473	483
Base CPI	0.49	0.77	0.49	0.46	0.37	4.30	0.90	0.80	0.40	0.57	1.22	0.32	0.59	1.24	0.41	0.43	0.91	0.67	0.49
Bottlenecks	L1D MEM	L1D L2D MEM	L1D MEM	L1D Fadd Fmul	L1D Fadd Fmul	L1D L2D MEM	L1D Fadd Fmul	L1D MEM Fmul	L1D Idiv	L1D L2D Imul	L1D L2D MEM Fmul	L1D L2D MEM	L1D MEM	L1D L2D MEM	L1D MEM Imul	L1D Fadd Fmul	MEM Fadd Fmul	L1D L2D MEM Imul	MEM L2D L1D

Figure 12: Bottlenecks and baseline CPIs of the applications

details of our architecture model. As described in Section IV, we modify the simulator to generate dynamic instruction trace with timing information, and implement the trace-to-dependence graph converter and RpStacks generator in C++. We also implement a pipeline-stall based analysis method on our simulator, based on the Frontend Miss Table (FMT) implementation described by Eyerman *et al.* [8].

We use SPEC CPU 2006 benchmark suite [14] to evaluate our methodology. Figure 10 shows the errors of our graph model (vs. simulator) for various applications and optimization scenarios. For optimizations, we impose one-cycle latency to the combinations of up to two events listed in the figure. The whiskers represent the minimum and maximum errors and the boxes show the second and the third quartiles. The results show that our dependence graph accurately models the behavior of the target microarchitecture, even for extreme optimization cases. As we apply less aggressive optimizations in the following sections, the actual impact of graph model error on RpStacks is expected to be smaller. Improving the graph model can increase the upper-bound accuracy of our method, however, is out of the scope of this paper.

B. *Performance prediction accuracy*

We first compare the accuracy of RpStacks and two popular simulation result analysis methods: *single critical path (CP1)* and *Frontend Miss Table (FMT)*. We apply latency optimizations to *up to two major bottleneck events* of the applications, and use the three methods to predict the performance (CPI). The performance prediction error against the simulator is used to evaluate the accuracy of the methods. As each application exhibits different bottleneck characteristics, we identify the major bottlenecks of the applications using the CPI stack of baseline architecture (Figure 12). The CPI of the baseline architecture is also provided as the size of prediction errors depends on the magnitude of it.

Figure 11a shows the error of the three methods when the latencies of the bottleneck events are *reduced to half*. For some applications, RpStacks and CP1 successfully predict the performance because the bottleneck events do not contribute significantly to the overall performance and/or the overlaps between different events are small. The applications without diverse long-latency events (e.g., integer applications such as 400, 401, and 429) tend to show such simple behavior. On the other hand, FMT frequently mispredicts the performance even for these applications, due to its inaccurate penalty accounting method. For the other applications, both CP1 and FMT have large errors as they do not consider the overlapping of events and incorrectly estimate the impact of penalties. This illustrates that addressing multiple

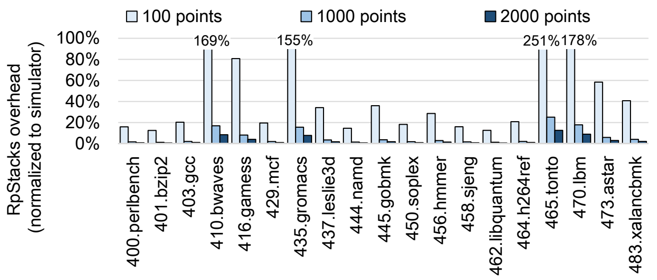


Figure 13: RpStacks design space exploration overhead

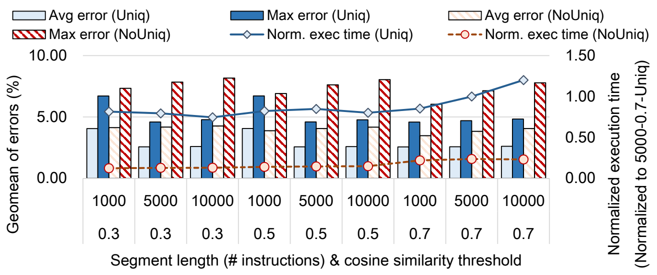


Figure 14: RpStacks execution parameter sensitivity

performance-critical paths is necessary to correctly evaluate a processor design against complex applications.

Next, we apply more aggressive optimizations by reducing the latencies to 10~25% (Figure 11b). The latency reduction ratio varies in accordance with the bottleneck event to ensure integer-cycle operations. Due to the large adjustments of latencies, many applications start to reveal the interactions and overlaps which were hidden in the previous scenario. As CP1 and FMT fail to consider such complicated behaviors, their predictions show significantly large errors. In contrast, RpStacks maintains high accuracy by making predictions based on multiple execution scenarios and correctly accounting the previously unseen interactions of events.

In summary, unlike the existing simulation result analysis methods, RpStacks provides reliable performance estimations for a variety of workloads and optimization cases, eliminating the possibility of wasting precious design exploration steps.

C. *Design space exploration overhead*

We now compare the design space exploration overhead of RpStacks and the baseline timing simulator. As RpStacks focuses on reducing the latency domain, we fix a structure and consider only latency design points in the evaluation. We omit the comparison with simulation acceleration methods as it can be easily assumed from the speedup values in Figure 2a.

Figure 13 shows the normalized design space exploration time of RpStacks for various applications and number of design points. RpStacks' exploration time is independent from the number of design points because a single run covers all possible latency combinations. In contrast, exploration using the simulator suffers from the increasing exploration time as the number of design points gets larger. Consequently, RpStacks starts to outperform the simulation when there are more than 38 points (on average). Since architects would typically investigate hundreds or thousands of designs, RpStacks is expected to significantly accelerate the design space exploration process compared to slow timing simulations.

D. *RpStacks execution parameter sensitivity*

We perform a sensitivity test on the *segment length* and *cosine similarity threshold* to find the best parameters for RpStacks execution. Figure 14 shows the geometric means of the average errors, max errors, and normalized execution times of all workloads. The average and max errors are obtained from the optimization cases shown in Figure 11b. We also discuss the impact of *uniqueness preservation*.

First, disabling the unique path preservation degrades the RpStacks' accuracy to an unacceptable level. Although the speed benefit is attractive, RpStacks often shows large peak errors (40+%) and thus becomes ineffective. Turning on the preservation increases the diversity of the paths and significantly improves the accuracy (max error less than 15%). In addition, RpStacks becomes less sensitive to the cosine threshold as the uniqueness preservation now has the first-order impact. For segment length, increasing the length increases the accuracy to some extent, as too small segments can eliminate the application characteristics and introduce errors as described in Section III-C. However, increasing the parameters above a certain level does not improve the accuracy and only increases the execution time, because RpStacks starts to preserve non performance-critical paths as well.

Considering the speed and accuracy, we decide to enable the uniqueness preservation and use 0.7 and 5000 for the cosine threshold and segment length, respectively.

VI. Related work

Analytic modeling approaches. An analytic model can be classified into three different categories: *empirical model*, *mechanistic model*, and *hybrid model*. The empirical model is built by using machine learning techniques without microarchitectural knowledge. Joseph *et al.* [18] use a linear regression model for performance prediction, while Lee *et al.* [19]–[22] use a spine-based regression model to capture non-linearity as well. A recently proposed ArchRanker [23] suggests to use a ranking model rather than a performance model to focus on and improve one-on-one design comparisons. Second, mechanistic models [9], [24]–[26] are constructed from microarchitectural knowledge and insights. Eyerman and Eeckhout [9] propose interval analysis, which depicts the operation of an out-of-order processor using the flow of issued instructions. The flow is split into several intervals according to the miss events interrupting the instruction issues. Heirman *et al.* [26] improve the interval analysis by considering critical paths as well. Finally, hybrid models [27] mix the approaches of empirical models and mechanistic models to leverage the strengths of the both.

Approaches using performance monitoring unit. Some researchers consider employing PMU to perform pipeline-stall analysis and study processor behavior. Initial work [28], [29] simply counts the number of miss events to address the penalties, which are not suitable for describing out-of-order superscalar processor behaviors due to the event overlaps. ProfileMe [30] proposes paired sampling which investigates an instruction as well as its closely related nearby instructions; it can capture the overlap between the two instructions, although not complete. Eyerman *et al.* [8] invent interval analysis to accurately address event penalties and identify the performance bottlenecks. However, the method cannot detect overlapped event penalties and thus is not suitable for performance predictions.

Critical path approaches. The critical path analysis has been introduced to find the bottlenecks of complex out-of-order superscalar processors. [10], [11], [16], [31]–[33] use the critical path theory to exploit the criticality and slack of instructions to optimize microarchitecture design. Saidi *et al.* [34] apply critical path analysis to full-system scope to identify key bottlenecks in workloads. Tullsen *et al.* [16] measure the criticality of an instruction by modifying the edges of the dependence graph. Fields *et al.* [12] propose the concept of interaction cost to accurately model the interactions between instructions. They perform trace-based simulations on multiple dependence graphs to calculate the interaction costs when the hardware constraints (e.g., event penalties) change. Compared to the work of Fields *et al.*, RpStacks requires only a single trace simulation on the graph, showing lower architecture exploration overhead.

VII. Conclusion

Due to slow timing simulations, exploring a wide variety of design options have become the main bottleneck in a processor development process. In this paper, we proposed RpStacks, a fast and accurate design space exploration method to identify key performance bottlenecks and make correct predictions on processor performance using a single simulation. RpStacks delivers highly accurate performance predictions by accounting hidden performance-critical paths, and enables agile architecture design evaluation. The fast and accurate prediction of RpStacks saves architects of time-consuming simulation steps, ultimately accelerating the processor development cycle.

Acknowledgment

This research was supported by Basic Science Research Program and Global PH.D Fellowship Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT & Future Planning(NRF-2014R1A1A1003746) and the Ministry of Education (2014052111), and by the MSIP under the IT Consilience Creative Program(NIPA-2014-H0201-14-1001).

References

[1] 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, pp. 45–57.

[2] 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 international symposium on Computer architecture*, 2003, pp. 84–97.

[3] 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 IEEE/ACM International Symposium on Microarchitecture*, 2007, pp. 249–261.

[4] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, and K. Asanovi, “Ramp gold: An fpga-based architecture simulator for multiprocessors,” in *Proceedings of the 47th Design Automation Conference*, 2010.

[5] M. Pellauer, M. Adler, M. Kinsy, A. Parashar, and J. Emer, “Hasim: Fpga-based high-detail multicore simulation using time-division multiplexing,” pp. 406–417, 2011.

[6] J. E. Miller, H. Kasture, G. Kurian, C. G. III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, “Graphite: A distributed parallel simulator for multicores,” in *International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 2010, pp. 1–12.

[7] T. E. Carlson, W. Heirman, and L. Eeckhout, “Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation,” in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 52.

[8] 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, pp. 175–184.

[9] ——, “A mechanistic performance model for superscalar out-of-order processors,” *ACM Trans. Comput. Syst.*, pp. 3:1–3:37, 2009.

[10] 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, pp. 47–58.

[11] 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, pp. 74–85.

[12] B. A. Fields, R. Bodik, M. D. Hill, and C. J. Newburn, “Interaction cost and shotgun profiling,” *ACM Trans. Archit. Code Optim.*, vol. 1, no. 3, Sep. 2004.

[13] 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.

[14] "Standard performance evaluation corporation (spec) website." [Online]. Available: http://www.spec.org/

[15] P. G. Emma., "Understanding some simple processor-performance limits," in *IBM journal of Research and Development*, May 1997.

[16] 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, pp. 104–.

[17] J. Han, M. Kamber, and J. Pei, *Data mining: concepts and techniques*. Morgan kaufmann, 2006.

[18] P. Joseph, null Kapil Vaswani, and M. Thazhuthaveetil, "Construction and use of linear regression models for processor performance analysis," *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, vol. 0, pp. 99–108, 2006.

[19] 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*, ser. ASPLOS XII. New York, NY, USA: ACM, 2006, pp. 185–194. [Online]. Available: http://doi.acm.org/10.1145/1168857.1168881

[20] 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*, ser. ASPLOS XIII. New York, NY, USA: ACM, 2008, pp. 36–47. [Online]. Available: http://doi.acm.org/10.1145/1346281.1346288

[21] 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*, ser. MICRO 41. Washington, DC, USA: IEEE Computer Society, 2008, pp. 270–281. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2008.4771797

[22] 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*, ser. HPCA '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 340–351. [Online]. Available: http://dx.doi.org/10.1109/HPCA.2007.346211

[23] 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.

[24] P. G. Emma, "Understanding some simple processor-performance limits," *IBM J. Res. Dev.*, vol. 41, no. 3, pp. 215–232, May 1997. [Online]. Available: http://dx.doi.org/10.1147/rd.413.0215

[25] T. S. Karkhanis and J. E. Smith, "A first-order superscalar processor model," in *Proceedings of the 31st annual international symposium on Computer architecture*, ser. ISCA '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 338–. [Online]. Available: http://dl.acm.org/citation.cfm?id=998680.1006729

[26] 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*, ser. IISWC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 38–49. [Online]. Available: http://dx.doi.org/10.1109/IISWC.2011.6114195

[27] A. Hartstein and T. R. Puzak, "The optimum pipeline depth for a microprocessor," in *Proceedings of the 29th annual international symposium on Computer architecture*, ser. ISCA '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 7–13. [Online]. Available: http://dl.acm.org/citation.cfm?id=545215.545217

[28] 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 Trans. Comput. Syst.*, vol. 15, no. 4, pp. 357–390, Nov. 1997. [Online]. Available: http://doi.acm.org/10.1145/265924.265925

[29] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz, "Performance analysis using the mips r10000 performance counters," in *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, ser. Supercomputing '96. Washington, DC, USA: IEEE Computer Society, 1996. [Online]. Available: http://dx.doi.org/10.1145/369028.369059

[30] 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*, ser. MICRO 30. Washington, DC, USA: IEEE Computer Society, 1997, pp. 292–302. [Online]. Available: http://dl.acm.org/citation.cfm?id=266800.266828

[31] 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, pp. 185 –195.

[32] 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*, ser. ISCA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 13–24. [Online]. Available: http://dx.doi.org/10.1109/ISCA.2008.39

[33] S. Subramaniam, A. Bracy, H. Wang, and G. H. Loh, "Criticality-based optimizations for efficient load processing," in *HPCA*, 2009, pp. 419–430.

[34] A. G. Saidi, N. L. Binkert, S. K. Reinhardt, and T. Mudge, "Full-system critical path analysis," in *Performance Analysis of Systems and software, 2008. ISPASS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 63–74.