DiagSim: Systematically Diagnosing Simulators for Healthy Simulations

JAE-EON JO, Department of Computer Science and Engineering, POSTECH
GYU-HYEON LEE, Department of Electrical and Computer Engineering, Seoul National University
HANHWI JANG, JAEWON LEE, and MOHAMMADAMIN AJDARI, Department of Computer Science and Engineering, POSTECH
JANGWOO KIM, Department of Electrical and Computer Engineering, Seoul National University

Simulators are the most popular and useful tool to study computer architecture and examine new ideas. However, modern simulators have become prohibitively complex (e.g., 200K+ lines of code) to fully understand and utilize. Users therefore end up analyzing and modifying only the modules of interest (e.g., branch predictor, register file) when performing simulations. Unfortunately, hidden details and inter-module interactions of simulators create discrepancies between the expected and actual module behaviors. Consequently, the effect of modifying the target module may be amplified or masked and the users get inaccurate insights from expensive simulations.

In this article, we propose DiagSim, an efficient and systematic method to diagnose simulators. It ensures the target modules behave as expected to perform simulation in a healthy (i.e., accurate and correct) way. DiagSim is efficient in that it quickly pinpoints the modules showing discrepancies and guides the users to inspect the behavior without investigating the whole simulator. DiagSim is systematic in that it hierarchically tests the modules to guarantee the integrity of individual diagnosis and always provide reliable results. We construct DiagSim based on generic category-based diagnosis ideas to encourage easy expansion of the diagnosis.

We diagnose three popular open source simulators and discover hidden details including implicitly reserved resources, un-documented latency factors, and hard-coded module parameter values. We observe that these factors have large performance impacts (up to 156%) and illustrate that our diagnosis can correctly detect and eliminate them.

CCS Concepts: • **Computing methodologies** → **Simulation support systems**; • **Computer systems organization** → *Architectures*;

Additional Key Words and Phrases: Simulator diagnosis, timing simulator verification, microbenchmarks

This work was partly supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (NRF-2015M3C4A7065647, NRF-2017R1A2B3011038), and Institute for Information & Communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. R0190-15-2012).

Authors' addresses: J.-E. Jo, H. Jang, J. Lee, and M. Ajdari, Dept. of Computer Science and Engineering, POSTECH, Pohang, Gyeongbuk, Korea; emails: {jojaeeon, hanhwi, spiegel0, majdari}@postech.ac.kr; G.-H. Lee and J. Kim (corresponding author), Dept. of Electrical and Computer Engineering, Seoul National University, Seoul, Korea; emails: {guhylee, jangwoo}@snu.ac.kr.

https://doi.org/10.1145/3177959

ACM Reference format:
Jae-eon Jo, Gyu-hyeon Lee, Hanhwi Jang, Jaewon Lee, Mohammadamin Ajdari, and Jangwoo Kim. 2018. DiagSim: Systematically Diagnosing Simulators for Healthy Simulations. *ACM Trans. Archit. Code Optim.* 15, 1, Article 4 (March 2018), 27 pages.
https://doi.org/10.1145/3177959

1 INTRODUCTION

Simulators are by far the most popular tools to study computer architecture, examine creative ideas, and derive new designs. They model various microarchitecture modules along with the cycle-level operation details and inter-module dependencies. Such cycle-level full-system simulators provide accurate architectural insights to eventually derive the next-generation computer system. A quick survey on recent publications (Figure 1) also confirms that they are widely used by the academia.

However, modern simulators have become prohibitively complex to fully understand and utilize. The growth of simulators is natural as the architecture is getting sophisticated over time and many developers are contributing to model new aspects. While this allows us to study the state-of-the-art architectures using simulators, it increases the overhead of understanding and using the simulators at the same time.

Consequently, simulator users start to give up on understanding the full details of simulators. They instead analyze and modify only the modules of interest. Unfortunately, the hidden/undocumented modeling details and inter-module interactions of modern simulators create *discrepancies* between the user-expected and actual simulator behaviors [25]. The discrepancies can exaggerate or underestimate the performance impact of modifying the target modules, to provide inaccurate insights which eventually lead to critical system development failures.

To address this problem, we propose *DiagSim*, an efficient and systematic open source software to diagnose simulators. It enables the simulator users to easily verify whether the simulator modules behave as expected, to promote *healthy* (i.e., accurate and correct) simulations.

Specifically, DiagSim uses a set of short-running microbenchmarks (i.e., diagnoses) to *measure* the effective value of modeling parameters and *verify* whether the modules behave as expected. As it quickly detects the modules showing mismatching behaviors, the users may *efficiently* inspect only the problematic modules without investigating the whole simulator. In addition, DiagSim provides a *systematic* way to safely and efficiently diagnose the modules. It uses *diagnosis dependency map* to hierarchically perform diagnoses and ensures that a diagnosis does not fail due to external factors (e.g., integrity of the other modules). This map prevents running redundant diagnoses (e.g., interacting modules hampering each other's diagnosis) and lets the users safely trust the diagnosis results. DiagSim is also highly *expandable* and *portable* because we classify microarchitecture modules and develop diagnoses based on *generic* ideas for each category.

We diagnose three popular open source simulators (gem5 [7], MARSSx86 [28], and Multi2Sim [32])[1] and discover different hidden modeling details and interactions, which create the discrepancy between the user's understanding of the simulator and actual simulator behavior. The examples include implicitly reserved resources, undocumented latency factors, and hard-coded module parameter values (Section 5.2 and Table 5 summarize the results). Our evaluation using SPEC CPU 2006 shows that these issues create large performance changes (up to 156%), but DiagSim allows one to efficiently detect and eliminate them.

[1]We use the earlier versions of the simulators (circa 2014) to emphasize the vulnerability. Most of the simulator-oriented problems discsussed in this article might have been or would be fixed by later versions of the simulators.

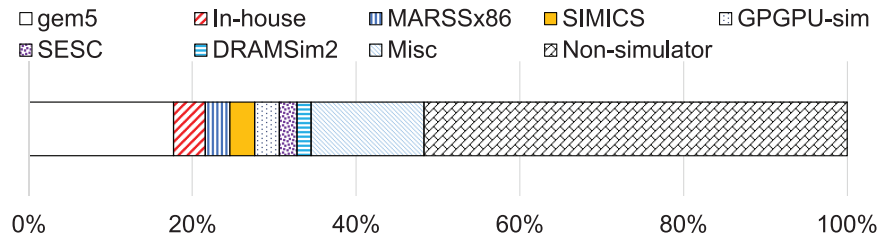


Fig. 1. Classification of evaluation methodology (MICRO & ISCA, 2015 & 2016): Simulators [2, 7, 8, 22, 28–30, 32] occupy a large portion.

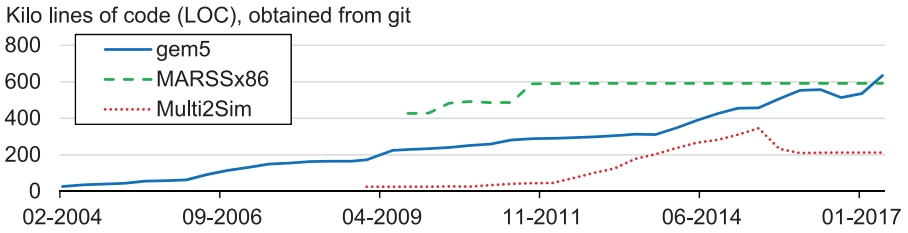


Fig. 2. Lines of Code (LOC) of three popular open-source full-system simulators.

The contributions of this article are as follows:

- **Critical problem.** We show that simulators' hidden modeling details and inter-module interactions incur significant performance discrepancies, which must be handled for healthy simulations.
- **Highly practical method.** DiagSim is generic and therefore can be applied to various simulators. Also, the current implementation (x86 ISA assembly-based microbenchmarks) allows one to quickly diagnose various x86 simulators with negligible overhead.
- **Realistic validation.** DiagSim successfully detects and handles large performance discrepancies in three widely used open source simulators.
- **Open source software.** DiagSim is open source software (https://hpcs.snu.ac.kr/DiagSim/), which encourages users to share their own diagnosis ideas.

The rest of the article is organized as follows. Section 2 introduces the existing methods as well as their limitations to motivate DiagSim. We introduce the key ideas of DiagSim in Section 3 and elaborate the details in Section 4. The evaluation and discussion are provided in Section 5 and Section 6. We discuss the related work in Section 7 and conclude the article in Section 8.

2 BACKGROUND AND MOTIVATION

In this section, we show the need for a systematic method to diagnose simulators. We also introduce the existing simulator diagnosis methods and discuss their limitations to motivate DiagSim.

2.1 Need For an Efficient Simulator Diagnosis Method

Simulators have been the first choice for computer architecture study and exploration, for they accurately model detailed behaviors to derive accurate results. As computer architecture has evolved over time, the simulators have been accordingly augmented to account for the changes. As a result, modern simulators have become an extremely complex piece of software.

Figure 2 shows the lines of code (LOC) for popular open source simulators over time. The 200K+ LOC strongly discourage the users from understanding the details before utilizing the simulators.

Fig. 3. Performance impact of L1 data cache prefetcher on gem5 simulator, for various architectural settings. The apparently independent parameters (i.e., register count and operation latency) affect the performance benefits. Note that we show only the sensitive workloads and parameters.

Furthermore, as the volume of the code grows, the users should put constant effort to track the changes even if they manage to understand the simulators.

To avoid such excessive overheads, the majority of simulator users inspect and modify *only* the modules of interest (i.e., target modules). Often, simulators have a modular design and carry documentations and example configuration files, to let the users easily modify and utilize simulators without fully understanding the underlying details.

However, such a practice eventually leads the users to *ignore* the non-target modules which can be *critical* in forming the comprehensive performance of a design. Figure 3 illustrates such an example. On gem5 simulator, we introduce an L1 data cache prefetcher and measure its performance impact. We change the architectural parameters of the non-target modules (i.e., modules other than L1 data cache) to check if they have an effect on the performance benefits of the prefetcher. For L2 cache latency, an experienced architect may expect performance effects because L1 cache's latency overlapping behaviors would change upon L2 cache latency changes. Surprisingly, the results show that even *seemingly independent* modules can also have performance impacts which are significant enough to affect design decisions (e.g., whether to introduce the prefetcher). It also indicates that a mismatch in the behavior of non-interesting modules can easily hamper the decisions from simulations. As an instance, if the effective number of FP registers is smaller than expected, we would overestimate the benefit of prefetchers for certain workloads (e.g., 482).

We therefore need an efficient diagnosis methodology to thoroughly check whether all simulator modules (i.e., not just the target modules) match the expected behaviors. The diagnosis would facilitate *healthy* simulations which provide accurate and meaningful insights.

2.2 Limitations of the Existing Simulator Diagnosis Methods

Many existing studies attempt to validate or test microarchitecture module behaviors and parameters; therefore, they can be considered as a simulator diagnosis. This section illustrates them and discusses the limitations to motivate DiagSim.

[9, 11] are early studies to validate a given microarchitecture. Based on the given architecture design, they define the fault model (i.e., what may go wrong) and check if the simulator passes the tests. A test passes if the pipeline shows the exact expected behavior. While the approaches allow us to validate simulator behaviors without investigating the details, it is less general in that the tests are bound to a specific hardware platform and design. DiagSim instead categorizes the modules into few types and provides generic methodologies to diagnose each type. In addition, the tests proposed in [9, 11] only tell whether the simulator passes or fails. Our diagnosis instead detects the effective module parameters to further help the users to adjust the behaviors if necessary. Lastly, the studies assume the independence between the modules, which is unrealistic. DiagSim tracks the inter-module interactions and alerts if they can be the root cause of discrepancies.

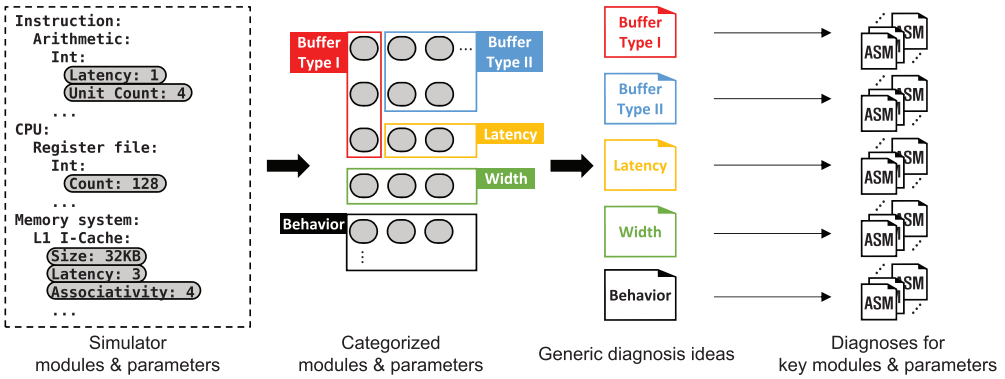


Fig. 4. DiagSim's diagnosis generation process. We categorize the modules & parameters, and provide generic diagnosis ideas per category to minimize the development overheads.

Another group of studies [14, 19] provides assembly-based microarchitecture performance benchmarks. The benchmarks test (1) the latency and throughput of known instructions and (2) the performance against various instruction patterns. First, the instruction latency and throughput benchmarks repeat a single instruction to measure the performance, but do not elaborate *why* such a performance is obtained. For example, we would not know which resources (e.g., register file, execution port) contribute to the performance and become the bottleneck. As DiagSim identifies individual resource parameters (e.g., register file size, execution width), it can better explain how each factor contributes to the performance. Second, the pattern benchmarks they provide can be used to reverse-engineer module behaviors and parameters. However, they do not consider inter-module interactions and cannot explain why certain results appear. Our method tracks the interactions to solve this issue. To summarize, these benchmarks are better suited for evaluating the effective design performance, to inform compiler designers, assembly coders, and application optimizers of the performance characteristics.

We further discuss the other testing and validation methods in Section 7.

3 DESIGN GOALS AND KEY IDEAS

In this section, we describe the design goals of DiagSim and the key ideas to achieve the goals.

3.1 Design Goals

Based on the analysis from Section 2, we aim to propose an *efficient*, *generic*, and *systematic* diagnosis for simulators, *to identify hidden modeling details and inter-module interactions*. The following paragraphs explain why such properties are necessary. We summarize the functional requirements as well.

Efficient. First and foremost, diagnosis must be efficient. Since the main purpose of diagnosis is to avoid the large overhead of manually investigating simulators, the diagnosis process must maintain small overhead.

Generic. The diagnosis must be generic to cover various designs and modules with negligible overhead. This also relates to the efficiency; if the diagnoses are difficult to maintain and apply, we lose our key motivation. In this article, we aim to cover any out-of-order superscalar systems.

Systematic. The diagnosis must be systematic to correctly validate and analyze the module behaviors. We already illustrated such an importance in Section 2.1.

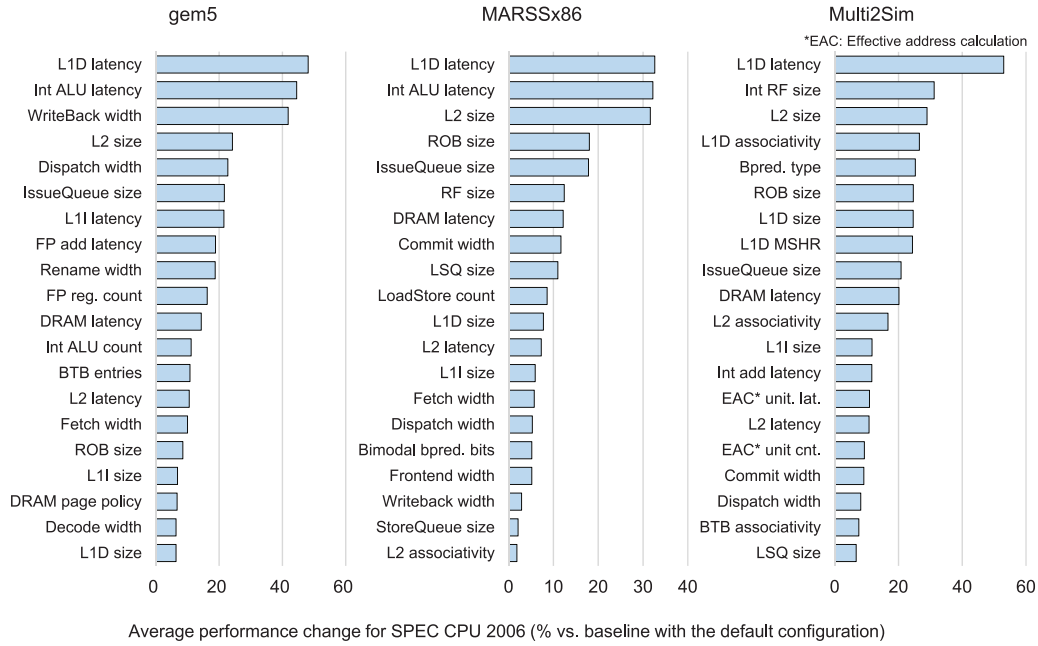


Fig. 5. Performance criticality of modules and parameters for three different simulators. We change the parameter values and observe how much the workload performance changes. The parameters causing larger changes have larger criticality. We report the average performance change over SPEC CPU 2006 workloads.

Functional requirements. The diagnosis must be able to check if a target module behaves as expected. Upon finding any discrepancies, it should identify hidden modeling details or interactions which may have caused the discrepancies. It should suggest how to fix the target module or the interacting modules to eliminate the discrepancies if possible. If the fix incurs nontrivial overhead (e.g., restructuring the simulator), it should explain which modules are the root causes of the discrepancies to inform the users of the caveats.

3.2 Key Ideas

We now describe the key ideas of DiagSim to achieve the design goals.

Category-based diagnosis generation. As simulators consist of many modules and parameters, designing individual diagnoses for each module and parameter requires significant efforts. To minimize the overhead of developing and maintaining diagnosis, we propose a *category-based* diagnosis generation (Figure 4). Specifically, we find that a group of modules and parameters can be validated using similar diagnosis ideas. Leveraging these characteristics, we classify the simulator modules and parameters into five categories—*Buffer Type I & II, Latency, Width, and Behavior*—and propose a generic diagnosis idea for each category to facilitate expanding the diagnoses. The classification is very straightforward. For instance, the *Latency* category includes various operation latencies (e.g., integer/FP arithmetic, cache access). The generic diagnosis idea of the *Latency* category is to form a chain of instructions whose latency is be diagnosed. From such generic ideas, we can easily make diagnoses for parameters in the category. Section 4 provides the details of per-category diagnosis ideas and Section 6 discusses how we effectively reduce the diagnoses implementation effort.

In this work, we focus on the modules and parameters which have a large impact on constructing the overall performance (i.e., high performance criticality). As a module/parameter with higher

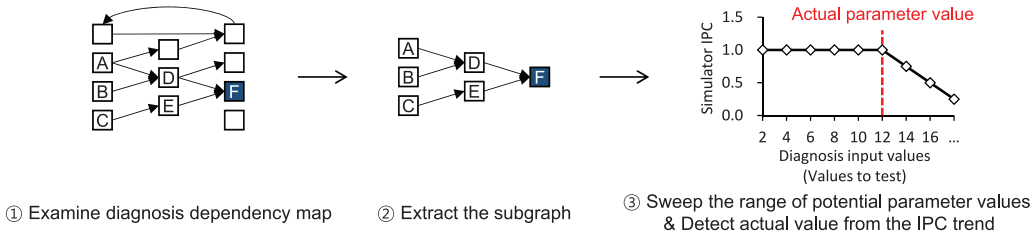


Fig. 6. Overview of DiagSim process.

Fig. 7. Practical challenges of DiagSim.

criticality would introduce more performance deviations if it fails to match the expected behavior, it is crucial to ensure the correctness of these parameters. To measure the criticality, we change the parameters (i.e., sweep from half to 4× the default value) and observe how much the performance changes. Figure 5 lists the top critical parameters for the three simulators. Note that the simulators show different sensitivity to the parameters but the list of critical parameters is similar across the simulators. This is mainly because the simulators are configured to have similar microarchitectural baselines. The analysis result indicates that focusing on these (common) performance-critical modules and parameters would allow us to successfully handle multiple simulators with (slightly) different performance sensitivities.

High-speed diagnosis. To minimize the overhead of utilizing diagnoses, we construct assembly-based microbenchmarks to diagnose the modules. The microbenchmarks are designed to run for short so that the overhead is negligible. The users may frequently utilize DiagSim (e.g., for every simulator modification) to ensure that they are performing healthy simulations. The microbenchmarks mainly use *IPC patterns* to report the diagnosis results (Section 4 discusses the examples). We decide to use IPC because it is a commonly used performance statistics with a straightforward definition, and thus is less error-prone. Utilizing other statistics (e.g., miss rates) is possible but it would require an additional inspection of the related modules and the statistics themselves; we therefore do not consider this option in the article.

Diagnosis dependency map. For *systematic* validation of the modules, we propose a *diagnosis dependency map*. Often, the diagnosis for a specific module requires the integrity of the other modules. Instead of trying to break such inter-module dependencies and make atomic diagnoses, we propose a dependency map and suggest to check the related modules in a *hierarchical* way so that a module can be examined without (the other modules') integrity concerns. Note that such a thorough diagnosis is feasible because each diagnosis has negligible overhead thanks to our design.

4 DIAGSIM DETAILS

In this section, we describe DiagSim in detail. We first introduce the overview of the diagnosis process and discuss the details of each step. We also provide the implementation details to discuss practical challenges.

Table 1. Details of DiagSim Design (1 of 2)

#	Diagnosis	Category	Operation detail
1	ROB Size	Buffer type I	Vary the # of nops (n) between two long-latency instructions. IPC drops right after n exceeds the ROB size.
2	IQ Size	Buffer type I	Vary the # of movs (n) between two long-latency instructions. IPC drops right after n exceeds the IQ size.
3	LSQ ize	Buffer type I	Execute n L1$-hit loads/stores between two cache-miss incurring loads. IPC drops right after n exceeds the LSQ size.
4	Physical Register Count	Buffer type I	Vary the # of movs (n) between two long-latency instructions. IPC drops when n exceeds the # of physical registers.
5	BTB Size	Buffer type II	Vary the # of branches (n). When n exceeds BTB size, IPC decreases due to increased branch misprediction.
6	BTB Associativity	Buffer type II	Vary the # of branches (n), which are aligned to share a BTB set. When n exceeds the associativity of BTB, IPC decreases due to increased branch misprediction.
7	RAS Size	Buffer type II	Fill RAS with n call instructions. When n exceeds the RAS size, IPC decreases due to increased branch misprediction.
8	Bimodal Bpred Size	Buffer type II	Fill bimodal counter table with n branches. When the table overflows, IPC decreases due to increased branch misprediction.
9	Bimodal Bpred Counter Bit Size	Buffer type II	Execute conditional branches that follow $\{Taken\}^n\{NotTaken\}^n$ pattern, increasing n from 1. IPC starts to increase when $n \geq 2^{counter\_bits}$ as the miss ratio reduces.
10	Two-level Bpred History Size	Buffer type II	Create n distinct branch patterns and vary n, If the pattern becomes too diverse to fit in the history buffer, IPC starts to degrade due to increased branch misprediction.
11	L1D$ Capacity	Buffer type II	Fill L1D$ with pointer-chasing memory accesses and vary the working set. When the working set size exceeds L1D$ capacity, IPC decreases.
12	L1D$ Block Size	Buffer type II	Access memory with stride n and vary n. As n increases, IPC decreases due to increased miss ratio, and IPC becomes minimized when n exceeds the block size.
13	L1D$ Associativity	Buffer type II	Fill the cache with n pointer-chasing loads that share a set. When n exceeds the set's associativity, IPC decreases.
14	L2D$ Capacity	Buffer type II	Similar to L1D$ Capacity diagnosis (#11)
15	L2D$ Block Size	Buffer type II	Similar to L1D$ Block Size diagnosis (#12)
16	L2D$ Associativity	Buffer type II	Similar to L1D$ Associativity diagnosis (#13)
17	DTLB Capacity	Buffer type II	Similar to L1D$ Capacity diagnosis (#14). Align memory addresses to page size instead of word size.
18	DTLB Associativity	Buffer type II	Similar to L1D$ Associativity diagnosis (#13). Align memory addresses to page size instead of word size and ensure that they access the same set.
19	L1D$ MSHR Outstanding #	Buffer type II	Execute n concurrent loads that incur L1 misses with different addresses. When n exceeds MSHR capacity, IPC drops as some misses are serialized.
20	L1D$ MSHR Coalescing #	Buffer type II	Similar to L1D$ MSHR Outstanding diagnosis (#19). Execute n concurrent loads accessing the same address instead.
21	I$ Capacity	Buffer type II	Similar to L1D Capacity diagnosis (#11). Execute non-memory instructions instead of loads to fill I$.
22	I$ Block Size	Buffer type II	Similar to L1D Block Size diagnosis (#12). Execute unconditional branches that jump with stride n, instead of loads.
23	I$ Associativity	Buffer type II	Similar to I$ Block Size diagnosis (#13). n branches jump to the next addresses sharing a set.
24	ITLB Capacity	Buffer type II	Similar to I$ Block Size diagnosis (#22). The branches should jump to the next pages. If n exceeds ITLB size, IPC decreases.
25	ITLB Associativity	Buffer type II	Similar to I$ Block Size diagnosis (#12). The branches should jump to the next nth pages that share a set.

(Continued)

Table 1. Continued

#	Diagnosis	Category	Operation detail
26	L1I$ MSHR Outstanding #	Buffer type II	Repeat instructions which take n cache blocks to create n outstanding L1I$ accesses. When n exceeds MSHR capacity, IPC reduces.
27	Functional Unit Latency	Latency	Execute a chain of instructions that use the target functional unit. CPI determines the target functional unit latency plus data forwarding latency used to form the chain.
28	Branch Misprediction Penalty	Latency	Execute a chain of mispredicted branches. CPI is same with branch misprediction penalty.
29	Forward Latency	Latency	Compare the CPI of two instruction streams, dependent instructions and independent instructions, which consist of the same operation. The difference shows data forward latency.
30	I$ Load-to-Use Latency	Latency	Execute a chain of mispredicted branches that fit in L1I$. CPI becomes I$ load-to-use latency plus misprediction penalty.
31	L1D$ Load-to-Use Latency	Latency	Execute a chain of L1D$-hit loads. CPI determines L1D$ load-to-use latency.

Table 2. Details of DiagSim Design (2 of 2)

#	Diagnosis	Category	Operation detail
32	L2D$ Load-to-Use Latency	Latency	Similar to L1D$ Load-to-Use Latency diagnosis (#31). Make the loads hit at L2D$.
33	DTLB Latency	Latency	Similar to L1D$ Load-to-Use Latency diagnosis (#31). Make the loads access distinct pages to miss at DTLB. CPI determines DTLB access/refill latency (+ L1D$, address calculation, data forward latency).
34	ITLB Latency	Latency	Similar to I$ Load-to-Use Latency diagnosis (#30). Execute branches that jump to distinct pages. CPI determines ITLB access/refill latency (+ L1I$, branch mis-prediction penalty).
35	DRAM Load-to-Use Latency	Latency	Similar to L1D$ Load-to-Use Latency diagnosis (#31). Make the loads access main memory.
36	Functional Unit Count/Throughput	Width	Execute independent instructions that use the target function unit. IPC determines the count/throughput of the target unit.
37	Commit Width	Width	Use a group of instructions that can commit together to control commit width utilization. Different groups cannot commit together due to data dependency. IPC becomes maximum when the group size matches the commit width.
38	Writeback Width	Width	Use a group of load instructions that utilize writeback width together; different groups cannot writeback together due to data dependency. IPC becomes maximum when the group size matches the writeback width.
39	Fetch Width	Width	Use an instruction group which ends with a taken branch to control fetch width utilization; the taken branch prevents fetching two groups together. IPC becomes maximum when the group size matches the fetch width.
40	Minimum Width	Width	Repeat nops that reside in the same I$ block. IPC determines the minimum width of pipeline stages.
41	Virtual to Physical Address Mapping	Behavior	Check how the simulator maps virtual address to physical address affects the performance. First, get a physically contiguous page with mmap() and run L2D$ Associativity diagnosis (#16). Next, get arbitrary-ordered pages and run the diagnosis again. If the results differ, the way the simulator maps virtual address to physical address has performance effect.

(Continued)

Table 2. Continued

#	Diagnosis	Category	Operation detail
42	RAW Dep. Check for Reg.	Behavior	Compare the IPC of two streams of instructions. One has independent instructions and the other has instructions with RAW dependencies via register. If the IPCs differ, the target simulator correctly enforces the dependency.
43	RAW Dep. Check for Mem.	Behavior	Compare the IPCs of two streams of instructions. One has independent loads/stores and the other has loads/stores with RAW dependencies via memory. If IPCs differ, the target simulator correctly enforces the dependency.
44	Spec. Memory Disambiguation	Behavior	Compare IPCs of two load/store streams that access different addresses. Address calculation of stores in one stream takes longer than the other's. If two IPCs are similar, we can deduce the target simulator does not perform memory speculative disambiguation.
45	Bpred Stability	Behavior	Check IPC converges to a certain value while repeatedly executing a stream of branches.
46	Bpred Type Check	Behavior	Execute branches with two patterns: one can be handled by a certain type of predictor and the other cannot. If IPC of the pattern that the certain type favors is higher than the other's, we can deduce the predictor is of that type.
47	Bpred Accuracy Check	Behavior	Verify IPC with given branch patterns with known accuracy.
48	Cache Replacement Policy	Behavior	Execute loads with two access patterns: one can be handled by a certain replacement policy and the other cannot. If IPC of the access pattern that the certain policy favors is higher than the other's, we can deduce the cache uses that type of policy.
49	Commit Granularity	Behavior	Check the target simulator commits an instruction only when all micro ops of the instruction complete. Execute two load instructions incurring LLC misses and separate them by ROB-size nop instructions to serialize two LLC misses. If the simulator commits an instruction while its micro ops are partially complete, two LLC misses unexpectedly overlap and IPC would be higher than that of the case where two LLC misses are serialized.
50	D$ Prefetcher	Behavior	Compare the IPCs of two data address patterns. One sequentially accesses memory for a certain type of prefetcher (type-A) to successfully predict, but the other randomly accesses it. If two IPCs differ, the simulator has the type-A predictor.
51	I$ Prefetch	Behavior	Similar to D$ Prefetcher diagnosis (#50). Use instruction address patterns instead of data address patterns.
52	LLC Miss Creation	Condition	Check whether a LLC miss occurs.
53	MLP ≥ 2	Condition	Check loads/stores run in parallel.
54	Unoptimized NOP	Condition	Check whether nops flow through the pipeline.
55	LLC Miss Latency $\gg$ Fetch Latency.	Condition	Check whether a LLC miss incurs larger penalty than instruction fetch.
56	Check whether the diagnosis fits in L1I$	Condition	Check whether the code size of a diagnosis fits in I$.

4.1 Overview of The Process

Figure 6 illustrates the overall steps to utilize DiagSim. First, we examine the diagnosis dependency map, which is a graph indicating the relationship between diagnoses. For example, the A→D dependency tells that diagnosis A should pass (i.e., confirm the correctness of a module/parameter)

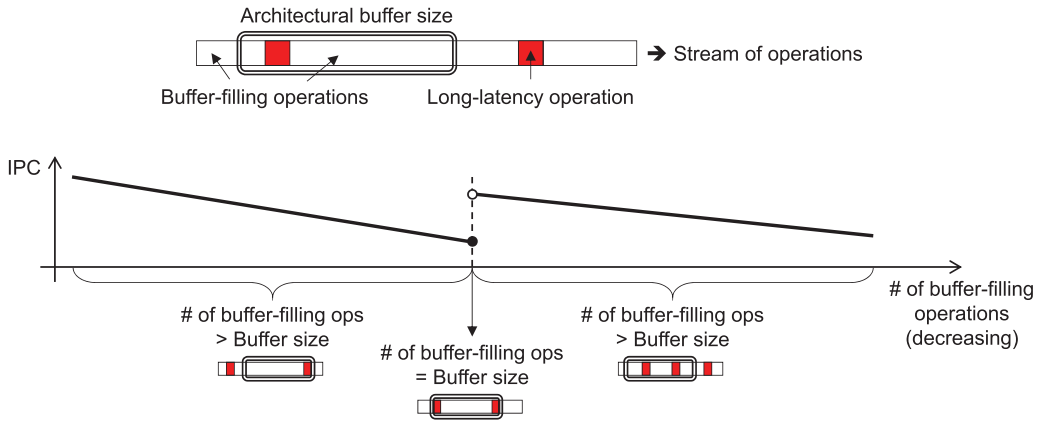


Fig. 8. Diagnosis method for Buffer type 1.

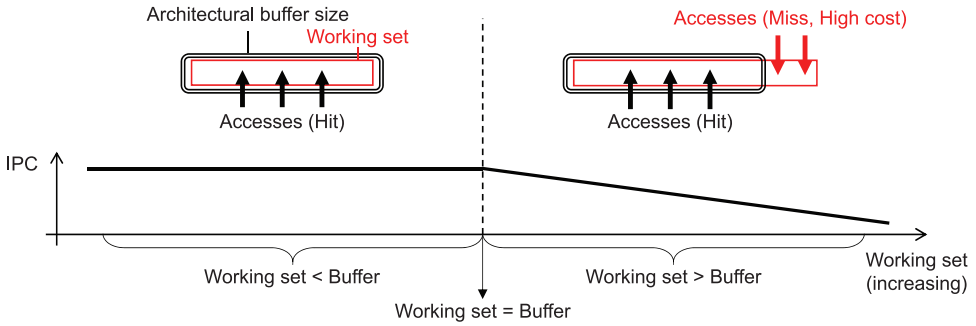


Fig. 9. Diagnosis method for Buffer type 2.

for diagnosis `D` to be functional, because diagnosis `D` is built assuming that diagnosis `A` will pass (i.e., the module/parameter to be correct). In other words, the dependency map informs us which diagnoses should *run first*.

When testing all modules and parameters, we run all the diagnoses starting from the lowest level in the dependency map (i.e., start with the ones without dependency requirements). In case we diagnose only a specific module, we extract the subgraph from the dependency map and run only the related diagnoses. In our example (Figure 6), to run diagnosis `F`, we only need to run diagnoses `A`, `B`, `C`, `D`, and `E`.

In DiagSim, running a diagnosis is to *sweep* a range of potential parameter values with an assembly-based microbenchmark. As illustrated in Figure 6, our approach observes the IPC pattern of the microbenchmark results to detect what the parameter's *actual* value is. Section 4.2 further describes how the microbenchmarks are designed and the patterns can be interpreted. Note that each microbenchmark run is very short so that testing a wide range of values does not incur large overhead.

From diagnosis results, we check whether the target module/parameter matches the expected behavior. If there is a discrepancy between the expected value and the detected value, assuming all the lower level diagnoses passed, there are hidden modeling details or bugs to deal with. The user may further investigate the module to modify the behavior, or simply adjust the simulator configurations to offset the discrepancies (if possible).

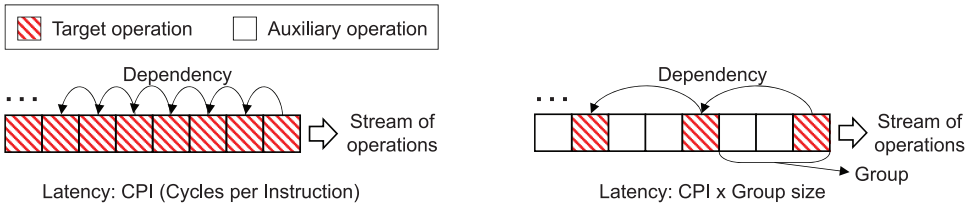


Fig. 10. Diagnosis method for Latency.

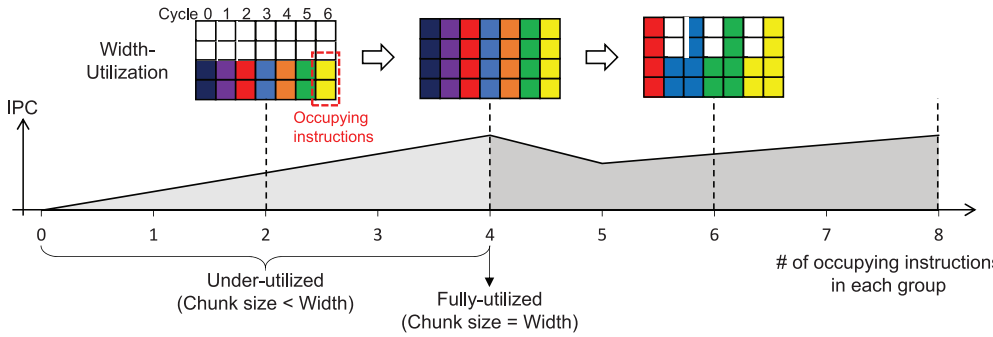


Fig. 11. Diagnosis method for Width.

We now introduce two practical issues in running the diagnosis (Figure 7). First, the diagnosis may *not* exist for some modules either due to a user's choice (i.e., decide not to implement) or the simplicity of the module (i.e., it is better to validate with source code investigation or documentation). We have two options for such cases. First, we may manually inspect the module to guarantee the correctness; this is the recommended option. Second, we may decide to *assume* its correctness and move on to higher level diagnoses. If a higher level diagnosis fails in this scenario, instead of suspecting the module related to the failed diagnosis, we should first inspect the skipped modules and validate their correctness.

The second issue is a cyclic diagnosis dependency which makes the hierarchy ambiguous. For this case, we recommend to manually investigate any (but preferably the simplest) module in the loop to break the cycle.

4.2 Generic Per-category Diagnosis Ideas

To efficiently construct the diagnoses, DiagSim classifies the modules and proposes a generic approach for each category. Table 1 and 2 show the modules and parameters we consider and how they are categorized. We observe that *six categories—buffer type I, buffer type II, latency, width, feature/behavior,* and *condition*—successfully cover the modules. We believe that they embrace currently uncovered modules as well. The following paragraphs elaborate the diagnosis idea for each category. Note that we also provide example codes for each category in Appendix A.

Buffer type I. Buffers are one of the most common structures in a processor. The first type of buffers are *semi-sequential* buffers such as issue queue, reorder buffer, load store queue, and register file. We use the term semi-sequential because these buffers tend to fill and drain a sequential stream of data. In other words, they are similar to queue but sometimes allow out of order insertion and removal.

Figure 8 shows how the size of this type of buffers can be examined. Basically, we put *buffer-filling* operations in between two long-latency operations (e.g., LLC miss), and observe the

Table 3. DiagSim Dependency Map

#	Diagnosis	Depends on	#	Diagnosis	Depends on
1	ROB Size	52, 53, 54, 55, 56	29	Forward Latency	56
2	IQ Size	52, 53, 55, 56	30	I$ Load-to-Use Latency	56
3	LSQ Size	52, 53, 55, 56	31	L1D$ Load-to-Use Latency	11, 56
4	Physical Register Count	52, 53, 55, 56	32	L2D$ Load-to-Use Latency	11, 14, 56
5	BTB Size	56	33	DTLB Latency	17, 56
6	BTB Associativity	56	34	ITLB Latency	24, 56
7	RAS Size	56	35	DRAM Load-to-Use Latency	14, 52, 55, 56
8	Bimodal Bpred Size	46, 56	36	Functional Unit Count/Throughput	56
9	Bimodal Bpred Counter Bit Size	46, 56	37	Commit Width	52, 55, 56
10	Two-level Bpred Counter Bit Size	46, 56	38	Writeback Wdith	52, 53, 55, 56
11	L1D$ Capacity	55, 56	39	Fetch Width	
12	L1D$ Block Size	11, 55, 56	40	Minimum Width	54, 56
13	L1D$ Associativity	11, 55, 56	41	Virtual to Physical Address Mapping	52, 55, 56
14	L2$ Capacity	11, 55, 56	42	RAW Dep. Check for Reg.	56
15	L2$ Block Size	12, 14, 55, 56	43	RAW Dep. Check for Mem.	52, 53, 55, 56
16	L2$ Associativity	13, 14, 55, 56	44	Speculative Memory Disambiguation	52, 53, 55, 56
17	DTLB Capacity	56	45	Bpred Stability	
18	DTLB Associativity	17, 56	46	Bpred Type Check	56
19	L1D$ MSHR Oustanding #	52, 53, 55, 56	47	Bpred Accuracy Check	46
20	L1D$ MSHR Coalescing #	12, 52, 53, 55, 56	48	Cache Replacement Policy	11, 56
21	I$ Capacity		49	Commit Granularity	1
22	I$ Block Size	21	50	D$ Prefetcher	56
23	I$ Associativity	21	51	I$ Prefetcher	
24	ITLB Capacity		52	LLC Miss Creation	14
25	ITLB Associativity	24	53	MLP ≥ 2	36
26	L1I$ MSHR Outstanding #	52, 53, 55, 56	54	Unoptimized NOP	
27	Functional Unit Latency	56	55	LLC Miss latency ≫ Fetch Latency	21, 30, 39
28	Branch Misprediction penalty	21, 56	56	Diagnosis fits in L1I$ Size	21

*Diagnoses using cache hit/miss behaviours depend on cache diagnoses. Diagnoses that may have large instruction/data footprint depend on I$ capacity/D$ capacity and Virtual-to-Physical mapping diagnoses. Lower-level memory hierarchy depends on higher levels (i.e., closer to CPU). A latency diagnosis depends on other latency diagnoses if its latency is a part of other latencies. Prediction unit diagnoses (e.g., branch predictor, prefetcher) inform the related units (e.g., branch, cache) of its existence and operation patterns.

performance sensitivity against the number of buffer-filling operations. If the buffer does not contain both the long-latency operations, reducing the buffer-filling operations simply drops the performance because the portion of long-latency operation becomes higher in the instruction stream. However, as two long-latency operations begin to fit in the buffer, the *performance drop reduces* because the long-latency operations now execute concurrently and (partially) overlap. In other words, the refraction (or discontinuous) point of the IPC–buffer-fill operation count plot shows the buffer size. Table 1 describes what should be the buffer-filling operation for different modules.

Buffer type II. The second type of buffers are *key-value-like storages*. Cache and main memory, miss handling state registers (MSHR), branch target buffer (BTB), return address stack (RAS), branch history buffer, and many others belong to this category. These buffers simply return the previously stored data for a given key (e.g., address, hash value, location determined by a logic).

To examine these buffers, we investigate the performance sensitivity against the working set (i.e., actively accessed region) size. As shown in Figure 9, the performance stays stable if the working set fits in the buffer, but starts to change if we access outside the buffer; usually, the performance

degrades because such an access is considered a miss. We leverage this nature to identify the size, associativity, and boundaries (e.g., bank, channel, transfer granularity) of various buffers. The key idea is to gradually change the working set size (or pattern, for associativity and boundary diagnosis) and observe the point where the performance changes (i.e., the boundary crossing happens). Figure 9 shows an example for the size. Table 1 summarizes which operations are used to form the working set for different buffers.

Latency. Figure 10 shows the method to test the operation latency of modules. We form a chain of dependent instructions where the instruction has the target operation latency. Since this chain becomes the critical path and determines the overall pipeline flow rate, we can deduce the operation latency from the performance.

Assuming an ideal case, we may form the chain using only the target operations (Figure 10, left). In this case, the cycles per instruction (CPI) equals to the operation latency. In practice, we often need auxiliary operations (e.g., conditionals to terminate the diagnosis; Figure 10, right). Assuming that the target operation has a long latency and therefore becomes the bottleneck of the pipeline flow, we can deduce that a group of operations (i.e., a set of target and auxiliary operations) retires every target operation latency. The latency is therefore the cycles per instructions multiplied by the group size.

Width. To measure the width of a module, we leverage the correlation between the performance and width utilization. Figure 11 visualizes the testing method. We flow *groups* of instructions through the pipeline to occupy the width as we wish. We control the dependencies between the instructions so that the instructions in a group execute concurrently but different groups do not run together. We detect the width by changing the group size (i.e., number of instructions in a group) and observing the corresponding performance. The maximum performance appears if the group size matches the width, because it maximizes the utilization. Further increasing the group size initially decreases the performance because the width utilization drops, but the performance reaches the maximum point again if the group size becomes multiples of the width.

Note that by setting an instruction to a specific type, we can measure other (band)widths as well. For example, by setting the instructions to always access DRAM (i.e., LLC miss), we can measure *effective* DRAM access throughput and hence the bandwidth (limited by LLC's MSHR or DRAM bandwidth). Section 5.4 explains the implication of the word *effective* in detail.

Feature/Behavior. Testing the feature or behavior of a module requires familiarity with the module's operation logic. The basic idea is to generate two instruction patterns—one that triggers the module logic and the other that does not trigger the logic—and observe whether there is a significant performance difference. If the module has the feature or behavior, the performance difference would be large. Otherwise, the two patterns would have negligible performance difference (i.e., we should design the patterns in this way).

For example, prefetcher diagnosis consists of memory access patterns such as random access and N-byte stride access. As random access keeps the confidence counter of any prefetchers low, we obtain almost identical IPC values regardless of what prefetcher (and even no prefetcher) is used. On the other hand, with N-byte stride patterns, any prefetchers yield significantly better IPC values than before, while the no-prefetcher case is unchanged. We can further differentiate various prefetchers by testing more complex memory access patterns.

As seen in the example, this category requires manual effort to implement the diagnosis case by case. However, we provide a few templates which can test popular features and behaviors such as branch prediction, cache replacement policy, and prefetcher. The users may expand these diagnoses to cover a wider set of behaviors.

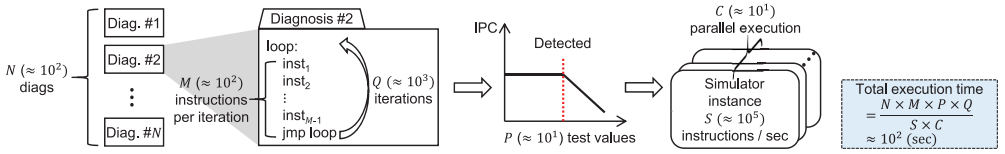


Fig. 12. DiagSim execution process and overhead.

Conditions. This category is a collection of *artificial* diagnoses which we define to tidy up the hierarchical/systematic diagnosis process. We observe that many diagnoses assume the correctness of simple architectural conditions (e.g., able to create LLC miss) to function correctly. To make the whole diagnosis process systematic, we consider these simple conditions as *diagnoses* and include them in the dependency map. It allows us to explain all the prerequisites using a single dependency map to keep the overall process clear and concise. We note that many of the diagnoses in this category are special conditions of the existing diagnoses or simple fact checks, and thus can be handled with negligible overheads (even without actual microbenchmarks to verify them).

4.3 Diagnosis Dependency Map

The diagnosis ideas proposed in the previous section in fact assume the integrity of the other modules to correctly operate, because they utilize their parameter values and behaviors to trigger certain operations. For example, to utilize LLC miss events as we wish, we should know the exact LLC capacity to trigger a miss at the right point. We therefore construct a *diagnosis dependency map* to thoroughly track such requirements and ensure a diagnosis does not fail due to external factors (i.e., other than the diagnosis itself).

Table 3 shows the dependency map for DiagSim. For the 56 diagnoses, we mark which other diagnoses must pass (i.e., ensure the correctness of a module/parameter) to guarantee the functionality of *the* diagnosis. A diagnosis usually depends on another module/parameter (and the diagnosis which validates it) if it uses the characteristics of the module/parameter to trigger specific events (e.g., LLC miss) or it is tightly coupled to the module/parameter itself (e.g., cache associativity and capacity). Note that the table specifies only the direct dependency between two diagnoses. To reveal all the dependencies related to a diagnosis, we should track it until we reach a diagnosis that depends on no others (i.e., leaf nodes in the graph), or we re-visit a diagnosis (i.e., forming a loop). For example, the cache replacement policy diagnosis (#48) has two direct dependencies (#11 and #56), which lead to further dependencies (#21, #55). We note that the maximum depth of the dependency is not very large (#49→#1→#52→#14→#11→#55→#30→#56→#21 in the current implementation) and the overall diagnosis overhead is manageable.

4.4 DiagSim Execution Overhead

In this section, we analyze the overhead of running the diagnoses. Figure 12 illustrates a typical DiagSim execution overhead. First, we assume running $N \approx 10^2$ diagnoses, which is about twice as many as the current diagnoses (Table 1). We make a conservative estimation as simulators are continuously adding modules to cover newer architectures, and the number of diagnoses to cover the modules would grow accordingly. Next, for each diagnosis, we have a loop of $M \approx 10^2$ instructions. As illustrated in Section 4.2, our diagnoses consist of relatively simple operations and therefore can be represented with few dozens of instructions. We then run the loop $Q \approx 10^3$ times to achieve the steady-state behavior. We empirically confirm that 1000-iteration is sufficient and set it as the hard limit. To sweep the parameter space, we would run such a diagnosis for $P \approx 10$ different values. To minimize the number of values to test, we first sweep a wide range with coarser intervals and focus on a narrower range with finer-grain values.

Putting the numbers together, we execute total $N \times M \times P \times Q = 10^8$ instructions to run *all* the diagnoses; running a specific diagnosis would reduce the overhead by an order. Running all diagnoses would take about 10^3 seconds (i.e., <1 hour) on a cycle-level simulator with 100 KIPS (i.e., 10^5 instructions per second) speed. In addition, we may further boost the whole process by running the independent diagnoses in parallel; the overhead would decrease to several minutes in this case. We emphasize that the diagnosis overhead is much more affordable compared to the performance evaluation simulations running billions of instructions.

4.5 Implementation Efforts

In this section, we introduce our efforts to realize the diagnosis ideas proposed in the previous sections. Currently, we implement DiagSim as x86-64 ISA assembly-based microbenchmarks. Appendix A illustrates example microbenchmark codes. Porting DiagSim to the other ISAs should be straightforward since we provide generic insights behind the diagnosis designs (Section 4.2) as well as the following implementation efforts.

First, we extensively leverage `nop` to generate various patterns. `nop`s allow us to align instructions to specific addresses, just as compilers. It also occupies pipeline (i.e., ROB) without dependencies and register consumptions. We may therefore insert an arbitrary amount of padding between the instructions of interest. For architectures crushing `nop` at issue stage, we may use equivalents (e.g., `or %reg1, %reg1`).

Second, we frequently use long-latency instructions to generate back-pressure to the pipeline. The most reliable way is to incur LLC misses; this is why many of the diagnoses depend on cache-related diagnoses. We generate an LLC miss using a series of three instructions: `lea` (i.e., load effective address), `and`, and `mov`. While simple random accesses may suffice, we decide to access a buffer larger than the LLC capacity to ensure that we *always* incur an LLC miss. The `lea` and `and` set up the address to access and `mov` triggers the LLC miss. The address calculation ensures that we perform the accesses within a given address range (otherwise, it would cause a segmentation fault). In addition, if access randomization is required, we put `mul` before the `lea` to implement linear congruential generator [27] and generate pseudo-random access sequences.

Third, we make a diagnosis binary for each parameter value to test, rather than taking the parameter value as an argument to the assembly program (i.e., diagnosis). In other words, if there are 10 parameters to test, we generate and run 10 different binaries. This allows us to eliminate unnecessary setup codes which can introduce noises to the IPC behaviors of the diagnoses.

Lastly, we make sure to measure the steady-state behavior by repeating the same diagnosis multiple times. This eventually introduces a loop (i.e., `jmp` instruction). For example, consider ROB size diagnosis which constitutes a loop of LLC misses followed by many `nop`s and finally `jmp`. If we decrease the number of `nop`s from a large number, at a certain point the ROB starts to capture two LLC misses (Figure 8). Due to instructions other than `nop`s, the number of `nop`s is not the ROB size; we need to offset the number of auxiliary instructions such as `lea`, `and`, `mov`, and `jmp`.

4.6 Adding a New Diagnosis

Although DiagSim covers major performance-critical modules and parameters, users may need to implement their own diagnoses to ensure the correctness of new/customized modules and parameters. This section describes the general steps to implement a new diagnosis.

First, users identify which category the diagnosis belongs to. If the diagnosis validates a behavior/feature, it is classified as the behavior/feature category. Otherwise, the diagnosis should be validating the module's parameter values. Depending on the type of the parameter (e.g., latency,

Table 4. Baseline Architecture Configuration

Module	Features	Values	Module	Features	Values	
					# Units	Latency (cycles)
Pipeline	Fetch width	6	Functional Units	Effective address calculation units	2	2
	Issue width	6		Int simple ALU	6	1
	Width of other stages	4		FP simple ALU	6	2
	IQ size	36		Int multiplication	1	3
	LSQ size	80 (48 for LQ / 32 for SQ)		Int division	1	15
				FP multiplication	1	4
	ROB size	128		FP division	1	15
	Physical register count	128 (Int & FP)		FP complex op	1	20
Branch Predictor	Type: Tournamant Local table: 1K entries, 8-bit history, 2-bit counter Global table: 1K entries, 2-bit counter Choice table: 1K entries, 2-bit counter		Memory System	L1D Cache	32KB 8-way 64B block 2 MSHR entries 4 cycles latency	
	BTB	256 entries, 4-way		L1I Cache	32KB 4-way 64B block 2 MSHR entries 3 cycles latency	
	RAS	32 entries		L2 Cache	2MB 8-way 64B block 16 MSHR entries 12 cycles latency	
	Branch misprediction penalty	2 cycles		Memory	150 cycles latency	

capacity, width), the diagnosis can be classified accordingly. Lastly, the buffer type (if the diagnosis is about a buffer) can be distinguished by the module's operation logic; ROB-like semi-sequential buffers fall into type I and key-value-like buffers fall into type II, as discussed in Section 4.2.

Second, users implement the diagnosis based on the category's generic ideas. We stongly recommend to modify the existing diagnoses of the same category to ease the task.

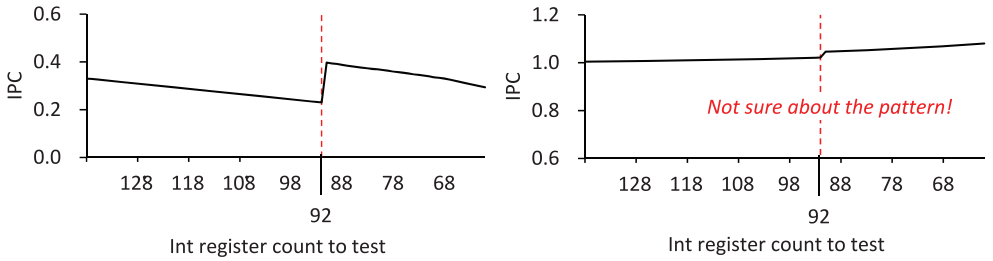
Lastly, users update the dependency map to incorporate the new diagnosis. Dependencies appear if a module's diagnosis assumes and leverages the correctness of the other modules. Some dependencies are obvious (e.g., to utilize LLC miss behavior, we should know the correct LLC capacity to generate misses), while some others are not (e.g., cache hierarchy). To avoid omitting these dependencies, we can perform sensitivity tests on a new diagnosis. For example, if changing the parameter of module B affects module A's diagnosis outcomes, we would know that module A's diagnosis depends on module B's.

5 EVALUATION

5.1 Experimental Setup

We use three popular open source x86 ISA cycle-level full-system simulators to evaluate DiagSim. Table 4 describes the baseline architecture configuration.

We perform three case studies to show the effectiveness of DiagSim. First, we perform *basic diagnosis* for the three simulators modeling the baseline architecture. We aim to identify any discrepancies between the specified architecture configuration and the actual architecture behavior. Second, we modify the simulators to account for hidden factors and *eliminate the discrepancies*



(a) Diagnosis *after* running prerequisite diagnoses. (b) Diagnosis *without* prerequisite diagnoses.

Fig. 13. Examples of diagnosing gem5's int register file size.

detected in the first step, and show the corresponding *performance impact*. Third, we change the baseline architecture and run DiagSim again to check whether there are *extra hidden factors* that were not accounted for in the first step.

Throughout the case studies, we use SPEC CPU 2006 workloads to consider a wide range of application behaviors. We present *all* the workloads that run on each simulator. We use SimPoint to select and simulate representative 100M instructions for each workload. Note that there are some workloads that are not presented for all three simulators, because they fail during checkpointing or simulation phases.

5.2 Case Study I: Basic Diagnosis

In this step, we run DiagSim for the three simulators modeling the baseline architecture. To detect all possible discrepancies, we run all the diagnoses following the dependency map.

First, we illustrate the importance of diagnosis following the dependency map. Figure 13 illustrates an example of detecting gem5's int register file size. Using the diagnosis, we sweep the range around the expected parameter value (128) and observe the IPC pattern to detect the actual value. Typically, the deflection point tells what the detected value is (detailed IPC behaviors to observe are described in Table 1 and 2). Figure 13(a) shows that we get clear and accurate diagnosis results (92) by faithfully following the dependency map. In this case, the prerequisite was to check the LLC capacity to correctly generate long-latency LLC misses. In Figure 13(b), we ignore the LLC capacity diagnosis and naively perform long distance memory accesses, expecting to get LLC misses. This time, the diagnosis results are *not clear* because we generate a mix of long/short latency operations and the int register diagnosis' functionality breaks. Note that through this example, we successfully detect that gem5 has smaller int register count (92) than expected (128).

Figure 14 shows the simulator diagnosis results illustrated as dependency graphs. For brevity, we show only the modules with discrepancy and the lower-level diagnoses required to test the modules. We observe that many of the modules *behave differently* from the expectations. For the cases where a lower level diagnosis shows discrepancies, we first modify the module to assure the expected behavior before moving on to the higher levels. This allows us to *significantly* reduce the diagnosis overhead. For example, if there were no such systematic orders, we would have run diagnosis #35 first and make adjustments to resolve the discrepancies, only to find that the other modules (diagnoses #31 and #32) affect diagnosis #35 and the previous adjustment was in fact incomplete.

Table 5 summarizes the discrepancies of the three simulators. We further investigate the simulators to identify the root causes as follows.

gem5. The physical register file is smaller than expected because the general and flag registers occupy a portion in the register file. BTB size is also different from the expectation due to alignment

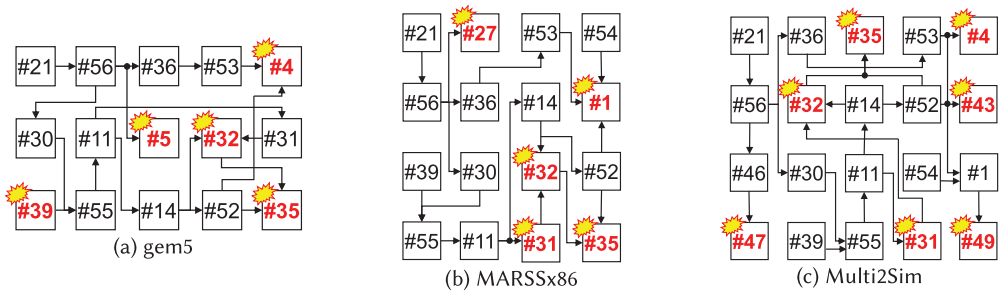


Fig. 14. Using dependency map to systematically diagnose simulators. We show only the diagnoses related to performance discrepancies.

Table 5. Discrepancies Between the Baseline Design and Actual Behavior

gem5		
Diagnosis	Expected	Detected
# Physical registers	128	92 (Int) 75 (FP)
BTB size	1024	2048
L2D$ latency	12	22
DRAM latency	150	193
Fetch width	6	4
DRAM bandwidth	4.3 GB/s	

MARSSx86		
Diagnosis	Expected	Detected
ROB size	128	127
IntAdd latency	1	2
IntMul latency	3	5
L1D$ latency	4	8
L2D$ latency	12	20
DRAM latency	150	160
DRAM bandwidth	17.4 GB/s	

Multi2Sim		
Diagnosis	Expected	Detected
# Physical registers	128	117
L1D$ latency	4	6
L2D$ latency	12	22
DRAM latency	150	193
Commit granularity	6	4
Load-store behavior	Discrepancies in the operation logic	
BTB behavior		
DRAM bandwidth	3.1 GB/s	

*Latency is in cycles. Memory system latency indicates the load-to-use latency.

issues; the entries have the low 2-bits of the addresses truncated and allow four different branch PCs to land on one entry (i.e., similar to 4-way set associative cache). The load-to-use latency of L2 data cache and DRAM are much higher than expected because there are hidden models related to message responses and evictions.[2] The fetch width is smaller due to the implicit skid buffer modeling.[3]

MARSSx86. The ROB size is smaller because the circular queue implementation loses one entry. The integer add and multiply latencies are higher because the back-to-back wakeup mechanism has an additional latency due to implementation issues. The memory latencies are higher due to hidden modeling factors similar to gem5.

Multi2Sim. The register file is smaller than expected because architectural registers are counted within the physical registers. The memory latencies are higher due to the similar reasons to the other two simulators. In addition, we observe that certain logics to check dependency conditions are not rigorous (i.e., macro-op commit, load-store address disambiguation, BTB entry update), yielding higher performance than expected.

Fortunately, we find that the discrepancies are generally straightforward to fix. We (1) modify the architecture configuration file to create offsets and match the expected parameters, (2) fix the

[2]We assume that a user considers the cache and DRAM latency parameters in simulator configurations as the load-to-use latencies. This is valid for many simulators with simplified memory systems, but our simulators have detailed interconnection models which add a *hidden* amount of extra latencies to the configured values. Since a simple inspection on simulator configuration cannot calculate this extra latency, we define it as a discrepancy and use DiagSim to measure its amount.
[3]Skid buffer holds the pending instructions from the current cycle and blocks the previous stage until the buffered instructions drain.

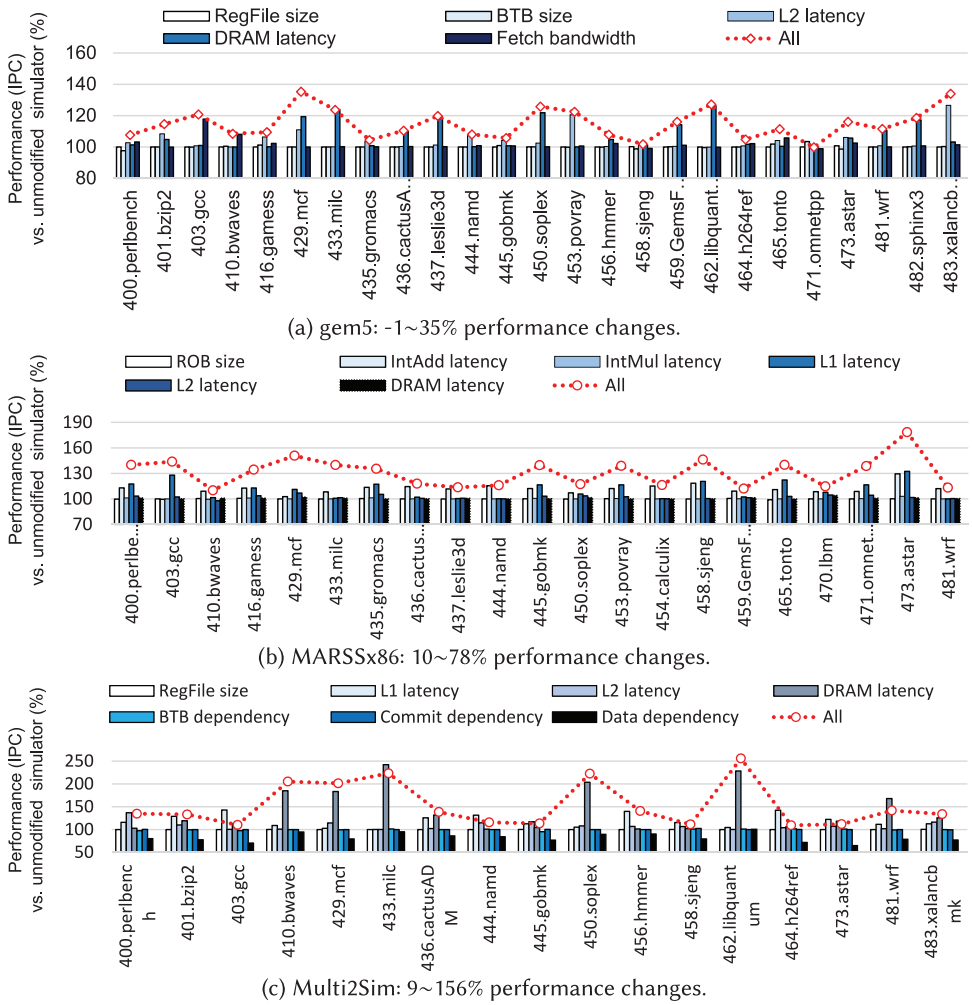


Fig. 15. Performance changes upon eliminating the discrepancies.

parameters in the source code to match the expected parameters, and (3) fix the logic in the source code to enforce expected behaviors. Note that without the help from DiagSim, it would have been challenging to detect and account for these small but performance-critical details.

5.3 Case Study II: Performance Impact of Accounting for the Hidden Factors

We now describe how the performance changes *after* addressing the issues diagnosed in Section 5.2. For each simulator, we eliminate the discrepancies and compare the new performance with the unmodified simulators' performance.

Figure 15 shows the performance change from fixing a single discrepancy (bars) and all the discrepancies (dashed line). We find that all three simulators have noticeable performance changes after all corrections, and eliminating a single discrepancy is *not enough* to reach the true performance. This emphasizes that to achieve the best possible results, we should employ DiagSim to *thoroughly* check and resolve all the discrepancies.

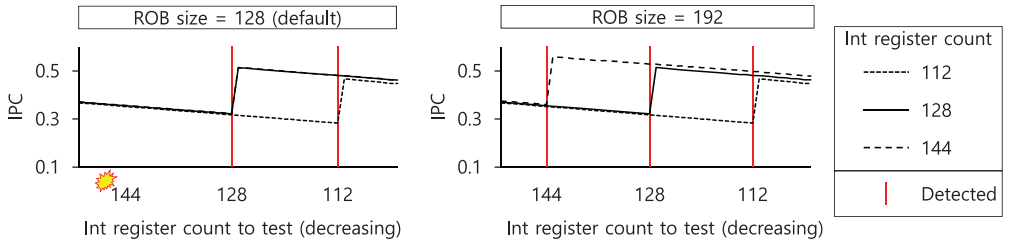


Fig. 16. Diagnosing int register file size for two different architectures (ROB size 128 and 192). For the small ROB architecture, the diagnosis detects the largest register file size (144) as ROB size (128) because it is impossible to utilize the register file past the ROB size. For the large ROB case, all sizes are correctly detected.

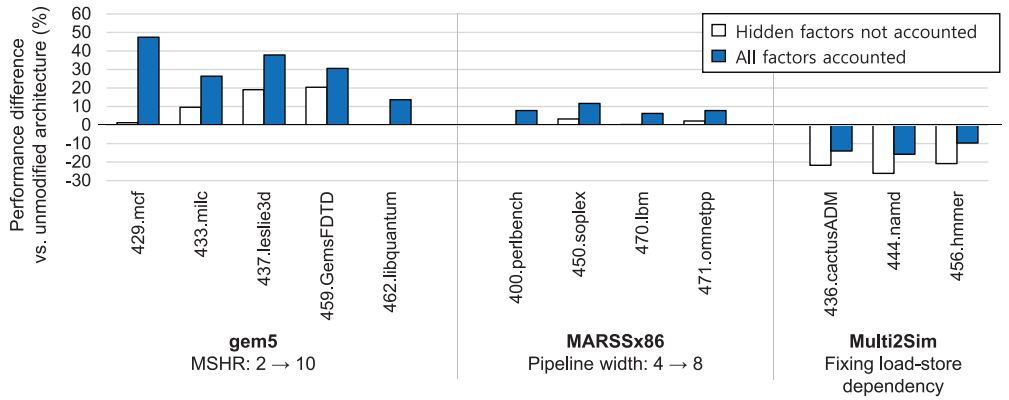


Fig. 17. Performance impact of considering extra hidden factors which appear upon architecture changes.

5.4 Case Study III: Diagnosis Upon Architecture Modifications

Lastly, we demonstrate how DiagSim always ensures healthy simulations for various architectural changes. In this scenario, we first address all the discrepancies detected in Section 5.2. We then change the architecture by modifying the module parameters, and run DiagSim again to ensure the correctness.

Surprisingly, we discover that *extra* hidden factors and issues start to appear as we modify the architecture. Figure 16 explains why such factors may exist. In the example, we observe that an int register file's size can be measured *up to* the ROB size, because it is impossible to utilize the register file past that size. Note that this is a normal and intended behavior as DiagSim detects the *effective/actual* parameter value. Therefore, if ROB size is maintained small, we would ignore the potential issues in int register file size as we cannot test the value larger than the ROB size (but consider it normal). For example, even if there is a hard-coded limit of 128 in the int register file size, it would remain hidden until we explore designs with ROB size and register file size >128. We now introduce such extra factors discovered for each simulator.

gem5. We notice that gem5 has a fixed limit in the writeback bandwidth. We find this issue upon increasing the MSHR size from 2 (default) to 10. For the baseline, the writeback bandwidth is (relatively) large enough so that it does not appear in the diagnosis. Increasing the bandwidth to match the MSHR size effectively improved the overall performance, as illustrated in Figure 17.

MARSSx86. We increase the pipeline width and functional unit counts to model a powerful core design (default: 4-way execution; new: 8-way execution). Strangely, we observe only 1.5% improvement in the performance (for SPEC). Running DiagSim reveals that the effective pipeline width is *still 4*, so we further investigate the execution stage. We eventually find out that the maximum pipeline width and functional unit counts are hard-coded. Fixing this issue improved the average IPC gain from 1.5% to 8.4%.

Multi2Sim. Multi2Sim has issues with load-store address dependency resolution (discussed in Section 5.2). We therefore fix the resolution logic to correctly enforce all the dependencies; this results in 22.9% degradation in performance. Without DiagSim, we would naively believe that such a performance degradation is normal as we have added more restrictions. However, running DiagSim again reveals that we actually enforced *too strict* dependency checks and the performance is underestimated. Relaxing this incorrectly enforced condition alleviated the performance degradation from 22.9% to 13.2%.

As shown in the examples, DiagSim allows us to constantly enforce healthy simulation for various architecture configurations. It detects previously hidden modeling issues as well as newly introduced user-induced errors (e.g., Multi2Sim). We therefore recommend to run DiagSim for every architecture modification.

6 DISCUSSION

Expansion to new modules. In this article, we focus on the modules which have first-order performance impacts (discussed in Section 3.2). We *open source* DiagSim so that the simulator user community may expand the diagnoses to cover new modules, following the steps in Section 4.6. Although implementing new diagnoses would require extra efforts, the developers of the new modules are the best people to understand the details and develop accurate diagnoses (i.e., diagnosis development would be much more difficult for ordinary users). It would ensure the future users of the module to easily verify the correctness of simulations.

Interpreting the DiagSim results. DiagSim requires manual efforts to observe the IPC trend, find the deflection points, and determine the detected value. We believe that this procedure can be automated with the help of a machine-learning technique in most cases. For ambiguous patterns, we can request a manual user inspection.

Expansion to other architectures. We develop DiagSim focusing on out-of-order architectures because the timing model of in-order processor is simpler and thus less subject to the modeling errors than out-of-order processors. However, it is possible for DiagSim to support in-order (or other) microarchitecture with some effort.

As in-order architectures (e.g., low-power processors, GPU cores) have generally different performance dynamics compared to the out-of-order architectures, we suggest to provide an independent diagnoses rather than configuring and reusing the existing diagnoses. For example, in-order processor diagnoses do not need to model out-of-order issue logics (e.g., register renaming, IQ, LSQ, complex ROB) and the list of diagnoses will be reduced. In addition, as the performance of in-order architectures are largely determined by individual instruction latencies (compared to out-of-order which more freely overlap latencies), modeling each diagnosis would be more straightforward.

As our next work, we are further developing DiagSim to handle various heterogeneous core architectures consisting of out-of-order and in-order processors.

Simulator modification affecting DiagSim. We emphasize that major modifications to a simulator such as introducing a new module may affect diagnoses themselves, but DiagSim would

correctly detect such changes and inform the users. For example, if a new module significantly affects the existing diagnoses (e.g., new D$ prefetcher interrupting cache related diagnoses), the dependency map will detect such events (e.g., prefetcher diagnoses) and tell which diagnoses (e.g., D$ related) get affected to inform the users. In our case, after detecting such effects, we added address randomization to D$ diagnoses to ensure a prefetcher does not affect the results.

Expansion to multicores. Currently, DiagSim targets a single core design (including a single core in a multicore processor) because we find enough performance discrepancies. We are expanding our work to cover multicore designs as well. Specifically, we are working on the coherency protocols and synchronization issues (e.g., atomic instructions), as they have large performance impact in general.

DiagSim for real machines. Theoretically, it is possible to run DiagSim for a real hardware if the hardware supports running bare-metal assembly codes. Nonetheless, we add a caveat that some of the diagnoses would be difficult to run or validate. For example, we must ensure whether the *condition* diagnoses pass or fail because we cannot manually investigate real hardware to check whether certain properties are met. In addition, we would need high-precision performance counters telling the exact number of instructions and cycles because we rely on IPC patterns to perform diagnoses.

7 RELATED WORK

Formal verification. Formal verification (FV) builds an abstract mathematical model (finite state machine) of the target system, and mathematically proves that the target system does not fall into erroneous states. Given the model faithfully describes the target system, FV rigorously guarantees no errors. However, building the abstract mathematical model is very complex and hard even for a very small system. Together with the simulation's primary goal, fast evaluation, the applicability of FV is usually constrained within some modules [34]. The examples include Intel's method to verify Pentium 4's floating point units [5], and Biere et al.'s [6] method to check a set of safety properties of a PowerPC processor. Unlike FV which focuses on formal correctness of a module, our work reveals hidden details and inter-module interactions of simulators in holistic ways.

Coverage-driven verification. Coverage-driven verification (CDV) is an alternative to slow formal verification for detecting real-hardware functional bugs. CDV uses statistical methods to generate test cases. The test cases exercise many different data paths to detect any possible errors. For this reason, CDV requires many simulations but can guarantee nearly error-free designs. As notable examples, Intel applied a variation of this verification approach to the RTL design of a real microprocessor [17]. Benjamin et al. also applied coverage-driven verification to the RTL design of another microprocessor[4] rather than a high level model in architectural simulators. Contrary to CDV, architectural simulators are more concerned with fast delivery of performance results than error-free design. In this regard, simulators favor abstracting out some unnecessary details and are not thoroughly examined to eliminate rare errors. These rare errors normally do not affect the performance statistics of the simulator. However, if the errors affect the timing of usual events, DiagSim can detect it.

Simulator validation/evaluation using (micro)benchmarks. Many prior works use benchmarks or microbenchmarks (synthetic workloads) to validate the simulator against a reference processor model. For example, Gutierrez et al. and Butko et al. [12, 18] evaluated the accuracy of gem5 simulator with SPEC and PARSEC benchmarks to model the existing multi-core systems, and Saidi et al. [31] did a similar analysis on the M5 simulator, but focused on the network

workloads. Bose [10] defined possible failure modes on a RISC processor similar to POWER and derived microbenchmarks to cover those failure cases. By comparing the CPI of microbenchmarks on the simulator and the expected CPI, Bose could discover the error of the simulator. Moudgil et al. [24] calibrated the Turandot simulator against the reference model. They used microbenchmarks to tune the simulator until the performance of running SPECInt across the simulator and the reference model becomes very close. Desikan et al. [13] designed a set of microbenchmarks to stress different stages of the pipeline and found the mismatches between a simulator and an Alpha 21264 processor.

These prior works try to validate or calibrate a single simulator model against the reference model. However, we provide architects with a systematic approach to apply generic diagnoses to different simulators and find the hidden details and complex interactions between their components. DiagSim eventually facilitates getting stable results on various simulators.

Recently, Wagstaff et al. [35] proposed a new set of microbenchmarks to mainly evaluate the execution speed of the full-system simulators. This work is orthogonal to ours as we focus on simulator anomalies rather than its execution speed.

Reverse-engineering of the real hardware. A group of synthetic workloads have been proposed to reverse-engineer certain features of a real system [1, 23, 33, 36]. We can adopt their ideas to add more diagnoses to the DiagSim framework. It is not appropriate to use their diagnoses implementations directly, as their diagnoses usually require a large number of instructions to overcome noises. Thanks to the fully controlled simulation environment, we can simplify the implementation to get efficient diagnoses.

Synthetic workloads as the real workload miniatures. Researchers have proposed a group of synthetic workloads as easy-to-run short programs which exhibit similar characteristics of a target real workload. These real workload miniatures are usually generated by extracting representative patterns of the respective real workload. For example, [3] provides synthetic workloads that mimic SPEC CPU 2000 and STREAM workloads, [26] for big-data workloads, and [15] for SPEC CPU 2006, and ImplantBench. DiagSim also depends on a set of short synthetic workloads. However, DiagSim workloads (*diagnoses* in our term) have different purposes and they do not require a reference from a real workload. We build diagnoses based on generic per-parameter category ideas (Section 4.2). During diagnosis simulation, we also do not directly use the generated IPC value; instead, we use the IPC trend to detect a specific point that reveals the effective value for the parameter of interest.

Synthetic workloads as stress generators. Several studies have proposed frameworks to generate synthetic workloads to stress certain features of a system [16, 20, 21]. These workloads iteratively stress the system and check the properties such as power consumption, until they reach a maximum value. DiagSim can also adopt the idea of the synthetic workloads that stress a specific module. However, DiagSim does not use the stress generation as a means to determine the maximum value of a performance metric like IPC or power consumption. Instead, DiagSim uses the performance trend to detect the effective value of the parameter of interest in a simulator.

8 CONCLUSION

In this article, we proposed DiagSim, an efficient and systematic method to diagnose simulators and eliminate behavioral discrepancies. We proposed generic methods to develop diagnoses to make DiagSim highly portable and expandable. We diagnosed three popular open source simulators and reported hidden discrepancies incurring large performance deviations. We demonstrated that DiagSim correctly detects and eliminates the issues to ultimately promote healthy simulations.

A APPENDIX: DIAGNOSIS IMPLEMENTATION

This section presents a code example for each generic per-category diagnosis idea. The codes follow the syntax of GNU Assembler (GAS) x86 assembly language.

```
MainLoop:
   leal   (L2_BLOCK_SIZE)(%eax), %ebx

   movl   Array(%eax), %ecx   # First long-latency
                              # operation

   .rept   (ROB_SIZE)   # The speculated size of ROB
      nop               # Buffer-filling operations,
   .endr                # which try to prevent the
                        # next long-latency operation
                        # from being executed
                        # concurrently.

   movl   Array(%ebx), %ecx   # Second long-latency
                              # operation

   leal   (L2_BLOCK_SIZE)(%ebx, %ecx), %eax
   andl   $(ARRAY_SIZE - 1), %eax
   jmp   MainLoop
```

(a) Buffer type I: ROB size.

```
jmp   .+2                  # A chain of 2-bytes
   .rept   (BTB_SIZE-2)/4  # conditional branches
      jmp   .+4            # occupying contiguous
      jmp   .-4            # address space region.
   .endr
jmp   .+5
nop                        # One byte padding for
jmp   .-5                  # accessing both odd and
   .rept   (BTB_SIZE-2)/4  # even memory address
      jmp   .+4
      jmp   .-4
   .endr
jmp   .-2
```

(b) Buffer type II: BTB size.

```
# Pattern 1: RAW Dependency
MainLoop:
   .rept (ROB_SIZE >> 1)
      movl   %eax, %ebx   # adjacent instructions
      movl   %ebx, %ecx   # have RAW dependency
      movl   %ecx, %edx
      movl   %edx, %eax
   .endr
   jmp   MainLoop
```

```
# Pattern 2: No Dependency
MainLoop:
   .rept (ROB_SIZE >> 1)
      movl   %eax, %eax   # adjacent instructions
      movl   %ebx, %ebx   # have no dependency
      movl   %ecx, %ecx
      movl   %edx, %edx
   .endr
   jmp   MainLoop
```

(c) Behavior: RAW dependency check (Register).

```
MainLoop:
   inc   %ecx       # A target operation to
                    # measure latency
   jmp   MainLoop   # An auxiliary operation for
                    # maintaining loop
```

(d) Latency: Functional unit latency.

```
MainLoop:
   movl   Array(%eax), %edx   # long-latency
                              # load instruction
                              # which leads a chunk

   .rept   (WB_WIDTH)            # a number of
      movl   Array(%ebx), %edx   # short-latency
   .endr                         # load instructions

   leal   (L2_BLOCK_SIZE * 2)(%eax), %eax
   andl   $(ARRAY_SIZE - 1), %eax

   leal   (L2_BLOCK_SIZE * 2)(%ebx), %ebx
   andl   $(ARRAY_SIZE - 1), %ebx
   jmp   MainLoop
```

(e) Width: Writeback width.

REFERENCES

[1] A. Abel and J. Reineke. 2014. Reverse engineering of cache replacement policies in Intel microprocessors and their evaluation. In *Proceedings of the 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'14)*. 141–142. DOI: https://doi.org/10.1109/ISPASS.2014.6844475

[2] Ehsan K. Ardestani and Jose Renau. 2013. ESESC: A fast multicore simulator using time-based sampling. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA'13)*. IEEE Computer Society, 448–459. DOI: https://doi.org/10.1109/HPCA.2013.6522340

[3] Robert H. Bell, Jr. and Lizy K. John. 2005. Improved automatic testcase synthesis for performance model validation. In *Proceedings of the 19th Annual International Conference on Supercomputing (ICS'05)*. ACM, New York, 111–120. DOI: https://doi.org/10.1145/1088149.1088164

[4] Mike Benjamin, Daniel Geist, Alan Hartman, Yaron Wolfsthal, Gerard Mas, and Ralph Smeets. 1999. A study in coverage-driven test generation. In *Proceedings of the 36th Design Automation Conference*. IEEE, 970–975.

[5] Bob Bentley. 2001. Validating the Intel (R) Pentium (R) 4 microprocessor. In *Proceedings of the Design Automation Conference*. IEEE, 244–248.

[6] Armin Biere, Edmund Clarke, Richard Raimi, and Yunshan Zhu. 1999. Verifying safety properties of a PowerPC-microprocessor using symbolic model checking without BDDs. In *Proceedings of the 11th International Conference on Computer Aided Verification*. Springer, 60–71.

[7] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. DOI: https://doi.org/10.1145/2024716.2024718

[8] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. 2006. The M5 simulator: Modeling networked systems. *IEEE Micro* 26, 4 (July 2006), 52–60. DOI: https://doi.org/10.1109/MM.2006.82

[9] Bryan Black and John Paul Shen. 1998. Calibration of microprocessor performance models. *Computer* 31, 5 (May 1998), 59–65. DOI: https://doi.org/10.1109/2.675637

[10] P. Bose. 1994. Architectural timing verification and test for super scalar processors. In *Proceedings of the 24th International Symposium on Fault-Tolerant Computing. FTCS-24. Digest of Papers*. 256–265. DOI: https://doi.org/10.1109/FTCS.1994.315635

[11] Pradip Bose and Thomas M. Conte. 1998. Performance analysis and its impact on design. *Computer* 31, 5 (May 1998), 41–49. DOI: https://doi.org/10.1109/2.675632

[12] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli. 2012. Accuracy evaluation of GEM5 simulator system. In *Proceedings of the 2012 7th International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC'12)*. 1–7. DOI: https://doi.org/10.1109/ReCoSoC.2012.6322869

[13] Rajagopalan Desikan, Doug Burger, and Stephen W. Keckler. 2001. Measuring experimental error in microprocessor simulation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA#x02019;01)*. ACM, New York, 266–277. DOI: https://doi.org/10.1145/379240.565338

[14] Agner Fog. 2000. Test programs for measuring clock cycles and performance monitoring. http://www.agner.org/optimize/#testp.

[15] Karthik Ganesan, Jungho Jo, and Lizy K. John. 2010. Synthesizing memory-level parallelism aware miniature clones for SPEC CPU2006 and implantbench workloads. In *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS'10)*. IEEE, 33–44.

[16] Karthik Ganesan and Lizy K. John. 2011. MAximum Multicore POwer (MAMPO): An automatic multithreaded synthetic power virus generation framework for multicore systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 53.

[17] Alon Gluska. 2003. Coverage-oriented verification of banias. In *Proceedings of the 40th Annual Design Automation Conference*. ACM, 280–285.

[18] Anthony Gutierrez, Joseph Pusdesris, Ronald G. Dreslinski, Trevor Mudge, Chander Sudanthi, Christopher D. Emmons, Mitchell Hayenga, and Nigel Paver. 2014. Sources of error in full-system simulation. In *Proceedings of ISPASS*.

[19] Alex Izvorski. 2006. mubench. http://mubench.sourceforge.net.

[20] Ajay M. Joshi, Lieven Eeckhout, Lizy K. John, and Ciji Isen. 2008. Automated microprocessor stressmark generation. In *Proceedings of the IEEE 14th International Symposium on High Performance Computer Architecture (HPCA'08)*. IEEE, 229–239.

[21] Youngtaek Kim, Lizy Kurian John, Sanjay Pant, Srilatha Manne, Michael Schulte, William Lloyd Bircher, and Madhu Saravana Sibi Govindan. 2012. AUDIT: Stress testing the automatic way. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'12)*. IEEE, 212–223.

[22] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. 2002. Simics: A full system simulation platform. *Computer* 35, 2 (2002), 50–58.

[23] Larry McVoy and Carl Staelin. 1996. Lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference (ATEC'96)*. USENIX Association, 23–23. http://dl.acm.org/citation.cfm?id=1268299.1268322

[24] M. Moudgill, P. Bose, and J. H. Moreno. 1999. Validation of turandot, a fast processor model for microarchitecture exploration. In *Proceedings of the 1999 IEEE International Performance, Computing and Communications Conference*. 451–457. DOI: https://doi.org/10.1109/PCCC.1999.749471

[25] T. Nowatzki, J. Menon, C. H. Ho, and K. Sankaralingam. 2015. Architectural simulators considered harmful. *IEEE Micro* 35, 6 (Nov. 2015), 4–12. DOI: https://doi.org/10.1109/MM.2015.74

[26] R. Panda and L. K. John. 2017. Proxy benchmarks for emerging big-data workloads. In *Proceedings of the 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'17)*. 139–140. DOI: https://doi.org/10.1109/ISPASS.2017.7975285

[27] Stephen K. Park and Keith W. Miller. 1988. Random number generators: Good ones are hard to find. *Commun. ACM* 31, 10 (1988), 1192–1201.

[28] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. 2011. MARSS: A full system simulator for multicore x86 CPUs. In *Proceedings of the 48th Design Automation Conference (DAC'11)*. ACM, 1050–1055. DOI: https://doi.org/10.1145/2024724.2024954

[29] Jose Renau, Basilio Fraguela, James Tuck, Wei Liu, Milos Prvulovic, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, and Pablo Montesinos. 2005. SESC simulator. Retrieved January 2005 from http://sesc.sourceforge.net.

[30] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. 2011. DRAMSim2: A cycle accurate memory system simulator. *IEEE Comput. Arch. Lett.* 10, 1 (Jan. 2011), 16–19. DOI: https://doi.org/10.1109/L-CA.2011.4

[31] Ali G. Saidi, Nathan L. Binkert, Lisa R. Hsu, and Steven K. Reinhardt. 2005. Performance validation of network-intensive workloads on a full-system simulator. In *Proceedings of the 1st Annual Workshop on Interaction Between Operating System and Computer Architecture*. 33–38.

[32] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. 2012. Multi2Sim: A simulation framework for CPU-GPU computing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. ACM, New York, 335–344. DOI: https://doi.org/10.1145/2370816.2370865

[33] V. Uzelac and A. Milenkovic. 2009. Experiment flows and microbenchmarks for reverse engineering of branch predictor structures. In *Proceedings of the 2009 IEEE International Symposium on Performance Analysis of Systems and Software*. 207–217. DOI: https://doi.org/10.1109/ISPASS.2009.4919652

[34] Antti Valmari. 1998. *The State Explosion Problem*. Springer, Berlin, 429–528. DOI: https://doi.org/10.1007/3-540-65306-6_21

[35] Harry Wagstaff, Bruno Bodin, Tom Spink, and Bjoern Franke. 2017. *SimBench: A Portable Benchmarking Methodology for Full-System Simulators*. IEEE.

[36] H. Wong, M. M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. 2010. Demystifying GPU microarchitecture through microbenchmarking. In *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS'10)*. 235–246. DOI: https://doi.org/10.1109/ISPASS.2010.5452013

Received June 2017; revised October 2017; accepted December 2017