A Survey of Cache Simulators

HADI BRAIS, Indian Institute of Technology Delhi
RAJSHEKAR KALAYAPPAN, Indian Institute of Technology Dharwad
PREETI RANJAN PANDA, Indian Institute of Technology Delhi

Computer architecture simulation tools are essential for implementing and evaluating new ideas in the domain and can be useful for understanding the behavior of programs and finding microarchitectural bottlenecks. One particularly important part of almost any processor is the cache hierarchy. While some simulators support simulating a whole processor, including the cache hierarchy, cores, and on-chip interconnect, others may only support simulating the cache hierarchy. This survey provides a detailed discussion on 28 CPU cache simulators, including popular or recent simulators. We compare between all of these simulators in four different ways: major design characteristics, support for specific cache design features, support for specific cache-related metrics, and validation methods and efforts. The strengths and shortcomings of each simulator and major issues that are common to all simulators are highlighted. The information presented in this survey was collected from many different sources, including research papers, documentations, source code bases, and others. This survey is potentially useful for both users and developers of cache simulators. To the best of our knowledge, this is the first comprehensive survey on cache simulation tools.

CCS Concepts: • **Computing methodologies** → **Simulation tools**; • **Computer systems organization** → *Processors and memory architectures*;

Additional Key Words and Phrases: Hardware caches, cache simulators, cache metrics, simulator validation

ACM Reference format:
Hadi Brais, Rajshekar Kalayappan, and Preeti Ranjan Panda. 2020. A Survey of Cache Simulators. *ACM Comput. Surv.* 53, 1, Article 19 (February 2020), 32 pages.
https://doi.org/10.1145/3372393

1 INTRODUCTION

A cache simulator is a software tool that mimics the behavior of a hardware cache subsystem. A cache is used to reduce the average cost of accessing main memory from the processor. The performance difference between main memory and the processor has increased over time, necessitating the development of more sophisticated caches. One way to evaluate a new cache subsystem is by producing a hardware prototype and evaluating its performance. However, this process is often costly and time-consuming. An alternative way is to implement the new idea in a simulator and

We thank Radhika Dharwadkar for running a large number of experiments as part of this project. Hadi Brais was supported by a TCS Fellowship while working on this project.
Authors' addresses: H. Brais and P. R. Panda, Department of Computer Science and Engineering, Indian Institute of Technology Delhi, Hauz Khas, New Delhi 110016, India; emails: hadi.b@live.com, panda@cse.iitd.ac.in; R. Kalayappan, Department of Computer Science and Engineering, Indian Institute of Technology Dharwad, Dharwad, India; email: rajshekar.k@iitdh.ac.in.

0360-0300/2020/02-ART19 $15.00
https://doi.org/10.1145/3372393

evaluate it using simulation. This facilitates design space exploration [93] and makes it accessible to a wider community of researchers and students. Simulators can also be used to study the behavior of programs on existing processors where hardware performance events are not sufficient or unavailable. In addition, when the study needs to be performed on many different existing systems, one could avoid purchasing all these systems by using a simulator.

Some of the cache simulators that were developed in the 1990s were used extensively in research [25, 62, 82]. Many other simulators have been developed in the following years, and new simulators are released almost every year. While many of them remained free and open source, some were commercialized, and others are rarely used anymore. We consider in this work only publicly known non-commercial simulators that are currently popular, were once popular, were developed recently, or employ unique innovative techniques. General-purpose processor architecture simulators do support cache simulation, and we consider them in this work. In other words, we consider the set of cache simulators to be a superset of general architectural simulators. Since our focus is on cache simulation, we will refer to them as cache simulators as well.

We present a comprehensive survey on cache simulators, providing a comparison between their capabilities and recommendations on the choice and use of simulators. We also discuss the intricacies involved in designing a cache simulator and highlight a number of major issues that currently exist in most or all academic cache simulators. Many simulators have been validated either against real hardware or against other simulators. We show that validation is often misinterpreted and provide guidelines on how to validate simulators and how to interpret the results of validation experiments. A full list of simulators that we study in this work is shown in Table 1. The information presented in this survey was collected from many different sources, including research papers, documentations, source code bases, simulator-specific forums, our correspondences with the authors or current developers of the simulators, and from our own experience in using theses tools. We made every effort to ensure the correctness of the information presented here with respect to the latest official versions of the simulators that were publicly available at the time of submitting this article for publication. We also maintain an up-to-date and complete version of this survey online[1] and accept contributions from the community.

The scope of this survey is limited to traditional CPU caches. Hardware caches have been used in almost all kinds of processors and appear in many different designs and configurations. Modern compute-capable GPUs typically have a two-level cache hierarchy. In contrast to traditional multicore CPUs, GPUs may have separate caches for scalar and vector data and may have special caches such as constant and texture caches. Some GPUs offer caches with configurable sizes. Caches in CPUs, however, are generally much more sophisticated. Manycore processors have cache hierarchies that are similar in simplicity to GPUs but without the special-purpose caches. Some of the simulators considered in this work support heterogeneous architectures. But our focus will be on CPU caches. Therefore, cache or architectural simulators that are dedicated for GPUs and other kinds of processors (such as embedded processors, DSPs, and other accelerators) are not considered here. Nonetheless, CPU caches share a lot of similarities with other kinds of caches and so we believe that our survey would be relevant and useful for the majority of the architecture research community.

Furthermore, we only consider a class of cache simulators known as instruction set simulators (ISS). Simulators at different levels of abstraction (e.g., transaction level and hardware level) are not considered [43, 65, 85]. Simulators that use techniques that are not popular in our target domain are not considered as well [18, 48]. Techniques that can be used to enhance the performance of

[1] https://github.com/hadibrais/archsim.

Table 1. Comparison of Some of the Design and Development Aspects of Cache Simulators

Simulator	Type	Level	Mode	Scope	Last Updated	Open Source
Cachegrind [59]	F	App	X	P	2015	O
Dinero IV [25]	F	N/A	MT	P	1999	O
CASPER [36]	F	N/A	MT	P	2003	N
CMP$im [38]	F	App	X	P	2009	N
Moola [76]	T	N/A	MT	P	2015	R
gem5 [14]	T	App, FS	E, IT	M	2019	O
Sniper [22]	T	App	X, IT	P, M	2019	R
Tejas [73]	T	App, FS	X, E, IT	M	2016	O
ZSim [72]	T	App	X	M	2016	O
MultiCacheSim [50]	F	App	X	P	2013	O
drcachesim [17]	F	App	X, MT	P	2016	O
MARSSx86 [64]	T	FS	E	M	2016	O
Multi2Sim [80]	T	App	E	M	2018	O
SimpleScalar [12]	T	App	E	M	2011	O
ESESC [10]	T	App	E	M	2019	O
Graphite [55]	T	App	X	M	2015	O
HORNET [69]	T	App	X, E	M	2011	O
SlackSim [23]	T	App	E	M	2010	N
Manifold [84]	T	App, FS	E	M	2016	O
vCSIMx86 [41]	F	N/A	MT	P	2013	O
pycachesim [31]	F	N/A	N/A	P	2017	O
SMPCache [82]	F	N/A	MT	P	2000	N
SiNUCA [8]	T	App	IT	M	2015	O
COTSon [11]	T	App, FS	E	M	2010	O
McSimA+ [4]	T	App	X, IT	M	2013	O
XIOSim [40]	T	App	X	M	2016	O
Zesto [49]	T	App	E	M	2009	R
MacSim [44]	T	App	IT	M	2019	O

Note: **F** = functional, **T** = timing, **App** = user-level, **FS** = full-system, **X** = execution-driven, **E** = emulation-driven, **MT** = memory trace-driven, **IT** = instruction trace-driven, **P** = cache simulation only, **M** = more than a cache simulator, **O** = open source, **N** = closed source, **R** = source code provided on request, **N/A** = not applicable.

simulators in software [16, 26, 28, 32, 39] or on FPGAs [9, 42, 66] are not discussed in this work, since they merit surveys on their own.

The memory technology that a cache is made of determines the power, performance, reliability, and security characteristics of the cache. Static random-access memory (SRAM) is typically used to build caches, but dynamic random-access memory (DRAM) has also been used for last-level caches in real processors. Most cache simulators have no inherent notion of memory technology. Instead, they take the characteristics of the desired technology as input, which are estimated by a separate cache modeling tool. Emerging non-volatile caches and 3D caches can be abstracted similarly. However, these caches do have unique features that may be important to consider during simulation. For example, non-volatile caches suffer from limited endurance and their energy consumption may be dependent on the data itself that is being written to it. To study such unique features, dedicated simulators need to be used. Such simulators are also not considered in this work.

To keep this survey manageable, we discuss the features supported by cache simulators at a somewhat abstract level of detail. For example, there can be many designs for lockup-free caches. Discussing exactly how each cache simulator supports lock-free caches would require a lot of effort, consume a lot of space from the article, and make it less readable, yet without adding much value, since the survey is not about any feature in particular. Instead, we specify which simulators support lockup-free caches without getting bogged down in exact implementation details.

A number of surveys have been conducted on processor architecture simulators. Akram and Sawalha [5] present a broad survey on computer architecture simulators. It covers techniques to enhance simulation speed and provides experimental comparison between numerous x86 simulators. Zang et al. [91] present a survey on power cache tuning techniques and discuss which simulators to use for that purpose. Uhlig et al. [81] present a detailed survey on trace-driven memory (including cache) simulators. That survey was published in 1997, and a large number of simulators and techniques have been developed since then. Aleem et al. [7] present a survey on heterogeneous processor simulators. To our knowledge, no surveys on cache simulators of any kind have been published in the past decade. Such tools are absolutely crucial for research in areas such as computer architecture and embedded systems. In addition, in this survey, the number of simulators considered is substantially larger compared to most of the aforementioned surveys,[2] thereby making it the most comprehensive survey on computer architecture simulators that is focused on CPU caches. We have identified 28 simulators that are within the scope of this survey, and detailed data are provided on each one of them.

The rest of this article is organized as follows. Section 2 discusses a number of attributes along which cache simulators can be classified. Section 3 discusses specific features and techniques used in CPU caches and points out the simulators that support them. Section 4 discusses how to meaningfully validate a cache simulator and interpret the validation results. Finally, Section 5 concludes the article.

2 ARCHITECTURE OF CACHE SIMULATORS

There are many options for designing a cache simulator. Consider, for example, what the output of the simulator would be. If the simulator is supposed to measure a timing-related metric, then it needs to keep track of the time required to complete various operations. Also, since caches coexist with other components of a computer system, a simulator may simulate some of these components in addition to the cache memory. We define a number of important dimensions along which cache simulators can be classified and demonstrate the advantages and disadvantages of the design choices available along these dimensions. Table 1 shows all the simulators,[3] in no particular order, that have been considered in this survey together with design-related and development-related attributes. These attributes are explained in the following subsections.

2.1 Types of Simulators

A cache simulator can be either a functional or timing simulator. A functional simulator[4] does not have a notion of time and only logically simulates the operations performed by the target architecture to run a program. Functional simulation can be used for three purposes: to perform

[2]Except for Reference [5], which does mention a larger number of simulators, but we have carefully excluded some of them.

[3]MARSSx86 is largely based on another simulator called PTLsim [90], which is fairly popular but not being maintained. Therefore, we decided to only discuss MARSSx86.

[4]We consider functional emulation different from functional simulation. Functional emulators do not model internal details of the architecture and are used to enable compatibility. Although a functional simulator may employ emulation to run the target Application Binary Interface (ABI) on the host ABI.

statistical evaluation by counting the frequency of interesting events (such as cache misses), to generate traces of the operations performed, and to verify the functional correctness of the design being simulated. While functional simulators can generate instruction and memory access traces and are often used for this purpose, dynamic binary instrumentation tools can be used to generate such traces much more efficiently for that particular purpose [33]. However, functional simulators can inherently generate traces that are much more detailed.

Timing simulators keep track of time by considering how much time each operation takes to execute. A timing simulator can be a complex cycle-level simulator that includes a detailed microarchitecture model. Examples of such simulators include gem5 and Sniper. Such level of detail significantly increases simulation time [38], thereby limiting the number of instructions and the number of configurations of the platform to be simulated [19]. Moreover, compared to functional simulators, timing simulators require much more effort to develop in general. Strictly speaking, a timing simulator may not necessarily support any statistical metrics. However, it can naturally produce these metrics, because it already performs the required functionality. All timing simulators listed in Table 1 are necessarily functional as well. A timing simulator does not necessarily support any of the timing-related cache simulation metrics discussed in Section 3.13, although it is much easier to support some of these metrics in a timing simulator than a functional simulator that has no notion of time built in.

A timing simulator can be simple wherein the amount of time it takes for an instruction to execute is only determined by the time required for the operands to become available, irrespective of the instruction itself. Moola follows this approach and places more emphasis on the impact of the cache subsystem on performance while disregarding the impact of the microarchitecture. Although Moola may be faster than cycle-level simulators, it can be less accurate, because the microarchitecture affects the order and the number of cache accesses. Simulation accuracy is discussed in more detail in Section 4.

We use the term "cycle-level" in preference to the more popular term "cycle-accurate," because it emphasizes that simulation is performed at the cycle granularity without specifying any guarantees about accuracy. That said, companies that design commercial architectures such as ARM do have internal cycle-accurate simulators and are needed to validate hardware designs before fabricating them, and to accurately evaluate new ideas before using them in production systems [27]. Such simulators require a lot more effort to develop and not available in academia, because many details of their designs are concealed.

2.2 Levels of Simulation

The level of simulation (Application or Full System) refers to whether the simulator runs one or more applications or one or more operating systems. In the former case, the simulator will only be capable of simulating the user-level execution of the applications, because it is not aware of the instructions being executed on a system call including privileged instructions. Such simulators are less complex and faster and convenient for applications that spend most of their time in user mode such as the SPEC CPU benchmark suite [34]. A simulator that can run one or more operating systems is called a full-system simulator and can be used to capture all instructions that will be executed, potentially resulting in a more accurate simulation [24]. Sniper and SimpleScalar are popular application-level simulators. gem5, Tejas, Manifold, and COTson support both simulation levels.

The authors of McSimA+ describe it as a simulator that is in between a full-system simulator and an application-level simulator. That is because when a program is loaded, the simulator redirects all standard *pthreads* library calls to a special pthreads library that creates virtual threads and schedules their execution. This enables the user of the simulator to easily devise a custom thread scheduler without needing to modify operating systems to run on a full-system simulator. That is indeed a convenient feature in McSimA+, because its purpose is to simulate emerging, potentially asymmetric, manycore computer systems for which operating systems may not be available or easily obtainable. However, for the purpose of this survey, we consider McSimA+ as an application-level simulator, because it does not support simulating system calls and privileged instructions.

2.3 Modes of Simulators

The mode of simulation defines how the simulator works. We define three modes of simulation:

- Execution-driven: The simulator runs the input program to be simulated natively on the host platform. There are three immediate consequences to this approach. First, the Application Binary Interface (ABI) of the input program has to be compatible with the host platform. Second, the performance overhead of running the input program itself is minimized. Third, system calls are executed natively and kernel-mode instructions typically do not impact the architectural state of the simulator. This is the case in all execution-driven simulators considered in this survey.
- Emulation-driven: The simulator creates a virtual platform on which the input program is run. This enables the simulator to simulate programs written in ABIs that may or may not be compatible with the host platform but at the cost of higher overhead of executing the program. An emulation-driven simulator emulates system calls rather than executing them natively, but the impact of system calls depends on the level of simulation. Multi2Sim supports only the App level. gem5 supports both options; in SE mode, system calls are emulated and do not impact the architectural state, while in FS mode, system calls and kernel-mode instructions impact the the architectural state of the simulator.

  An emulation-driven simulator does not only emulate the instructions of the target executables but also process creation and termination. Therefore, the first instruction that will be simulated is the first instruction in the entry point of the main executable of the program being simulated and the last instruction is the last instruction in the program. However, an execution-driven simulator relies on the OS to create, initialize, and terminate the process in which the target program will run. The simulator may simulate instructions that are not exactly part of the program but belong to process initialization or termination. This behavior might be desirable, might perturb the measurements, or might not make any significant difference, depending on what is being measured and for what purpose.
- Trace-driven: The simulator takes as input a trace file generated from an execution or emulation of a program. Trace-driven simulation completely eliminates the overhead of executing the input program. However, emitting, maintaining, and consuming large trace files introduces additional overhead. A trace file may either contain dynamic instructions or descriptions of memory access streams, depending on the purpose of the simulator. Trace-driven simulators tend to be simpler in design and implementation. In addition, they enable us to simulate programs on incompatible systems, since trace file formats are typically independent of the platform the program was run on. Once a trace file is generated by running a program, simulation can be performed many times targeting different configurations without having to run the program again. Both full-system simulators and application-level simulators can generate instruction or memory traces that can be fed to a trace-driven simulator. The level of simulation can have a substantial impact on the traces generated [60]. Trace-driven simulators are more deterministic compared to other simulators, which makes them in some cases more accurate for performance evaluation without having to use statistical methods [6, 46]. Such determinism only enhances accuracy in situations where

the non-deterministic effects, such as those stemming from inter-thread interactions, are not of interest. To our knowledge, no techniques have been developed that would enable a trace-driven simulator to "revive" non-deterministic effects. Typically, traces do not contain instructions or memory accesses resulting from speculative execution,[5] but existing trace-driven simulators can be enhanced to support speculation [56]. A speculative processor can execute instructions speculatively with non-deterministic effects.

It is possible to modify an execution-driven simulator to emulate some instructions, resulting in a hybrid mode of simulation. For example, consider a situation where the Sniper simulator is used to simulate an application that uses an instruction that is not supported on the host processor (either because the instruction is newly proposed or it is supported on other processors that cannot be easily obtained). Sniper uses the Intel Pin dynamic binary instrumentation framework [51] to determine the instructions to be simulated. In this case, the application can still be simulated using an execution-driven simulator like Sniper by intercepting that instruction at the time it is about to be executed, simulating its behavior in software, and skipping its execution on the native hardware. This is a form of emulation-driven simulation, because the instruction is not natively executed.

Some simulators support multiple modes of simulation. Typically, full-system simulators use emulation while application-level simulators use native execution. Memory trace-driven simulators can be either full-system or application-level simulators, depending mostly on the contents of the trace files and how they were generated. Therefore, the level of simulation is not applicable to such simulators.

pycachesim is a unique simulator and does not use any of these modes of simulation. In pycachesim, both the configuration of the cache subsystem and the memory accesses to be simulated are specified in a Python script. By executing the script, memory accesses are simulated, but there is no easy way to use such a simulator to simulate precompiled executable binaries. It is mostly useful for educational purposes.

2.4 Other Attributes

Simulators can be classified according to what is being simulated. They can be purely cache simulators, single-core system simulators, system-on-chip simulators, cluster simulators, and so on. However, in this survey, we are only concerned with cache simulation and so we define only two scopes: pure cache simulators and simulators that can do more than cache simulation. Pure cache simulators are simpler but may be less accurate for measuring certain metrics. However, pure cache simulators are smaller in size in terms of total lines of code, and therefore, they may have less bugs overall. Sniper is the only simulator that can run either as a pure cache simulator or as a multicore system simulator. In cache-only mode, each core executes each instruction in a fixed amount of time except when it accesses memory, in which case memory latencies are considered.

Two development-related attributes are considered: *"last updated"* and *"open source."* The "last updated" attribute specifies the year during which the simulator was last maintained. Some relatively old simulators, such as gem5 and Multi2Sim, are still being maintained and improved. While other simulators, such as Dinero IV and SMPCache, have been developed a long time ago and are no longer maintained. The reason that these simulators are mentioned in this survey is that they were extensively used in the late 1990s and early 2000s. In addition, we would like to show throughout

[5]Non-determinism is different from speculation. In the former, the operations to be performed and the possible values that can be observed are known, but the order in which they are performed is not known until they are (mostly) executed or accessed. In the latter, the operations to be performed or the values to be accessed are not known yet, and, hence, they are predicted.

this survey how cache simulators have evolved over time and coped with the increasing complexity of the cache subsystem.

The other attribute, “open source,” specifies whether the source code of the simulator is available for download and how. Most simulators are open source (with potentially different licenses) where the source is directly available. Moola, Sniper, and Zesto can be obtained by sending a request to the authors or filling an application. Some simulators are not available and have been considered in this survey for completeness, because they have been used in many research papers. However, it is generally recommended that open source simulators be used [61]. This enables the readers and reviewers to examine and independently verify the suitability of a particular version of the simulator used to evaluate a proposed technique and also to reproduce the results, thereby enabling further research.

Another important design aspect of simulators is the programming languages used for implementation (not shown in Table 1 due to limited space). Familiarity with the programming language enables the user to make changes more easily. Most simulators are implemented in C or C++. Some parts of gem5 and pycachesim are implemented in Python. Most of Tejas is implemented in Java.

2.5 Target Platforms

One important feature of a cache simulator is the platforms that can be simulated (referred to as the target platforms). When choosing a cache simulator, those that do not support the target platform can be immediately ruled out. We have omitted supported target platforms from Table 1 to keep it simple. Some simulators support a large number of platforms, but most of them support input programs that target x86/Linux. The concept of target platforms does not apply to memory trace-driven simulators.

Although the documentations of the respective simulators state that x86/Linux is supported, x86 is actually a family of instruction sets that gets expanded with more instructions and features when new processors are released. We found that most simulators do not quickly cope with new additions to x86 and poorly document the details of their support. Researchers usually refrain from using new extensions to x86 when compiling benchmarks to avoid this issue. However, we believe neglecting new additions to x86 and other ISAs may reduce the quality, impact, or contributions of the research work. This is one of the major issues in most or all academic cache simulators that are currently available. We highlight other major issues throughout the article.

2.6 Summary

The categorization based on the levels of simulation discussed in Section 2.2 is consistent with [5] (where it is called the scope of the target). However, the type-based and mode-based categorization we have proposed here are different from Reference [5]. First, we consider all timing simulators to be functional simulators as well. However, this is not necessary according to Reference [5], although no example of a timing simulator that is not functional was given. Second, we place the execution-driven and emulation-driven simulation techniques in separate categories, because they technically work differently and have different pros and cons as discussed in Section 2.3.

Figure 1 presents a high-level pictorial depiction of the cache simulator architecture design space. Different simulators have different attributes—a simulator designer or user may use this diagram to navigate the design space and focus on those simulators that have the attributes that would best serve them in their current endeavour.

3 ASPECTS OF CACHE SIMULATION

Various aspects of a cache’s functionality could be modeled in a simulator. The cache model adopted by the simulator depends upon the usage envisioned for it. The model may not incorporate

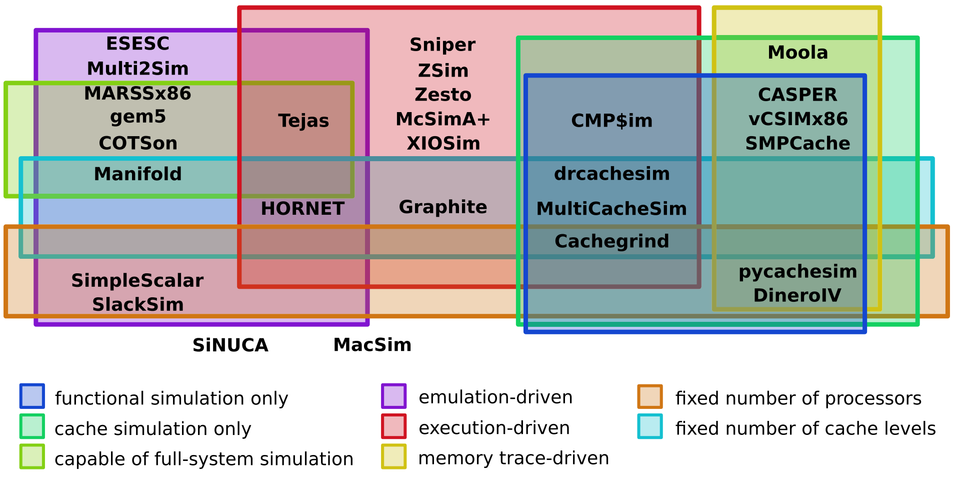

Fig. 1. The placement of the surveyed simulators in the cache simulator design space.

all the aspects, as they may not be required for the study the researcher wishes to perform. Reducing or approximating certain aspects has many benefits—faster simulation time, fewer chances of bugs, and faster development time. In this section, we will discuss the different aspects of a cache model, what kind of studies are enabled by these aspects, what it costs to include them in the model, and how certain popular cache simulators implement these aspects. A summarized comparison of different simulators in terms of their configurability is given in Table 2.

A substantial number of cache architecture variations have been proposed over the past few decades [2, 68, 71, 75]. However, many of these proposals have not been used in production processors. In this survey, we focus on cache architectures and techniques that have been used in production processors and discuss to what degree the selected cache simulators support them.

3.1 Stand-Alone Caches

3.1.1 A Minimal Model. We first discuss the modeling of a stand-alone cache. Every practical cache simulator allows the specification of the size of the cache, the line size, and the associativity. Another important aspect is the *line replacement policy*, that is, the policy followed to select the line to be removed from the set to make place for a new line. Some of the common policies described in the literature include the Least Recently Used (LRU), First In First Out (FIFO), and Random policies. Many simulators, such as gem5 and Sniper, allow choosing the replacement policy to be used. Other simulators, such as Tejas, have the policy fixed (LRU, in this case). However, extending these simulators to support other policies is not a laborious task.

The four aspects of cache size, line size, associativity, and replacement policy are minimally sufficient to calculate the *hit and miss rates* experienced by the cache. Thus, if the researcher is interested in merely these metrics, then this simple cache model is sufficient. Such a model is adopted by simulators such as DineroIV. The simplicity of these models generally results in higher simulation speeds.

3.1.2 Considering Cache Access Latency. The researcher may not be interested in just hit and miss numbers but also in the latencies of accesses. Two additional aspects are required to incorporate latencies in a cache simulator:

First, when dealing with simply hits and misses, the *trace*, or an ordered list of addresses to be accessed, forms the input. When latencies need to be modeled, the times at which the access

Table 2. Comparison of the Configurability of Cache Simulators

Simulator	Configuration Parameters																			
	NL	TS	A	LS	NP	SU	PS	UN	CO	AL	RP	WP	BK	PO	BW	BB	BA	FE	IE	LF
Cachegrind	F	Y	Y	Y	F	F	F	F	*	*	F	F	F	*	*	*	*	F	F	*
DineroIV	Y	Y	Y	Y	F	Y	F	F	*	*	Y	Y	F	*	*	*	*	Y	F	*
CASPER	Y	Y	Y	Y	Y	Y	NP	F	Y	Y	Y	F	F	F	*	*	*	Y	F	?
CMP$im	Y	Y	Y	Y	Y	Y	Y	?	?	?	Y	Y	Y	Y	?	?	?	?	Y	?
Moola	Y	Y	Y	Y	Y	Y	NP	Y	Y	Y	Y	Y	F	F	*	*	*	Y	F	L
gem5	Y	Y	Y	Y	Y	Y	Y	F	Y	Y	Y	Y	F	Y	Y	Y	Y	Y	Y	LF
Sniper	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	F	Y	Y	Y	Y	Y	F	LF
Tejas	Y	Y	Y	Y	Y	Y	Y	Y	F	Y	F	Y	Y	Y	Y	Y	Y	F	F	LF
ZSim	Y	Y	Y	Y	Y	Y	Y	Y	F	Y	Y	Y	Y	F	F	F	F	F	F	LF
MultiCache	F	Y	Y	Y	Y	F	NP	F	Y	*	F	F	F	*	*	*	*	F	F	*
drcachesim	F	Y	Y	Y	Y	F	NP	F	*	*	Y	F	F	*	*	*	*	F	F	*
MARSSx86	Y	Y	Y	Y	Y	Y	Y	F	Y	Y	F	Y	F	Y	Y	Y	Y	Y	F	L
Multi2Sim	Y	Y	Y	Y	Y	Y	Y	F	F	Y	Y	Y	F	Y	Y	Y	Y	F	F	LF
SimpleScalar	Y	Y	Y	Y	F	Y	F	F	*	Y	Y	F	F	F	*	*	*	F	F	LF
ESESC	Y	Y	Y	Y	Y	Y	NP	F	Y	Y	Y	F	Y	Y	*	*	*	Y	F	LF
Graphite	F	Y	Y	Y	Y	F	NP	Y	Y	Y	Y	Y	Y	F	Y	Y	Y	Y	F	?
HORNET	F	Y	Y	Y	Y	F	NP	F	Y	Y	Y	F	F	Y	Y	Y	Y	F	Y	?
SlackSim	Y	Y	Y	Y	F	Y	F	F	*	Y	Y	F	F	F	*	*	*	F	F	LF
Manifold	F	Y	Y	Y	Y	F	NP	F	F	Y	F	F	F	Y	Y	Y	Y	Y	F	LF
vCSIMx86	Y	Y	Y	Y	Y	Y	Y	F	F	*	Y	Y	F	F	*	*	*	Y	F	*
pycachesim	Y	Y	Y	Y	F	F	F	F	*	*	Y	Y	F	F	*	*	*	F	F	LF
SMPCache	Y	Y	Y	Y	Y	F	F	F	Y	*	Y	Y	F	F	Y	*	Y	F	F	?
SiNUCA	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	F	Y	F	Y	Y	Y	Y	F	LF
COTSon	Y	Y	Y	Y	Y	Y	Y	F	Y	Y	Y	Y	F	F	Y	Y	Y	Y	F	LF
McSimA+	Y	Y	Y	Y	Y	Y	NP	Y	Y	Y	F	F	Y	F	Y	Y	Y	Y	F	LF
XIOSim	Y	Y	Y	Y	Y	Y	Y	F	*	Y	Y	Y	Y	F	Y	Y	*	Y	F	LF
Zesto	Y	Y	Y	Y	Y	Y	Y	F	*	Y	Y	Y	Y	F	Y	Y	*	Y	F	LF
MacSim	Y	Y	Y	Y	Y	F	F	Y	F	Y	F	F	Y	Y	Y	Y	Y	Y	F	Y

Note: **NL** = number of cache levels, **TS** = total cache size, **A** = associativity, **LS** = cache line size, **NP** = number of processor cores, **SU** = split data/instruction caches v/s unified caches, **PS** = private and shared caches, **UN** = uniform and non-uniform cache access, **CO** = cache coherence, **MT** = memory technology, **AL** = access latency, **RP** = replacement policy, **WP** = write policy, **BK** = number of cache banks, **PO** = number of ports, **BW** = bus width, **BB** = bus bandwidth, **BA** = bus arbitration, **FE** = fetch policy, **IE** = inclusive v/s exclusive caches, **LF** = lock-up free, or non-blocking caches. **Y** = configurable to some degree, **F** = fixed and simple, **L** = lock-up only, **LF** = lock-up free only, **NP** = private to one core or shared with all cores, * = not simulated or considered, **?** = unknown.

requests are made are also required. This timing information is typically provided by a co-running simulator of the processor core pipeline that models instruction-level dependencies and the related timing. Such an approach is used by all whole processor simulators (i.e., those simulators that simulate more than the cache hierarchy).

Second, the *access latency*, or the time taken to search for a line in the cache (the time taken for read and write requests may be mentioned separately), needs to be specified. All cache timing simulators, by definition, allow the specification of the access latency in the configuration file.

Knowledge of the request arrival time, the time taken to search for a line in the cache and satisfy the request, and the basic four aspects discussed earlier form the basis of a basic cache timing simulator. The four aspects help decide whether the request resulted in a hit or a miss. The latency model then determines how much time needs to elapse before the request can be serviced. Therefore, the simulator can now provide the clock cycle when a given request will be serviced. This model is still quite simplistic. Many other facets may be added to the model to make it more accurate and representative of real hardware. These will be elucidated in the next section.

3.1.3 Improving the Latency Model: Parallelism in Cache Access. A cache is a hardware structure that has fundamental limits on how many requests it can accept and how many it can begin servicing at any given point in time. The limitations have an effect on the time that the request has to wait before the cache begins servicing it and on the time the cache takes to service the request. This limitation is typically a function of the cache's current state.

Accesses to a cache are made through ports. A port may accept one request every cycle and can be read or write specific or can support both read and write requests. The number of ports that the cache has determines how many requests it can accept each cycle. Modeling ports requires modeling queues of waiting requests and arbitrating between simultaneous requests to the cache. Some simulators choose to do away with these complications and opt for an infinite number of ports at each structure. Other simulators allow configuring the number of ports.

Let the cache look-up time be T_{lookup}. Let the minimum time that has to elapse after one request has begun being serviced, before a second request can begin being serviced be $T_{request}$. At one extreme, a cache may be modeled so as to not begin processing a subsequent request until the current request has been fulfilled, that is, $T_{request} = T_{lookup}$. An alternative, more realistic, model allows the cache to begin processing successive requests even while the current one has not completed, that is, $T_{request} < T_{lookup}$. The latter model is termed "lock-up free" and offers pipelined parallelism in cache access. Most simulators support only a simplistic lock-up free model where $T_{request} = 1$ cycle. Moola and MARSSx86 support only the lock-up model.

The blocks in a cache may be organized into banks, typically done if the cache size is large. Banks may be accessed in parallel, independent of the others. Cache banks, thus, bring out another level of parallelism in cache access. Although modern processors have banked caches, support for this in academic simulators is limited.

3.2 The Cache Hierarchy

3.2.1 Organizations. Each processor core typically has multiple *levels* of caches between itself and the main memory. When a cache is requested for a line it does not hold, it requests for it from its lower level cache (that is closer to the main memory). A cache at a higher level, that is, one that is closer to the cores, is typically *private* to a core (though this is not a necessity). Lower caches are typically *shared* among two or more cores, with the *last-level cache* (LLC) typically shared between all the cores (these again are not necessities). The resultant cache hierarchy resembles an inverted tree structure.

Some simulators support a configurable cache hierarchy. Others support a fixed hierarchy of private L1 instruction and data caches and an L2 cache shared among all the cores. This used to be the most prevalent hierarchy more than a decade ago and is still common for low-power processors. Modern high-performance processors, however, have three to four levels of caches, with caches that are shared by some subset of the cores.

3.2.2 Write Policies. The write policy adopted by the caches forms a factor in determining the traffic (requests/responses) between the different cache levels and the contents of the caches themselves. A write to a cache line can be immediately communicated to the next level of the memory hierarchy or can be communicated when the line is evicted from the cache. The former is termed the "Write-Through" policy, and the latter is termed the "Write-Back" policy. Additionally, when a write is to be performed to a line not present in the cache, the line may be brought to the cache and written to, or the write can simply be communicated to the next level of the memory hierarchy. The former is termed the "Write Allocate" policy, while the latter is termed the "No Write Allocate" policy. Most simulators allow the user to choose between Write-Back and Write-Through. However, they may restrict the choice when it comes to allocation policy.

3.2.3 Cache Coherence Protocols and Cache Inclusiveness. Different private caches, belonging to the same cache level, may hold the same cache line. A coherence mechanism is required to ensure that all such lines contain the same content. A wide range of cache coherence protocols exist. The two major dimensions along which coherence protocols can be categorized are snoopy vs. directory and update vs. invalidate.

The model followed by Sniper has all private caches within a tile synchronize through a snoopy protocol, while inter-tile synchronization is through a directory protocol. Reducing the number of cores per tile to one can achieve a complete directory-based protocol. Sniper offers a range of protocol options such as MSI, MESI, and MESIF.

gem5 includes two cache subsystem models: the Classic model and the Ruby model. The Classic model only supports a simplified implementation of the MOESI snooping protocol. However, Ruby is a detailed, modular, and highly configurable cache subsystem model. It supports, among many other things, numerous coherence protocols.

Other simulators offer limited options. Cache coherence protocols might be conceptually simple, but implementing a new cache coherence protocol in a simulator requires a significant amount of effort in development and testing. gem5, however, particularly excels in this aspect, since Ruby's coherence protocols can be defined using a domain specific language called Specification Language for Implementing Cache Coherence (SLICC). SLICC has no parallel in any other simulator.

Cache inclusiveness is intertwined with coherence, and therefore we discuss it in this subsection too. If the contents of a cache are a strict subset of the contents of its immediate lower level cache, then it is termed an "inclusive" cache hierarchy. At the other extreme, if the contents of a cache are strictly not present in its lower level cache, then it is termed an "exclusive" cache hierarchy. A hierarchy can also be neither inclusive nor exclusive as well, with partial overlaps allowed between successive cache levels. Most simulators follow the inclusive model. Exceptions like gem5, however, allow configuring exclusive caches as well.

3.3 The Relationship with the Interconnect

3.3.1 Accessing Shared Caches. The paths between the cores and the private cache levels have deterministic, constant delays. Consequently, typical simulator models subsume this delay within the cache's access time. However, interactions with the shared caches and the main memory involve an interconnect shared with other on-chip elements. A simulator, in the interest of simplicity, may choose to model these interactions with a constant (possibly zero) delay. Cache-only simulators like drcachesim do not contain models of the interconnect. Many of the architectural simulators allow detailed specification of the interconnect in terms of bus width, router/arbiter delays, and arbitration policies. A strong interconnect model is required to achieve good cache simulator accuracy, when dealing with the more prevalent modern usage scenarios of multi-core, multi-threaded benchmarks, and non-uniform cache architectures.

3.3.2 Non-Uniform Cache Architectures. A single logical cache may be physically decomposed into disjoint components on the chip. This is typically done for the large LLC in the interest of reducing both access latency and power consumption. If it is decomposed, then accesses to different addresses by the same processor core incur different latencies. This is termed a "non-uniform" cache. If the cache is physically maintained as a single large component, then it is termed a "uniform" cache, as all accesses, regardless of the address, incur the same latency.

Moola, Sniper, Tejas, ZSim, Graphite, SiNUCA, and McSimA+ support static NUCA. Tejas and ZSim support dynamic NUCA as well. The rest support only uniform caches.

3.4 Cache Addressing Techniques

When a memory access is issued to a cache, four pieces of information are required: an index that specifies the row or the set in which the required cache line to be searched for may reside, a tag that identifies a line within a set, an offset into the line, and the number of bytes to access (which usually should not cross a cache line boundary). Typically, the first three pieces of information are specified by a memory address and the size is specified separately by the instruction that issued the memory access. Many architectures support virtual addressing in which the instructions being executed use virtual addresses that need to be translated to physical addresses to access main memory. To get the three pieces of information (index, set, offset), physical addresses, virtual addresses, or a combination of both can be used. This results in four main cache addressing techniques: physically indexed and tagged (PIPT), virtually indexed and tagged (VIVT), virtually indexed and physically tagged (VIPT), and physically indexed and virtually tagged (PIVT).

Trace-driven simulators inherently support both VIVT and PIPT caches. If the input trace contains virtual addresses, then the simulated caches are VIVT. If the input trace contains physical addresses, then the simulated caches are PIPT. For a trace-driven simulator to support other addressing schemes, it has to accept traces that contain both virtual and physical addresses. None of the existing trace-driven simulators support a mixed addressing scheme. Cachegrind, CMP$im, Tejas, ZSim, MultiCacheSim, SimpleScalar, Graphite, HORNET, SlackSim, pycachesim, and McSimA+ support only VIVT caches. gem5, Multi2Sim, Manifold, MacSim, and COTSon support only PIPT caches. MARSSx86 and ESESC support only VIPT caches. Sniper, XIOSim, and Zesto support VIVT and PIPT caches.

In some processors, different caches use different addressing techniques [37]. Some Intel Pentium 4 processors use both physical tags and a subset of virtual tags called virtual hints or vhints. The vhints are used to select one of the cache ways and then the physical tags are compared. Even before the tag comparison completes, the cache controller could provide the cache line to the processor and allow it to speculatively continue execution. All simulators considered in this survey do not support such techniques.

Modern Intel processors implement an addressing scheme known as "complex addressing" for the L3 cache. In that scheme, the L3 cache is divided into multiple slices, there being at least as many slices as cores. A given address is used to compute a hash value that determines the slice where the corresponding cache line can be found. The purpose of complex addressing is to improve effective L3 bandwidth and make some cache attacks more difficult [53]. None of the cache simulators considered in this article have built-in support for complex addressing.

3.5 Translation Lookaside Buffers

A translation lookaside buffer (TLB) is a special cache used to reduce the latency of mapping virtual addresses to physical addresses. Each entry in the TLB holds a virtual address of a page and the corresponding page table entry that contains the physical address of the page. The exact organization and functionality of a TLB depends on the instruction set architecture being simulated and the addressing techniques used for the various caches in the cache hierarchy. The organization of a TLB is similar to traditional caches in that TLBs can be associative, can be split or unified, can have multiple levels, and can include advanced techniques such as prefetching and victim caches.

Some simulators support only a fixed TLB organization. If a different organization is required, then non-trivial or significant source code changes have to be performed. All simulators that support TLBs provide some statistical metrics describing the behavior of TLBs but may or may not consider the resulting impact on latency. gem5 and MARSSx86 support a fixed TLB organization and consider its latency. drcachesim supports configurable TLBs but provides only statistical metrics. Sniper, Tejas, Multi2Sim, SimpleScalar, ESESC, Graphite, SlackSim, Manifold, COTson, McSimA+, XIOSim, Zesto, and MacSim all support configurable TLBs and consider latency. There are two latencies: one for a TLB hit and another for a TLB miss. The hit latency can be accurately modelled as a fixed quantity. However, the miss latency depends on whether the miss is handled by the OS or automatically by the hardware and how it is handled in either way. A miss is typically handled by performing an operation called a page walk that involves navigating a hierarchy of data structures. Different misses can have very different latencies. Only gem5, ESESC, MARSSx86, XIOSim, Zesto, and MacSim perform a page walk and measure the corresponding latency. The other simulators consider the miss latency as a fixed quantity. Another source of inaccuracy in simulation is in how the simulator handles accesses that cross page boundaries. Some simulators, such as Zesto, issue a single TLB lookup for such accesses. However, such accesses are rare, because compilers and memory allocators, by default, properly align allocated objects.

A very subtle impact that TLBs have on the traditional caches is related to what happens when a TLB miss occurs. If the memory holding the page tables is uncacheable, then the TLB will have to directly access them from main memory and the impact would be almost nonexistent. This is the approach followed by all simulators. However, if the page tables are cacheable, then they may exist in one or more levels of the cache subsystem. Therefore, a TLB miss will cause one or more cache accesses.

A typical TLB hierarchy consists of a data-TLB, an instruction-TLB, and a second-level TLB that is unified, split, or supports only data accesses. Split first-level TLBs are supported by all of the simulators. The following simulators support a second-level TLB: Sniper, ESESC, Manifold, XIOSim, and Zesto. Memory management units that are more complicated than that (such as the ones used in modern Intel microarchitectures and the three-level ITLB hierarchy used in the AMD Zen microarchitecture) are not supported by any of the simulators. This is a major shortcoming in all of the 28 simulators.

TLBSim [29] is a high-performance, functional simulator that is dedicated for simulating TLB hierarchies, which is why we have decided to not include it in Table 1. The simulator is available as open source and currently only supports the RISC-V ISA. Some of the noteworthy features of TLBSim include support for shared TLBs and infinite-sized TLBs. The accuracy of TLBSim was not studied.

3.6 Cache Controllers

The controller of a cache is responsible for directing the banks of the cache, and the tag comparison units, as well as communicating with the other components of the processor. This can include the pipeline if it is a first-level cache, a possible array of upper-level caches, a lower-level cache that is potentially divided into different slices (NUCA organization), directories providing coherence, and the on-chip main memory controllers. The controller may choose to prioritize between requests (like that done by main memory controllers). The controller may also adopt non-trivial tag matching and data organization schemes, such as those introduced by Panwar et al. [63].

Thus, research in this area requires that the cache controller be modeled as a separate entity, with configurable latency and energy parameters. Most simulators do not provide this flexibility, modeling the latency and energy of the cache as a single monolith. gem5, Sniper, MARSSx86, and

Multi2Sim allow the individual specification of the latency of the cache controller but not the energy.

3.7 Cache Enhancements

3.7.1 Victim Caches. A cache can be accompanied by a special cache called a victim cache. A victim cache is a small fully associative cache that holds lines that are evicted from the main cache. When a line is requested from the cache, it is simultaneously searched for in both the main and the victim caches. If the line was recently evicted from the main cache, then there is a good chance it will be found in the victim cache, thereby improving the hit rate.

Off-the-shelf, only pycachesim provides support for a victim cache. However, support can be added in most other simulators with reasonable effort.

3.7.2 Write Buffers. Like the victim cache, data and unified caches may be associated with a write buffer (also called a writeback buffer). Whenever a line needs to be written to the next level of the cache hierarchy, it is written into the write buffer. The write is considered complete, and the cache can proceed to the next request. When the bus to the next level is available, the write is performed. Lines enter and leave the write buffer in a first-in first-out order. If the write-buffer fills up, then the cache stalls. If while in the write buffer another write request arrives for the same line, then the two writes may be *combined* (or *coalesced*) into one, thereby saving bandwidth usage of the bus to the next level.

Two aspects exist regarding the modeling of write buffers—whether the buffer is finite sized or infinite and whether write combining is supported. Maintaining the waiting writes at a cache as events in the simulator is akin to an infinite buffer. This approach simplifies modeling and is adopted by DineroIV, HORNET, and COTSon. However, gem5, Sniper, Tejas, MARSSx86, pycachesim, SiNUCA, XIOSim, Zesto, and MacSim model finite-sized buffers. DineroIV, gem5, MARSSx86, vCSIMx86, and SiNUCA do not support write combining, while Sniper, Tejas, Multi2Sim, pycachesim, XIOSim, and Zesto do.

3.7.3 Prefetch Units. A cache may have functionality that predicts further requests and fetches the relevant lines before they are requested. Thus, when the request actually arrives, it enjoys a hit in the cache. Many simulators, including DineroIV, CASPER, Moola, gem5, Sniper, MARSSx86, ESESC, Graphite, Manifold, vCSIMx86, SiNUCA, COTSon, McSimA+, XIOSim, Zesto, and MacSim, provide support for prefetching. Evaluating custom prefetchers may be relatively easier on such simulators.

3.8 Dynamic Voltage and Frequency Scaling

With power among the primary concerns in processor design, a lot of effort is going into researching ways to reduce the power consumption. Most schemes involve scaling down the operating voltage and frequency of units, when it is known that the consequent performance loss will be negligible or will be within acceptable limits. To aid such research, simulators should provide support for dynamic voltage and frequency scaling (DVFS). This involves maintaining separate clock domains and capturing the interactions between them, in a temporal sense, accurately. While gem5, Sniper, ESESC, Graphite, Manifold, XIOSim, and MacSim provide support for DVFS, the majority of the simulators do not.

3.9 Pipeline Models

The pipeline determines the rates at which accesses are made to the cache hierarchy. Multiple pipeline models are possible. At one extreme, we can have a continuous stream of memory requests, with no regard to whether the previous requests have completed. This approach is followed by simulators like Cachegrind, DineroIV, CASPER, CMP$im, MultiCacheSim, drcachesim, vCSIMx86, pycachesim, and SMPCache, where only the hit and miss rates are of interest and not the latencies. It is also supported by HORNET and Manifold. Another extreme can be to issue a request only after the previous one has finished. This can capture the latency of each request, but as far as the overall execution time of the benchmark is concerned, it provides an over-estimation in general. Also, this model does not have requests interfering with each other in the cache subsystem. But in real systems, interference is prevalent. This brings us to the next model: approximate pipelines. An approximate pipeline assumes that a non-memory instruction commits every *n* cycles and models the timing of memory instructions by issuing requests to the memory system. Moola, Sniper, HORNET, and Manifold support approximate pipelines. A more realistic model is that of the in-order pipeline, which is commonly found in many embedded processors. In these, simultaneous accesses to the i-cache and the d-cache exist, and these may affect each other in the lower levels of the hierarchy in terms of evicting each other's lines, competing for ports, competing for the memory bus, and so on. Modeling an in-order pipeline is significantly more involved than the earlier two models, requiring an observance of data dependencies between instructions, availability of functional units, and many other details. A fourth model is that of the out-of-order pipeline, found in most modern general purpose and server class processors. Here, instructions may be executed out-of-order. Loads may be serviced in the pipeline itself through load-store queues in the pipeline. Memory requests may be issued out-of-order to the cache subsystem. All these have a significant impact on how the cache subsystem operates and, consequently, its performance. It is therefore necessary to model the pipeline—the feeder to the cache subsystem—accurately to study and compare different cache subsystems. The following simulators provide both in-order and out-of-order pipelines: gem5, Sniper, Tejas, ZSim, MARSSx86, Multi2Sim, SimpleScalar, ESESC, Graphite, SlackSim, Manifold, SiNUCA, COTSon, McSimA+, XIOSim, Zesto, and MacSim.

Another aspect of the pipeline that is of concern to the design and analysis of the cache subsystem is that of speculative execution. Speculation is done in the interest of performance, and a pipeline typically employs various speculation techniques—branch prediction, load value prediction, memory access address prediction, among others. If the speculation is incorrect, then the instructions executed as a result of this speculation make no functional contribution toward the workload's execution. They may, however, contribute toward the degradation of the performance of the processor, including its memory sub-system. Speculative instructions can include memory accesses. Even though the misspeculated accesses are functionally useless, they affect the execution of other memory accesses by contending for space in the cache and contending for cache ports and for other resources like networks-on-chip and main memory controllers. Thus, modeling speculation is an important facet of a cache simulator. Simulators like gem5, Sniper, ZSim, MARSSx86, Multi2Sim, SimpleScalar, ESESC, Graphite, SlackSim, Manifold, COTSon, XIOSim, and Zesto have provisions for speculative instructions in their pipeline models.

3.10 Multithreading and Thread Scheduling

Many benchmarks and production applications are multithreaded, and it is important for a simulator to support multithreaded workloads. Dinero IV, SimpleScalar, SlackSim, pycachesim, and Zesto can simulate only single-threaded workloads.

An important aspect of simulating multiple threads is how the threads are scheduled. In full-system simulation mode, the threads are scheduled by the OS running on the simulator. This enables capturing real thread schedules. In user-level simulation mode, a simulator can either capture (or sample) the thread scheduling of the underlying OS or employ its own thread scheduling algorithms. Many simulators employ a fixed scheduling algorithm that may not be representative of how the threads would actually be scheduled. This can make the measurement of many output metrics, including hit and miss counts, highly inaccurate. Trace-driven simulators may use a thread schedule that is given as input as part of the input profile. Such simulators include CASPER and vCSIMx86. Execution-driven simulators need to be able to capture how threads are scheduled. Only COTson has this feature.

The schedule of threads on cores is a dominant factor in deciding the performance, power, and temperature of a multicore processor. This area of research has been highly active over the last decade and will continue to be so with the increasing number of cores on chip. Hardware schedulers are also widely proposed. Consequently, simulators must provide robust support for modeling schedulers, both software and hardware. Sniper gives the user a flexible interface to specify the scheduler algorithm and the times at which the scheduler is invoked.

3.11 Multiprogrammed Workloads

In many operating systems, a thread exists within the context of a process. A process provides a linear address space from which memory can be allocated and used by the threads of that process. In the previous section, the discussion assumed that all the threads being simulated belong to the same process. For a simulator to simulate threads from multiple processes, it has to be able to handle multiple address spaces. Simulating TLBs and multiple processes are two different features. Some simulators support only one of them, while others support neither or both. Trace-driven simulators inherently support simulating multiple processes by providing them with physical addresses rather than virtual addresses. Any of the full-system simulators that perform a page walk inherently support simulating multiple processes as well. In general, there are three techniques that can be used to support simulating multiple processes:

(1) Capturing the physical addresses that are mapped to the allocated portions of the virtual address space during execution.
(2) Assigning a unique identifier to each process and use it to differentiate between virtual addresses in different processes.
(3) Creating mapping from virtual to physical addresses during simulation. This mapping is likely to be different from the actual mapping used during execution.

CMP$im, Sniper, and Tejas use the second technique. The trace-driven mode of gem5 and Sniper use the third technique. Multi2Sim, XIOSim, Zesto, and MacSim use the third technique as well. drcachesim uses the first technique.

From an accuracy perspective, the way virtual addresses are mapped to physical addresses matter irrespective of whether multiple processes are being simulated. While this may or may not impact hit counts and other metrics, it can impact effective latency and bandwidth depending on which banks or slices of each cache level are being accessed concurrently.

3.12 Capturing Operating System Behavior

Many real-world applications like web servers are operating-system intensive; a considerable amount of the work they do is through requests made to the operating system. Therefore, while simulating such workloads, the memory requests made by the operating system code must also be simulated. However, capturing the working of the operating system is more difficult than capturing user-level programs because of the various security restrictions involved. Execution-driven simulators, employing emulators such as Intel Pin, cannot instrument the operating system. Two alternatives exist to capture the execution of kernel-mode code:

(1) Modifying the operating system itself to provide the simulator with the necessary information.
(2) Executing the application and the operating system using an emulator, and modifying the emulator to provide the necessary information to the simulator.

Most simulators do not provide support for capturing operating system behavior, owing to the considerable effort involved. The COTSon simulator works with modified operating systems, that is, the first of the above listed variants. The second variant is more popular and is adopted by gem5, Tejas, MARSSx86, and Manifold.

Even if a simulator was not designed to capture operating system behavior, it still needs to deal with all the user-kernel transitions that may occur due to many different reasons such as system calls, returning from system calls, interrupts, and exceptions. The simulator should ideally capture all such transitions and any state changes performed by the kernel, which can be challenging [57, 78].

3.13 Cache Metrics

During simulation, a cache simulator keeps track of the number of times certain events have occurred or the amount of time it took to perform a specific operation. Examples of such events include an access to the cache or a cache miss. There are many metrics that are potentially useful to users of cache simulators. At the very least, a cache simulator is expected to measure the number of misses and the miss rate. In this section, we define and categorize useful cache metrics and discuss which simulators support which metrics. Those metrics that are semantically similar are put together in the same category. All the metrics defined here are ISA-neutral, meaning that they are relevant to any ISA. The categories are the following.

Hit and Miss Metrics (HIM): This category includes all metrics that count the number of hits and/or misses. Consider the following attributes in order: Demand Read, Software Prefetch Read, Hardware Prefetch Read, Write, Hit, and Miss. Each HIM metric can be represented as a bit vector where each bit specifies whether the corresponding attribute is accounted for by the metric. For example, the metric with bit vector 100110 is the number of demand read and write accesses that hit in the cache. As another example, the metric with bit vector 111101 is the miss count considering all memory accesses. Note that some bit vectors are not valid, because they do not make sense. For example, 111100 and 000011 are invalid metrics. The first metric excludes accesses that miss or hit, which implies that it does not count anything. The second metric excludes all access types.

So far, only counting metrics have been considered. But rate metrics are as important. A rate is a ratio of two quantities. We define a rate metric by specifying two bit vectors separated by an underscore. For example, 010001_111101 is the rate of software prefetching misses to total misses.

Some architectures, such as x86, allow unaligned memory accesses and accesses that span multiple cache lines. From a binary code perspective, a single access that spans, say, two cache lines is still considered to be a single access. However, the hardware handles such accesses by splitting them into two or more accesses, one to each cache line. A cache simulator should ideally consider this issue to accurately measure HIM metrics. In addition, this issue impacts many other metrics in other categories. Some cache simulators do not do split accesses appropriately and end up underestimating the number of misses or hits. However, since most compilers strive to make all data structures aligned, the issue may not be significant.

For example, consider an access to the memory location with address 62 of size 4 bytes and assume that the size of a cache line is 64 bytes. Even though this appears to be a single memory access, the processor will issue two memory accesses. Our definitions of the metrics indicate that this access should be accounted for as two accesses. For example, if that was the only access, then a measurement of the metric 1111110 reports the value 2 rather than 1. This also applies to all other metrics defined later in this section.

Latency Metrics (LAT): The latency of a memory access is the amount of time for that access to complete; the first word of the data to be transferred has reached its destination. The starting point of the latency of an operation is dependent on the simulator. The latency metrics related to determining whether an access is a hit or miss are included in the TAG category. Each metric in this category has the following attributes: Demand Read, Software Prefetch Read, Hardware Prefetch Read, Write, Hit, Miss, and Average/Minimum/Maximum. The first five attributes are the same as those from the previous category. The last attribute, Average/Minimum/Maximum, specifies whether the metric represents the average, minimum, or maximum latency over all specified accesses. This attribute is needed, since different accesses might have different latencies. Note that we do not distinguish metrics based solely on the unit of latency (clocks, nanoseconds, etc.).

Since the last attribute has three potential values, we use a vector of base-3 digits to name metrics in this category. For example, 1000112 is the metric that represents the maximum latency for the first word to reach over all demand read accesses. Note that the amount of time it takes for all words to reach is dependent on the throughput, not just the latency. As another example, the metric 11111110 is the average latency for the first word to reach over all accesses. Note that not all vectors are valid.

Cause and Effect Metrics (CEF): These metrics count the number of hits or evictions considering both the reason that the cache line was fetched into the cache and the reason for the hit or eviction. A cache line is evicted when there is a need to make space to store another cache line. A large number of evictions typically indicates that there is little temporal locality of reference. Each metric has the following attributes: Caused by Demand Read, Caused by Software Prefetch Read, Caused by Hardware Prefetch Read, Caused by Write, Fetched by Demand Read, Fetched by Software Prefetch Read, Fetched by Hardware Prefetch Read, Fetched by Write, Accessed, Never Accessed, and Hit/Eviction. The Caused by attributes specify the the type of access that caused the eviction and the Fetched by attributes specify the type of access that caused the cache line to be fetched into the cache. The Accessed and Never Accessed attributes specify whether the line has been accessed at least once after it has been fetched and before eviction or never accessed. The last attribute specifies the event that occurred. For example, the metric 10001111011 is the number of evicted lines caused by all demand reads that were not accessed after the demand read and the metric 11100110101 is the the number of evicted lines caused by all reads that were originally fetched by prefetching and were accessed at least once. Two vectors can be used to form a rate metric.

Tag Metrics (TAG): Metrics related to accessing and maintaining tags are included here. To our knowledge, tags are only read to check whether a particular cache line exists or not. Therefore, we do not define a specific latency metric for reading from the tag array excluding the comparison. In addition, we do not define a separate latency metric for writing to the tag array. There is only a single write latency that includes both writing the data of the cache line and its tag, which is part of the LAT category. We define three subcategories for tag metrics: Counts (C), Occupancy (O), and Latency (L). The first has the following attributes: Demand Read, Software Prefetch Read, Hardware Prefetch Read, Write, TagRead, and TagWrite. This is used to count the number of tags read and/or written due to the specified memory accesses. The second subcategory has the following attributes: Total/Set and Average/Minimum/Maximum. This is used to measure the occupancy per-set or for the whole cache. We leave the definition of the average occupancy ambiguous: It is calculated over all different occupancies or over occupancies sampled at some frequency. The third subcategory includes only a single metric, namely the latency for the cache controller to determine whether a cache line exists or not.

Lockup-Free Ports Metrics (LFP): A lockup-free port is a port that does not block when a request misses in the cache and accepts further requests [45]. Many implementations have been proposed for lookup-free caches and the metrics may depend on the implementation. We define three subcategories for LFP metrics: Latency (L), Hit and Miss Counts (H), and Block Counts (B). The first has the following attributes: Demand Read, Software Prefetch Read, Hardware Prefetch Read, Write, Hit and Combine, Miss and Add, Block, and Average/Minimum/Maximum/Total. The first four attributes specify the type of access, as usual. The next three attributes specify how the request was handled. If it was an MSHR hit, then it is combined with an existing request. If it was an MSHR miss and there is a free MSHR, then it is added. Otherwise, the port blocks. The last attribute specifies the type of latency. The second subcategory has the following attributes: Demand Read, Software Prefetch Read, Hardware Prefetch Read, Write, Hit, and Miss. These are to be interpreted similarly to those from the HIM metrics but with respect to MSHRs. Two vectors can be used to form a rate metric. The third subcategory has three metrics: the number of times the port was blocked (T), the total number of cycles the port was blocked (C), and the average number of cycles the port was blocked (A).

Coherence Metrics (COH): Coherence is the property that any memory location is observed by all processors to the have the same content at any point in time and that every written value is eventually observed unless overridden by a succeeding write. This requires writes to the same data to be serialized. Two commonly used mechanisms are used to implement cache coherence: snooping and directory-based. We categorize COH metrics into five subcategories: Snooping (SNO), Directory (DIR), State Transition Latency (STL), State Transition Count (STC), and False Sharing (FAS). The following metrics are defined in SNO: number of snoops that hit in the snoop filter (HIT), number of snoops that miss in the snoop filter (MIS), number of invalidations due to coherence (INV), number of write-backs due to coherence (WRB), and number of updates due to coherence (UPD). Note that if there is no snoop filter, then HIT would be zero and MIS would be equal to the total number of snoops. Both HIT and MIS are per snoop filter. The metrics INV, WRB, and UPD also exist under DIR. In addition, DIR contains metrics with the following attributes: Demand Read, Software Prefetch Read, Hardware Prefetch Read, Write Miss, Write Hit, Evict, Directory Hit, and Directory Miss. The STL metrics have the following attributes: Current State, Next State, and Average/Minimum/Maximum. The STC metrics have the following attributes: Access Type, State Transition Reason, Current State, and Next State. The exact STL and STC metrics depend on the coherence states and protocol being used.

False sharing is a memory access pattern in which two cores are frequently modifying different pieces of the same cache line in a small duration of time. This can lead to significant performance degradations due to coherence. Therefore, measuring false sharing is important. By definition, false sharing occurs when there are multiple processors and so all FAS metrics are invalid in a single-processor simulation. There are several ways in which false sharing can be precisely defined [15]. Most existing simulators do not support any false sharing metrics and so there is no need at this point to define a metric for each possible definition of false sharing. We define in this subcategory two metrics and leave the kind of false sharing being measured as an implementation detail. The first is the number of lines exhibiting false sharing (C). The second is the number of invalidations caused by false sharing (I).

Bus Metrics (BUS): A bus is a collection of wires that connect two or more components together and potentially an arbitration logic. A cache is typically connected by a number of buses to other caches and processors. A packet is a collection of data that is transferred on the bus as one unit. In this category, the following metrics are defined: read throughput, write throughput, total time during which the cache is idle, total time during which the cache is servicing requests, longest/shortest continuous duration during which the cache is idle/non-idle, average idle/non-idle duration, total number of packets, total size of all packets, number of coherence-related packets, and size of coherence-related packets.

Translation Lookaside Buffer Metrics (TLB): A TLB is a special cache used to speed up translation of virtual addresses to physical addresses. TLB metrics are important not only for architecture designers but also for system and application developers to detect access patterns that cause a lot of TLB misses, resulting in performance degradation. We define several subcategories for TLB metrics: Access Counts (A), Flush Counts (F), and Latency (L). The first subcategory has the same attributes as HIM. The second includes two metrics: the number of entries that have been flushed from the TLB (E) and the number of times the complete TLB was flushed (C). The third subcategory has the same attributes as LAT. The APT metrics defined below also constitute a subcategory under TLB.

Speculative Execution Metrics (SPE): Each metric in this category has all the attributes from HIM prepended by the following three metrics: Speculative Demand Read, Speculative Software Prefetch Read, and Speculative Hardware Prefetch Read. Two vectors can be used to form a rate metric.

Area, Power, and Thermal Metrics (APT): A cache simulator might include an area, power, or thermal model. These aspects are related and can be further categorized into four subcategories: Area (A), Energy (E), Power (P), and Thermal (T). There is only one metric in A, namely Area. The following metrics are defined in the Thermal subcategory: Average Temperature (AP), Minimum Temperature (MP), and Peak Temperature (XP). The E metrics have the following attributes: Subthreshold Leakage with Power Gating, Subthreshold Leakage, Gate Leakage, and Dynamic. The energy consumption is usually estimated by multiplying the estimated power by the execution time or duration. The P metrics have the following attributes: Subthreshold Leakage with Power Gating, Subthreshold Leakage, Gate Leakage, Dynamic, and Average/Minimum/Peak. Typically, leakage power is constant with respect to a particular cache circuit design and technology. The dynamic power changes over time depending on the cache activity. The same APT metrics are defined under the TLB category.

Detailed Metrics (DET): Most of the metrics defined so far can be defined at a finer granularity rather than over the whole program execution or the whole cache. To avoid cluttering all the categories, we decided to define a specific category for such metrics. There are many ways in which DET metrics can be defined. DET metrics can be defined based on HIM metrics by reducing their scope to per source code function (SCF), per instruction (INS), per source code line (SCL), per hardware thread for shared caches (HTH), per software thread (STH), per cache port for multiport caches (POR), per bank for multibank caches (BAN), per cache set (SET), per cache block (BLK), or per cache line state (STA). Another way DET metrics can be defined based on HIM metrics is by splitting the miss attribute into four attributes corresponding to the four miss types (MIS): compulsory, capacity, conflict, and coherence. DET metrics can be defined based on LFP metrics by considering the cause of blocking (LFP). Coherence metrics can be further classified depending on the source of the access (I/O device, GPU, or local or remote processor). Many other DET metrics can be defined similarly. In this article, we name only those DET metrics that we use here.

3.13.1 Cache Metrics Naming Convention. There is a large number of cache metrics. To avoid confusion and ambiguity, we propose a convenient naming convention. Each metric is named by starting with the category shorthand name suffixed by an underscore, followed by the subcategory name suffixed by an underscore (if applicable), and terminated by the name (bit vector) of the metric if there is more than one metric in the same category or subcategory. For example, the miss count over all memory accesses is named HIM_111101, the maximum latency for the first word to reach over all demand read accesses is named LAT_1000112, the minimum total occupancy is TAG_O_01, and the number of lines exhibiting false sharing is COH_FAS_C. In addition, to

Table 3. Cache Simulators and Supported Output Metrics

Simulator	Output Metrics
Cachegrind	HIM, DET_HIM_SCF, DET_HIM_INS.
Dinero IV	HIM, DET_HIM_MIS.
CASPER	HIM, COH_SNO.
CMP$im	HIM, COH_?, ?.
Moola	HIM, LAT, COH_?, COH_FAS.
gem5	HIM, LAT, CEF, TAG_C, TAG_O, TAG_L, LFP_L, LFP_H, LFP_B, COH_SNO, BUS, TLB_A, TLB_F, TLB_L, DET_HIM_INS, DET_HIM_HTH, DET_HIM_STH, DET_LFP, DET_LAT_INS, DET_LAT_HTH, DET_LFP_INS, DET_LFP_HTH.
Sniper	HIM, LAT, CEF, LFP_H, COH_SNO, COH_STC, TLB_A, DET_HIM_SCF, DET_HIM_INS, DET_HIM_STA, DET_CEF_STA.
Tejas	HIM, CEF, COH_DIR, TLB_A.
ZSim	HIM.
MultiCacheSim	HIM, COH_SNO, COH_STC.
drcachesim	HIM, TLB_A.
MARSSx86	HIM, TLB_A, TLB_L.
Multi2Sim	HIM, CEF, TLB_A, TLB_F.
SimpleScalar	HIM, CEF, TLB_A.
ESESC	HIM, LAT, LFP_H, COH_SNO, COH_DIR, COH_STC, TLB_A, TLB_F, TLB_L.
Graphite	HIM, CEF, COH_SNO, COH_STC, DET_HIM_MIS.
HORNET	HIM, COH_DIR.
SlackSim	HIM, CEF.
Manifold	HIM, TLB_A.
vCSIMx86	HIM, DET_HIM_MIS.
pycachesim	HIM.
SMPCache	HIM.
SiNUCA	HIM, CEF.
COTSon	HIM, COH_SNO, COH_DIR, COH_STC.
McSimA+	HIM, CEF, COH_DIR, TLB_A.
XIOSim	HIM, CEF, LFP_H.
Zesto	HIM, CEF, LFP_H.
MacSim	HIM, LAT, BUS, TLB_APT_P, APT_P.

The "?" symbol indicates that we were not able to find the data.

distinguish between different caches, the cache level and name is prepended to the metric name. For example, C1_L1_D_HIM_111101 is the miss count in the data cache in L1 of core 1. Access latency requires specifying two parties; the requester and the server. Using this naming convention, metrics that involve interactions between a cache and other components can be defined using two or more intra-cache metrics.

3.13.2 Cache Simulators Output Metrics. Table 3 shows which simulators offer built-in support for which metrics. We show supported metrics at the level of subcategories. If a simulator supports at least one metric within a subcategory (or category in case there are no subcategories), then it is considered to support that subcategory.

Some simulators might support a particular metric that is similar to but does not exactly match a metric defined here. For example, Cachegrind treats an access that crosses a cache line as a single access, which is slightly different from the HIM metrics. However, even in these cases, we do still consider that the simulator supports the metric, because usually in such cases, it is easy to modify the source code to rectify the way a metric is measured if required.

Our categorization helps in estimating the amount of source code changes required to support a new metric. If a simulator supports some metrics of a subcategory, then most or all other metrics in that subcategory can probably be supported with minimal code changes. For example, if HIM_111101 and HIM_111110 are supported, then HIM_111111, HIM_111001, and HIM_000101 are among the metrics that can be easily supported. However, metrics from other categories or subcategories probably require a significant amount of work to support.

APT models are usually developed separately and integrated into the simulator. Therefore, we did not include APT metrics in Table 3. Commonly used APT models include the McPAT power, area, and timing modeling framework [47] and the HotSpot thermal model [77]. McPAT includes the Cacti-P modeling tool for SRAM, DRAM, and 3D stacked DRAM caches. Any of the simulators, with some effort, can be integrated with any APT model. This can be done either by having the simulator take input from or provide input to the APT model or running the APT model during simulation. gem5, Sniper, Tejas, Multi2Sim, ESESC, Graphite, Manifold, COTSon, and XIOSim are available with integrated McPAT. ESESC and Manifold are available with integrated HotSpot.

3.13.3 From Statistical Metrics to Traces. All the metrics defined so far are statistical. That is, they are measured over a number of events or operations. For example, the average temperature is computed by sampling the temperature at some frequency, adding up all the samples and dividing them by their number. With some source code changes, we can obtain a trace of temperatures by emitting all the sampled temperatures rather than the average.

4 VALIDATION OF CACHE SIMULATORS

Many of the simulators have already been validated either by the original authors themselves or by others. We discuss these attempts at validating some of the simulators, then argue that some common methods to validate academic simulators are deficient, and present a new understanding of simulator validation. While our focus is on cache simulators and relevant metrics, this section applies to architectural simulators in general.

4.1 Introduction

We briefly discuss how some of the simulators were validated. Cachegrind was validated [58] using a suite of single-threaded benchmarks, and the target platform consisted of a single-core 32-bit x86 processor with two levels of caches. The exact configuration of the caches was not specified, but it was fixed for all the experiments. Validation was done against hardware performance counters and the validated metrics were the total number of L1 and L2 misses. The exact microarchitecture and hardware performance counters used were not specified. The experiments showed that Cachegrind underestimates L1 and L2 misses by 3–62%. Remarkably, the authors documented numerous shortcomings with their simulator, of which users should be aware before using it.

CMP$im was validated [38] using a suite of single-threaded benchmarks, and the target platform consisted of a multicore 32-bit x86 processor with three levels of caches. Since the benchmarks are single threaded, only one core was used during each experiment (multiple cores were used for other experiments not related to validation). The exact configuration of the caches was specified, and it was fixed for all the experiments. Validation was done against another simulator that the authors did not name. The validated metric was the number of misses per 1000 instructions (MPKI), although it was not specified whether that referred to the total number of misses including all cache levels or just one particular level. It was concluded that CMP$im is within 13% of the other simulator.

gem5 was validated [20] using a suite of multithreaded benchmarks, and the target platform included a dual-core ARM Cortex-A9 (a 32-bit out-of-order, speculative, superscalar processor with two levels of caches). Validation was done against a real system and the metric used was the wall-clock execution time. gem5 has two cache subsystem models (Classic and Ruby); the model used was not specified. The experiments showed that gem5 may overestimate execution time by as much as 16.12% and underestimate it by as much as 17.94%. Further experiments were conducted to investigate the mismatch, and it was concluded that main memory model is the reason for it. Another work [5] used a collection of single-threaded benchmarks compiled to 32-bit or 64-bit x86 binaries (depending on the simulator). The target platform consisted of an x86 processor with three levels of caches. The exact configuration of the caches was fixed for all the experiments. Validation was done against hardware performance counters and the validated metrics were the IPC, data L1 misses, data L3 misses, and branch mispredictions. The exact microarchitecture and hardware performance counters used were specified. The gem5 memory model used was not specified. The experiments showed that gem5 may overestimate miss rates by up to 7.5× and underestimate it by as much as 0.1×. The authors speculated the reasons for the observed significant mismatch. A more recent work [83] proposes a methodology to identify significant sources of error in a simulator. The methodology was applied to the gem5 simulator to validate it against two real ARM processors using a collection of single- and multi-threaded benchmarks. Finally, the gem5 measurement of a number metrics was compared against Intel Haswell (and other simulators) using single-threaded benchmarks [5]. These metrics include IPC, the number of L1 data cache misses, the number of L3 cache misses, and the number of mispredicted conditional branches.

Similar efforts have been made to validate the following simulators: Moola [76], Tejas [73], Multi2Sim [5, 80], Sniper [3, 5, 21, 22], SiNUCA [8], COTSon [11], ZSim [5, 72], MARSSx86 [5], and McSimA+ [4]. All other simulators have not been validated against either hardware performance counters or other simulators to the best of our knowledge.

Many of these simulators are claimed to be validated (not necessarily by the authors), because the experiments show that some of the cache-related output metrics are close enough. In particular, the output metrics that have been validated include only the number or rate of caches misses or the execution time and considering only a single target platform. However, we have discussed throughout this article that there are many metrics and many possible configurations with complex interactions that are dependent on the programs being simulated. But on a more fundamental level, after looking at all of these simulators, we could not find an explicit unambiguous definition of what makes a simulator validated or what it means to validate a simulator and why that matters. This can be easily seen by noticing that different simulators were validated using different benchmarks, different configurations, different output metrics, and different methods.

4.2 Validation Techniques

We identify four methods of validation:

- Validation against hardware performance counters.
- Validation against another simulator.
- Validation against analytical models.
- Validation against a model written in a hardware description language (HDL).

Some architects and researchers believe that academic simulators can be used to evaluate new ideas in computer architecture without having to validate them against real hardware, while others insist on validation against real hardware. We would like to discuss the first method in detail. In the first method, only the output metrics that have corresponding performance events on the target processor can be validated. Also, the input programs used for validation should run on both the target processor and the simulator (in case of a non-trace-driven simulation). These limitations are not issues and do not undermine the validity of this method of validation by themselves. However, there are several major issues with this method.

First, hardware performance events are very difficult to use correctly. The same event (e.g., L1 cache miss) may mean different things across processors from different vendors and even the same vendor [54]. It is absolutely important to precisely specify the processor to do the validation against and determine the exact meaning of the performance events to be used for validation on that processor. In addition, it is important to correctly configure the performance monitoring unit (PMU) to count the right events. The issue does not end there; after selecting an event to be used for validation and determining what it means precisely, the event may not be deterministic [87]. That is, the counts of an event for multiple runs of the same program may be substantially different. Even if one event is deterministic on a particular processor, it may not be so on another processor. Some events can be made more deterministic by avoiding certain instructions. Overall, the hardware performance counters are not necessarily accurate and so using them for any meaningful validation can be very tricky.

Second, let us assume a perfectly accurate performance event was used to validate a particular metric, and it turns out there is some amount of mismatch. How should that mismatch be interpreted? If the target processor was open source and the simulator supports all of its features, then it would probably mean that there is a bug either in the simulator or in the way the event was counted. But if the target architecture was not open source (e.g., x86), it would be very difficult or impossible to determine the reason for the mismatch. One could speculate, but it may not be possible to ascertain for sure. It could be because of an unknown feature that the simulator does not support or was not intended to support.

Third, assume a perfectly accurate performance event was used to validate a particular metric, and it turns out there is no or minimal mismatch. Can we conclude that the simulator is perfectly or highly accurate? It could be the case that the features of the processor that have been unknowingly abstracted away by the simulator have not been triggered by the program being simulated. Or it could be the case that there are multiple bugs in the simulator or errors in experimentation that canceled each other's effects, resulting in an apparently high simulation accuracy. Moreover, similarity in microarchitecture-dependent metrics does not necessarily mean that the microarchitecture-independent characteristics were captured accurately by the simulator [35]. For example, the simulator and the PMU may report nearly identical data cache miss rates, yet the memory access patterns exhibited by individual instructions may be different.

It is difficult to overcome these problems in general. One might use the errors that were observed from validation against real hardware to basically "fit" the simulator to the hardware so that it measures certain metrics more accurately and report those measurements. This leads to a pitfall called over-generalization in which users of simulators think that the simulator will be that much accurate for all inputs and configurations. This issue is particularly severe in simulators that support multiple ISA families [67], yet such simulators are usually only validated against a single ISA. Another pitfall is to assume that the empirical mismatches will trend, meaning that the errors will always be similar or bounded. The issue with this assumption is that when a researcher attempts to evaluate a new idea or technique, it may amplify the errors depending how it interacts with the causes or sources of mismatches. These and other pitfalls are discussed in more detail in [61].

In summary, using the first method, it is very difficult to interpret mismatches and make any useful (perhaps, slightly broad) statements about the accuracy of the simulator.[6] Some authors of simulators emphasize that their simulator is not intended to accurately model real processors and that it simulates abstracted models of platforms. This is convenient for academic simulators. From this point of view, it may make more sense to use the second method and validate academic simulators against other academic or non-academic simulators that were validated either against other simulators in turn, against real hardware, or gained the trust of the research community through extensive usage over time. It is important that the two simulators target the same area in computer architecture. For example, some of the simulators considered in this survey were validated against gem5, which is the most detailed and one of the most cited simulators for multicore processors. It would be a lot more sensible to reason about mismatches using this method. In this case, the goal of validation would be to make the measurements of metrics of one simulator to be as close as possible to another well-known simulator. Causes for mismatches include bugs in either simulator or differences in abstractions. It is possible to determine the exact causes by knowing the internals of each simulator. A popular myth is that the so called cycle-accurate simulators are always more accurate with respect to real hardware and therefore one should always validate against such simulators. But this is just a myth [86]. Thus, the extremely slow cycle-accurate simulators may not be the best for validation.

The third method for validating a simulator is by using analytical models of cache subsystems [13, 52, 70, 88]. To our knowledge, analytical models were never used for this purpose. Analytical models are generally easier to build and much faster than simulators. However, they tend to be less accurate. Therefore, the simulator being validated should produce results that are at least as accurate as the analytical models used. In case of a significant discrepancy, there are three possibilities. First, the simulator is more accurate than the model, which can justify its usefulness, considering its inferior performance. Second, the model is more accurate than the simulator and therefore, the simulator should be improved. Third, the model is more accurate than the simulator, but the simulator was not designed to capture the aspects that led to the discrepancy. An investigation is required to determine which of the possibilities is the case.

There are different types of analytical models [26], some of which require empirical results to build. These can be obtained through native execution of benchmarks, dynamic binary instrumentation–(DBI) based tools, or simulations using other already validated simulators. The more sophisticated the analytical models are, the more reliable the simulator becomes.

The fourth and least common method of validation is validation against an HDL model. This method makes sense when the goal of the simulator is to simulate a particular HDL model. But in general, this method is rarely used, because HDL models of some popular architectures are difficult to obtain.

4.3 Guidelines for Validating and Using Simulators

An important question common to all methods is what metrics to validate. Some metrics are more fundamental than others, and validating them would be more useful. For example, plotting misses per thousand instructions (MPKI) over time and comparing it can shed light on the causes of divergence in measurement during execution, in contrast to using total number of misses. Even if some metrics are related to others, all metrics must be validated to make sure they are reliable [30]. If a researcher wants to use a metric that has never been validated before, then it is recommended that the researcher validates the metric. Alternatively, an argument should be presented that the important relevant effects are modeled in the simulator being used for evaluation of the proposed technique.

[6]However, it can be used to reverse-engineer certain features of a particular processor [1] and then provide a realistic built-in support for them [3, 74] in modeling and/or configuration.

Academic simulators (including all those considered in this article) should be designed for the purpose of evaluating new ideas rather than to implement identical or close models to real processors. Thus, every simulator will have its own abstractions, supported features, and shortcomings, all of which have to be clearly specified in the documentation. The purpose of validation should be to determine the impact of these abstractions and shortcomings on the measurements and potentially highlight issues or bugs in the simulator that have never been thought of or discovered before. Validation should help users understand the intended purposes of the simulator and guide them on how to use it correctly. Validation should be used to establish the correctness of the implementation and the correct usage of the academic simulator rather than its fidelity to real hardware. Perhaps, the word "correctness" can be used as an alternative to or instead of "accuracy" in Section 2.

When selecting a simulator to validate against, one should either use a dedicated simulator or a simulator that has been extensively used. This reduces the chances that the chosen simulator has many bugs. Note that dedicated simulators are typically much smaller in terms of lines of code compared to general architectural simulators. Validation against real hardware can be used to measure the extent of abstraction in the simulator or as some kind of assurance to its correctness by revealing issues when unexpected mismatches are observed. This approach was followed in Reference [30]. But one has to be careful when interpreting the results and not make invalid conclusions about accuracy. When measurements of particular metrics are nondeterministic [6], statistical methods should be used [19, 89].

When a researcher wants to choose one of the simulators to evaluate a new idea, they should first determine all the aspects or components of a system that are affected by implementing that idea or that have an effect on it. If the proposed technique is a new cache replacement policy, then a pure application-level cache simulator may be sufficient. If one needs investigate how operating system code uses the instruction caches, then a full system simulator needs to be used. A technique that aims at improving cache access latency requires a cycle-level simulator to be evaluated. At the same time, an overly simplistic simulator may lead to erroneous conclusions [79, 92].

5 CONCLUSION

This survey has provided a detailed discussion on 28 CPU cache simulators, including state-of-the-art, recent, or popular simulators. A significant portion of the information presented could only be found by examining the source code of these simulators. The type, level, mode, and scope of a simulator are major design characteristics that can significantly influence the suitability of a simulator for a particular research project. There is also a large number of detailed design aspects of caches and cache hierarchies, such as the number of cache levels, replacement policies, write policies, and coherence protocols. We have discussed the extent to which each of the simulators has built-in support or can support each of these aspects. Among the major issues that are common to all simulators that has been highlighted in this survey include very limited and poorly documented support for modern ISA (e.g., x86) extensions, cache addressing techniques, sophisticated address translation cache hierarchies, and the impact of page walks on the caches, with respect to modern processors. Another major issue is that all simulators were poorly validated, because existing validation works consider only a very limited set of metrics and configurations compared to what is supported by the simulator. We hope that this survey contributes toward more effort being spent toward addressing these issues in the future.

REFERENCES

[1] Andreas Abel and Jan 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)*. IEEE, 141–142.

[2] Jaume Abella and Antonio González. 2006. Heterogeneous way-size cache. In *Proceedings of the 20th Annual International Conference on Supercomputing (ICS'06).* ACM, New York, NY, 239–248. DOI: https://doi.org/10.1145/1183401.1183436

[3] Almutaz Adileh, Cecilia González-Álvarez, Juan Miguel De Haro Ruiz, and Lieven Eeckhout. 2019. Racing to hardware-validated simulation. In *Proceedings of the 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'19).* IEEE, 58–67.

[4] Jung Ho Ahn, Sheng Li, O. Seongil, and Norman P. Jouppi. 2013. McSimA+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling. In *Proceedings of the 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'13).* IEEE, 74–85.

[5] Ayaz Akram and Lina Sawalha. 2019. A survey of computer architecture simulation techniques and tools. *IEEE Access* 7 (2019), 78120–78145.

[6] Alaa R. Alameldeen and David A. Wood. 2006. IPC considered harmful for multiprocessor workloads. *IEEE Micro* 26, 4 (Jul. 2006), 8–17. DOI: https://doi.org/10.1109/MM.2006.73

[7] Muhammad Aleem, Muhammad Arshad Islam, and Muhammad Azhar Iqbal. 2016. A comparative study of heterogeneous processor simulators. *Int. J. Comput. Appl.* 148, 12 (2016).

[8] Marco Antonio Zanata Alves, Carlos Villavieja, Matthias Diener, Francis Birck Moreira, and Philippe Olivier Alexandre Navaux. 2015. Sinuca: A validated micro-architecture simulator. In *Proceedings of the 2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC'15), Proceedings of the 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS'15), and Proceedings of the 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS'15).* IEEE, 605–610.

[9] Hari Angepat, Derek Chiou, Eric S. Chung, and James C. Hoe. 2014. FPGA-accelerated simulation of computer systems. *Synthesis Lectures on Computer Architecture* 9, 2 (2014), 1–80.

[10] Ehsan K. Ardestani and Jose Renau. 2013. ESESC: A fast multicore simulator using time-based sampling. In *Proceedings of the High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on.* IEEE, 448–459.

[11] Eduardo Argollo, Ayose Falcón, Paolo Faraboschi, Matteo Monchiero, and Daniel Ortega. 2009. COTSon: Infrastructure for full system simulation. *ACM SIGOPS Operat. Syst. Rev.* 43, 1 (2009), 52–61.

[12] Todd Austin, Eric Larson, and Dan Ernst. 2002. SimpleScalar: An infrastructure for computer system modeling. *Computer* 35, 2 (2002), 59–67.

[13] Wenlei Bao, Sriram Krishnamoorthy, Louis-Noel Pouchet, and P. Sadayappan. 2017. Analytical modeling of cache behavior for affine programs. *Proc. ACM Program. Lang.* 2 (2017), 32:1–32:26.

[14] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH Comput. Arch. News* 39, 2 (2011), 1–7.

[15] William J. Bolosky and Michael L. Scott. 1993. False sharing and its effect on shared memory performance. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV).* 57–71.

[16] Gustaf Borgström, Andreas Sembrant, and David Black-Schaffer. 2017. Adaptive cache warming for faster simulations. In *Proceedings of the 9th Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools.* ACM, 1.

[17] Derek Bruening and Saman Amarasinghe. 2004. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation.* Ph.D. Dissertation. Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science.

[18] Martin Burtscher and Ilya Ganusov. 2004. Automatic synthesis of high-speed processor simulators. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture.* IEEE Computer Society, 55–66.

[19] Anastasiia Butko, Rafael Garibotti, Luciano Ost, Vianney Lapotre, Abdoulaye Gamatie, Gilles Sassatelli, and Chris Adeniyi-Jones. 2015. A trace-driven approach for fast and accurate simulation of manycore architectures. In *Proceedings of the 2015 20th Asia and South Pacific Design Automation Conference (ASP-DAC'15).* IEEE, 707–712.

[20] Anastasiia Butko, Rafael Garibotti, Luciano Ost, and Gilles 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).* IEEE, 1–7.

[21] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. 2014. An evaluation of high-level mechanistic core models. *ACM Trans. Arch. Code Optim.* 11, 3 (2014), 28.

[22] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. 2011. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of the 2011 International Conference on High Performance Computing, Networking, Storage and Analysis (SC'11).* IEEE, 1–12.

[23] Jianwei Chen, Murali Annavaram, and Michel Dubois. 2009. SlackSim: A platform for parallel simulations of CMPs on CMPs. *ACM SIGARCH Comput. Arch. News* 37, 2 (2009), 20–29.

[24] Computer Sciences Department, Harold W. Cain, Kevin M. Lepak, On A. Schwartz, and Mikko H. Lipasti. 2002. Precise and accurate processor simulation. In *Proceedings of the 5th Workshop on Computer Architecture Evaluation using Commercial Workloads.* 13–22.

[25] Jan Edler. 1998. Dinero IV trace-driven uniprocessor cache simulator. Retrieved from http://www.cs.wisc.edu/~markhill/DineroIV/.

[26] Lieven Eeckhout. 2010. Computer architecture performance evaluation methods. *Synth. Lect. Comput. Arch.* 5, 1 (2010), 1–145.

[27] Jakob Engblom. 2008. Is cycle accuracy a bad idea? Retrieved July 20, 2017 from http://jakob.engbloms.se/archives/153.

[28] Qi Guo, Tianshi Chen, Yunji Chen, and Franz Franchetti. 2016. Accelerating architectural simulation via statistical techniques: A survey. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 35, 3 (2016), 433–446.

[29] Xuan Guo and Robert Mullins. 2019. Fast TLB simulation for RISC-V systems. *arXiv preprint arXiv:1905.06825* (2019).

[30] 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 the 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'14).* IEEE, 13–22.

[31] Julian Hammer. 2015. pycachesim. Retrieved from https://github.com/RRZE-HPC/pycachesim.

[32] Mohammad Shihabul Haque, Andhi Janapsatya, and Sri Parameswaran. 2009. Susesim: A fast simulation strategy to find optimal l1 cache configuration for embedded systems. In *Proceedings of the 7th IEEE/ACM International Conference on Hardware/software Codesign and System Synthesis.* ACM, 295–304.

[33] Kim Hazelwood. 2011. Dynamic binary modification: Tools, techniques, and applications. *Synth. Lect. Comput. Arch.* 6, 2 (2011), 1–81.

[34] John L. Henning. 2000. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer* 33, 7 (2000), 28–35.

[35] Kenneth Hoste and Lieven Eeckhout. 2007. Microarchitecture-independent workload characterization. *IEEE Micro* 27, 3 (2007).

[36] Ravi Iyer. 2003. On modeling and analyzing cache hierarchies using CASPER. In *Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems (MASCOTS'03).* IEEE, 182–187.

[37] Bruce Jacob and Trevor Mudge. 1998. Virtual memory in contemporary microprocessors. *IEEE Micro* 18, 4 (Jul. 1998), 60–75. DOI: https://doi.org/10.1109/40.710872

[38] Aamer Jaleel, Robert S. Cohn, Chi-Keung Luk, and Bruce Jacob. 2008. CMP$im: A pin-based on-the-fly multi-core cache simulator. In *Proceedings of the 4th Annual Workshop on Modeling, Benchmarking and Simulation (MoBS'08, co-located with ISCA'08).* 28–36.

[39] Chuntao Jiang, Zhibin Yu, Lieven Eeckhout, Hai Jin, Xiaofei Liao, and Chengzhong Xu. 2015. Two-level hybrid sampled simulation of multithreaded applications. *ACM Trans. Archit. Code Optim.* 12, 4, Article 39 (Nov. 2015), 25 pages. DOI: https://doi.org/10.1145/2818353

[40] Svilen Kanev, Gu-Yeon Wei, and David Brooks. 2012. XIOSim: Power-performance modeling of mobile x86 cores. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design.* ACM, 267–272.

[41] Hui Kang and Jennifer L. Wong. 2013. vCSIMx86: A cache simulation framework for x86 virtualization hosts. Stony Brook University (2013).

[42] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, et al. 2018. FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In *Proceedings of the 45th Annual International Symposium on Computer Architecture.* IEEE Press, 29–42.

[43] A. C. J. Kienhuis. 1999. *Design Space Exploration of Stream-based Dataflow Architectures.* Ph. D. Dissertation. Delft University of Technology.

[44] Hyesoon Kim, Jaekyu Lee, Nagesh B Lakshminarayana, Jaewoong Sim, Jieun Lim, and Tri Pho. 2012. Macsim: A cpu-gpu heterogeneous simulation framework user guide. Georgia Institute of Technology.

[45] David Kroft. 1981. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the International Symposium on Computer Architecture* (1981).

[46] Kevin M. Lepak, Harold W. Cain, and Mikko H. Lipasti. 2003. Redeeming IPC as a performance metric for multithreaded programs. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03).* IEEE Computer Society, Washington, DC, 232–. http://dl.acm.org/citation.cfm?id=942806.943855

[47] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of*

the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42). ACM, New York, NY, 469–480. DOI: https://doi.org/10.1145/1669112.1669172

[48] Jinzhao Liu, Yuezhi Zhou, and Di Zhang. 2016. Transim: A simulation framework for cache-enabled transparent computing systems. *IEEE Trans. Comput.* 65, 10 (2016), 3171–3183.

[49] Gabriel H. Loh, Samantika Subramaniam, and Yuejian Xie. 2009. Zesto: A cycle-level simulator for highly detailed microarchitecture exploration. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'09)*. IEEE, 53–64.

[50] Brandon Lucia. 2010. MultiCacheSim. Retrieved from https://github.com/blucia0a/MultiCacheSim.

[51] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM Sigplan Notices*, Vol. 40. ACM, 190–200.

[52] Hao Luo, Pengcheng Li, and Chen Ding. 2017. Thread data sharing in cache: Theory and measurement. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 103–115.

[53] Clémentine Maurice, Nicolas Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. Reverse engineering intel last-level cache complex addressing using performance counters. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID'15)*. Springer-Verlag, New York, NY, 48–65. DOI: https://doi.org/10.1007/978-3-319-26362-5_3

[54] John McCalpin. 2013. Notes on the mystery of hardware cache performance counters. Retrieved from https://sites.utexas.edu/jdm4372/2013/07/14/notes-on-the-mystery-of-hardware-cache-performance-counters/.

[55] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. 2010. Graphite: A distributed parallel simulator for multicores. In *Proceedings of the 2010 IEEE 16th International Symposium on High Performance Computer Architecture (HPCA'10)*. IEEE, 1–12.

[56] Harit Modi, Lawrence Spracklen, Yuan Chou, and Santosh G. Abraham. 2005. Accurate modeling of aggressive speculation in modern microprocessor architectures. *Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. 75–84.

[57] Satish Narayanasamy, Cristiano Pereira, Harish Patil, Robert Cohn, and Brad Calder. 2006. Automatic logging of operating system effects to guide application-level architecture simulation. *ACM SIGMETRICS Perf. Eval. Rev.* 34, 1 (2006), 216–227.

[58] Nicholas Nethercote. 2004. *Dynamic Binary Analysis and Instrumentation*. Technical Report. University of Cambridge, Computer Laboratory.

[59] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan Notices*, Vol. 42. ACM, 89–100.

[60] Siddharth Nilakantan, Scott Lerner, Mark Hempstead, and Baris Taskin. 2015. Can you trust your memory trace? A comparison of memory traces from binary instrumentation and simulation. In *Proceedings of the 2015 28th International Conference on VLSI Design (VLSID'15)*. IEEE, 135–140.

[61] Tony Nowatzki, Jaikrishnan Menon, Chen-Han Ho, and Karthikeyan Sankaralingam. 2014. gem5, gpgpusim, mcpat, gpuwattch, "your favorite simulator here" considered harmful. University of Wisconsin - Madison.

[62] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. 1997. RSIM: An execution-driven simulator for ILP-based shared-memory multiprocessors and uniprocessors. *IEEE Techn. Comm. Comput. Arch. Newslett.* (1997).

[63] Ramesh Panwar and David Rennels. 1995. Reducing the frequency of tag compares for low power I-cache design. In *Proceedings of the 1995 International Symposium on Low Power Design (ISLPED'95)*. ACM, New York, NY, 57–62. DOI: https://doi.org/10.1145/224081.224092

[64] Avadh Patel, Furat Afram, and Kanad Ghose. 2011. Marss-x86: A qemu-based micro-architectural and systems simulator for x86 multicore processors. In *Proceedings of the 1st International Qemu Users Forum*. 29–30.

[65] Ardavan Pedram, David Craven, and Andreas Gerstlauer. 2009. Modeling cache effects at the transaction level. In *Proceedings of the International Embedded Systems Symposium (IESS'09)*. Springer, 89–101.

[66] Michael Pellauer, Michael Adler, Michel Kinsy, Angshuman Parashar, and Joel Emer. 2011. HAsim: FPGA-based high-detail multicore simulation using time-division multiplexing. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE, 406–417.

[67] Aashish Phansalkar, Ajay Joshi, and Lizy K. John. 2007. Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA'07)*. ACM, New York, NY, 412–423. DOI: https://doi.org/10.1145/1250662.1250713

[68] Moinuddin K. Qureshi, David Thompson, and Yale N. Patt. 2005. The V-Way cache: Demand based associativity via global replacement. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA'05)*. IEEE Computer Society, Washington, DC, USA, 544–555. DOI: https://doi.org/10.1109/ISCA.2005.52

[69] Pengju Ren, Mieszko Lis, Myong Hyon Cho, Keun Sup Shim, Christopher W Fletcher, Omer Khan, Nanning Zheng, and Srinivas Devadas. 2012. Hornet: A cycle-level multicore simulator. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 31, 6 (2012), 890–903.

[70] Jasmine Madonna Sabarimuthu and T. G. Venkatesh. 2018. Analytical miss rate calculation of L2 cache from the RD profile of L1 cache. *IEEE Trans. Comput.* 67, 1 (2018), 9–15.

[71] Daniel Sanchez and Christos Kozyrakis. 2010. The ZCache: Decoupling ways and associativity. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43)*. IEEE Computer Society, Washington, DC, 187–198. DOI: https://doi.org/10.1109/MICRO.2010.20

[72] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 475–486.

[73] Smruti R. Sarangi, Rajshekar Kalayappan, Prathmesh Kallurkar, Seep Goel, and Eldhose Peter. 2015. Tejas: A java based versatile micro-architectural simulator. In *Proceedings of the 2015 25th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS'15)*. IEEE, 47–54.

[74] Yukinori Sato and Toshio Endo. 2017. An accurate simulator of cache-line conflicts to exploit the underlying cache performance. In *European Conference on Parallel Processing*. Springer, 119–133.

[75] André Seznec. 1993. A case for two-way skewed-associative caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA'93)*. ACM, New York, NY, 169–178. DOI: https://doi.org/10.1145/165123.165152

[76] Charles F. Shelor and Krishna M. Kavi. 2015. Moola: Multicore cache simulator. In *Proceedings of the International Conference on Computers and Their Applications.*

[77] Kevin Skadron, Mircea R. Stan, Karthik Sankaranarayanan, Wei Huang, Sivakumar Velusamy, and David Tarjan. 2004. Temperature-aware microarchitecture: Modeling and implementation. *ACM Trans. Archit. Code Optim.* 1, 1 (Mar. 2004), 94–125. DOI: https://doi.org/10.1145/980152.980157

[78] Alex Skaletsky, Tevi Devor, Nadav Chachmon, Robert Cohn, Kim Hazelwood, Vladimir Vladimirov, and Moshe Bach. 2010. Dynamic program analysis of microsoft windows applications. In *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS'10)*. IEEE, 2–12.

[79] Sadagopan Srinivasan, Li Zhao, Brinda Ganesh, Bruce Jacob, Mike Espig, and Ravi Iyer. 2009. CMP memory modeling: How much does accuracy matter? In *Proceedings of the Fifth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS'09).*

[80] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. 2012. Multi2Sim: A simulation framework for CPU-GPU computing. In *Proceedings of the 2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. IEEE, 335–344.

[81] Richard A. Uhlig and Trevor N. Mudge. 1997. Trace-driven memory simulation: A survey. *ACM Comput. Surv.* 29, 2 (1997), 128–170.

[82] Raul Martin Miguel A. Vega-Rodriguez and Francisco A. Zarallo Gallardo. 2006. Smpcache: Simulator for cache memory systems on symmetric multiprocessors. Retrieved from http://arco.unex.es/smpcache/.

[83] Matthew Walker, Sascha Bischoff, Stephan Diestelhorst, Geoff Merrett, and Bashir Al-Hashimi. 2018. Hardware-validated CPU performance and energy modelling. In *Proceedings of the 2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'18)*. IEEE, 44–53.

[84] Jun Wang, Jesse Beu, Rishiraj Bheda, Tom Conte, Zhenjiang Dong, Chad Kersey, Mitchelle Rasquinha, George Riley, William Song, He Xiao, et al. 2014. Manifold: A parallel simulation framework for multicore systems. In *Proceedings of the 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'14)*. IEEE, 106–115.

[85] Zhonglei Wang and Jörg Henkel. 2013. Fast and accurate cache modeling in source-level simulation of embedded software. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'13)*. EDA Consortium, San Jose, CA, 587–592. http://dl.acm.org/citation.cfm?id=2485288.2485432

[86] Vincent M Weaver and Sally A McKee. 2008. Are cycle accurate simulations a waste of time. In *Proceedings of the 7th Workshop on Duplicating, Deconstructing, and Debunking*. 40–53.

[87] Vincent M. Weaver, Dan Terpstra, and Shirley Moore. 2013. Non-determinism and overcount on modern hardware performance counter implementations. In *Proceedings of the 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'13)*. IEEE, 215–224.

[88] Chencheng Ye, Chen Ding, Hao Luo, Jacob Brock, Dong Chen, and Hai Jin. 2017. Cache exclusivity and sharing: Theory and optimization. *ACM Trans. Arch. Code Optim.* 14, 4 (2017), 34.

[89] Joshua J. Yi and David J. Lilja. 2006. Simulation of computer architectures: Simulators, benchmarks, methodologies, and recommendations. *IEEE Trans. Comput.* 55, 3 (2006), 268–280.

[90] Matt T. Yourst. 2007. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Proceedings of the 2007 IEEE International Symposium on Performance Analysis of Systems & Software*. IEEE, 23–34.

[91] Wei Zang and Ann Gordon-Ross. 2013. A survey on cache tuning from a power/energy perspective. *ACM Comput. Surv.* 45, 3 (2013), 32.

[92] Sizhuo Zhang, Andrew Wright, Daniel Sanchez, et al. 2016. Validating simplified processor models in architectural studies. *arXiv preprint arXiv:1610.02094* (2016).

[93] Li Zhao, Ravi Iyer, Jaideep Moses, Ramesh Illikkal, Srihari Makineni, and Don Newell. 2007. Exploring large-scale CMP architectures using ManySim. *IEEE Micro* 27, 4 (2007).

Received July 2019; revised November 2019; accepted November 2019