CHAPTER THREE

Hardware and Application Profiling Tools

Tomislav Janjusic[*], **Krishna Kavi**[†]
[*]Oak Ridge National Laboratory, Oak Ridge, Tennessee
[†]University of North Texas, Denton, Texas

Contents

1. Introduction 107
   1.1 Taxonomy 107
   1.2 Hardware Profilers 110
   1.3 Application Profilers 112
   1.4 Performance Libraries 113
2. Application Profiling 114
   2.1 Introduction 114
   2.2 Instrumenting Tools 116
   2.3 Event-Driven and Sampling Tools 124
   2.4 Performance Libraries 128
   2.5 Debugging Tools 129
3. Hardware Profiling 130
   3.1 Introduction 130
   3.2 Single-Component Simulators 132
   3.3 Multiple-Component Simulators 133
   3.4 FS Simulators 137
   3.5 Network Simulators 143
   3.6 Power and Power Management Tools 147
4. Conclusions 153
5. Application Profilers Summary 154
6. Hardware Profilers Summary 155
References 156

Abstract

This chapter describes hardware and application profiling tools used by researchers and application developers. With over 30 years of research, there have been numerous tools developed and used, and it will be too difficult to include all of them here. Therefore, in this chapter, we describe various areas with a selection of widely accepted and recent tools. This chapter is intended for the beginning reader interested in exploring more

Advances in Computers, Volume 92
ISSN 0065-2458
http://dx.doi.org/10.1016/B978-0-12-420232-0.00003-9

about these topics. Numerous references are provided to help jump-start the interested reader into the area of hardware simulation and application profiling.

We make an effort to clarify and correctly classify application profiling tools based on their scope, interdependence, and operation mechanisms. To visualize these features, we provide diagrams that explain various development relationships between interdependent tools. Hardware simulation tools are described into categories that elaborate on their scope. Therefore, we have covered areas of single to full-system simulation, power modeling, and network processors.

ABBREVIATIONS

ALU arithmetic logic unit
AP application profiling
API application programming interface
CMOS complementary metal-oxide semiconductor
CPU central processing unit
DMA direct memory access
DRAM dynamic random access memory
DSL domain-specific language
FS full system
GPU graphics processing unit
GUI graphical user interface
HP hardware profiling
HPC high-performance computing
I/O input/output
IBS instruction-based sampling
IR intermediate representation
ISA instruction set architecture
ME microengine
MIPS microprocessor without interlocked pipeline stages
MPI message passing interface
NoC network on chip
OS operating system
PAPI performance application programming interface
PCI peripheral component interconnect
POSIX portable operating system interface
RAM random access memory
RTL register transfer level
SB superblock
SCSI small computer system interface
SIMD single instruction, multiple data
SMT simultaneous multithreading
TLB translation lookaside buffer
VC virtual channel
VCA virtual channel allocation
VLSI very large-scale integration
VM virtual machine

1. INTRODUCTION

Researchers and application developers rely on software tools to help them understand, explore, and tune new architectural components and optimize, debug, or otherwise improve the performance of software applications. Computer architecture research is generally supplemented through architectural simulators and component performance and power efficiency analyzers. These are architecture-specific software tools, which simulate or analyze the behavior of various architectural devices. They should not be confused with emulators, which replicate the inner workings of a device. We will refer to the collective of these tools simply as hardware profilers. To analyze, optimize, or otherwise improve an application's performance and reliability, users utilize various application profilers. In the following subsections, we will distinguish between various aspects of application and hardware analysis, correctness, and performance tuning tools. In Section 2, we will discuss application profiling (AP) tools. In Section 3, we will discuss hardware profiling (HP) tools. Sections 2 and 3 follow the diagrams in Fig. 3.1. Finally, we will conclude the chapter in Section 4.

1.1. Taxonomy

Many tools overlap in their scope and capabilities; therefore, some tools covered in this chapter are not mutually exclusive within the established categories. For example, both instrumentation tools and profiling libraries can

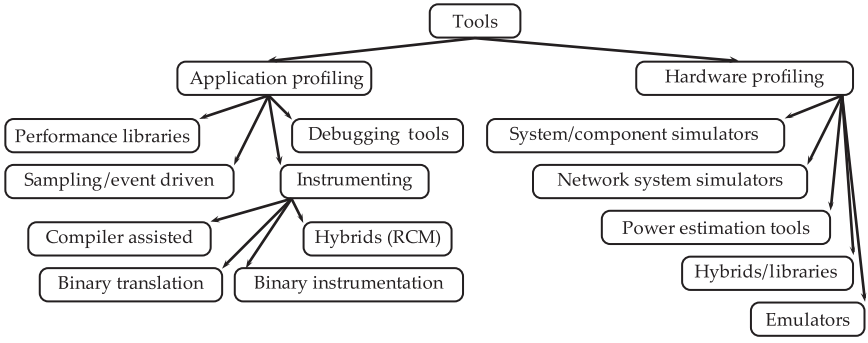

Figure 3.1 Application and hardware profiling categories.

analyze an application's memory behavior, albeit from different perspectives. We must also note that several other tools are built on top of libraries; however, they are classified into different groups based on their scope of application. Likewise, hardware profilers can profile memory subcomponent behavior for a given application. Assuming the simulation environment resembles the native system's hardware, one can expect the profiled behavior to be similar to an actual execution behavior. In this chapter, we classify the tools based on their scope of application.

Our categorization starts by distinguishing between AP and HP. Hardware profiling tools are divided into system and component simulators, power estimation tools, network-system simulators, and architectural emulators. We must note that the terms emulation and simulation are sometimes used interchangeably, but the difference is often seen in completeness. Emulators are faithful imitations of a device, whereas simulators may simulate only certain behaviors of a device. The usefulness of emulators is primarily in their ability to run software, which is incompatible with host hardware (e.g., running codes designed for microprocessor without interlocked pipeline stages (MIPS) processors on Intel x86 processors). Researchers and developers use simulators when interested in a subset of behaviors of a system without worrying about other behaviors; the goal is to simulate faithfully the behaviors of interest. For example, cycle-accurate simulators can be used to observe how instructions flow through a processor execution cycles but may ignore issues related to peripheral devices. This subcategory will be annotated with an s or e in our table representation.

Technically speaking, network-system simulators and power estimation tools should be considered as simulators. We felt that a separate category describing these tools is appropriate because most tools in this category are very specific in their modeling that they distinguish themselves from other generic instruction set simulators.

Some tools are closely related to tool suites and thus cannot be pigeonholed into a specific group. We will list these tools in the category of hybrids/libraries. The category of hybrids includes a range of application and hardware profilers. For example, the Wisconsin Architectural Research Tool Set [1] offers profilers and cache simulators and an overall framework for architectural simulations. Libraries are not tools in the sense that they are readily capable of executing or simulating software or hardware, but they are

designed for the purpose of building other simulators or profilers. Table 3.1 lists a number of HP tools listed in Section 6.

We will cover detailed tool capabilities in the subsequent sections. Figure 3.1 is a graphic representation of our tool categories. The tools, which are described in this chapter, are placed in one of the earlier mentioned categories. For example, emulators and simulators are more closely

Table 3.1 Hardware Profilers
Hardware Profilers

System/component simulators	**Network-system simulators**
DRAMSim [2]	NepSim [3]
DineroIV [4]	Orion [5]
EasyCPU [6]	Garnet [7]
SimpleScalar [8]	Topaz [9]
ML_RSIM [10]	MSNS [11]
M-Sim [12]	Hornet [13]
ABSS [14]	**Power estimation tools**
AugMINT [15]	CACTI [16]
HASE [17]	WATTCH [18]
Simics [19]	DSSWATCH [20]
SimFlex [21]	SimplePower [22]
Flexus [23]	AccuPower [24]
SMARTS [25]	**Frameworks**
Gem5 [26]	CGEN [27]
SimOS [28]	LSE [29]
Parallel SimOS [30]	SID [31]
PTLsim [32]	
MPTLsim [33]	
FeS_2 [34]	
TurboSMARTS [35]	

related to each other than to AP tool; thus, they are listed under hardware profilers. The list of all tools mentioned in this chapter is summarized in Sections 5 and 6. The tables categorize the tools based on Fig. 3.1.

Software profilers and libraries used to build profiler tools are categorized based on their functionality. We felt that application profilers are better distinguished based on their profiling methodology because some plug-in and tools built on top of other platforms have some common functionalities. Consider tools, which may fall under the category of instrumenting tools; these tools could provide the same information as library-based tools that use underlying hardware performance counters. The difference is in how the profiler obtains the information. Most hardware performance counter libraries such as performance application programming interface (PAPI) [36] can provide a sampled measurement of an application's memory behavior. Binary instrumentation tool plug-ins such as Valgrind's Cachegrind or Callgrind can give the user the same information but may incur overheads. Some recent performance tools utilize both performance libraries and instrumentation frameworks to deliver performance metrics. Therefore, a user attempting to use these tools must choose the appropriate tool depending on the required profiling detail, overhead, completeness, and accuracy.

Table 3.2 outlines the tools from Section 5 categorized as instrumenting tools and sampling/event-driven tools and performance libraries and interfaces.

1.2. Hardware Profilers

The complexity of hardware simulators and profiling tools varies with the level of detail that they simulate. Hardware simulators can be classified based on their complexity and purpose: simple-, medium-, and high-complexity system simulators, power management and power-performance simulators, and network infrastructure system simulators.

Simulators that simulate a system's single subcomponent such as the central processing unit's (CPU) cache are considered to be simple simulators (e.g., DineroIV [4], a trace-driven CPU cache simulator). In this category, we often find academic simulators designed to be reusable and easily modifiable.

Medium-complexity simulators aim to simulate a combination of architectural subcomponents such as the CPU pipelines, levels of memory hierarchies, and speculative executions. These are more complex than single-component

Table 3.2 Application Profilers
Application Profilers

Instrumenting	Sampling/event-driven	Performance libraries/kernel interfaces
gprof [37]	OProfile [38]	PAPI [36]
Parasight [39]	AMD CodeAnalyst [40]	perfmon [41]
Quartz [42]	Intel VTune [43]	perfctr
ATOM [44]	HPCToolkit [45]	Debuggers
Pin [46]	TAU [47]	gdb [48]
DynInst [49]	Open SpeedShop [50]	DDT [51]
Etch [52]	VAMPIR [53]	
EEL [54]		
Valgrind [55]		
DynamoRIO [56]		
Dynamite [57]		
UQBT [58]		

simulators but not complex enough to run full-system (FS) workloads. An example of such a tool is the widely known and widely used SimpleScalar tool suite [8]. These types of tools can simulate the hardware running a single application and they can provide useful information pertaining to various CPU metrics (e.g., CPU cycles, CPU cache hit and miss rates, instruction frequency, and others). Such tools often rely on very specific instruction sets requiring applications to be cross compiled for that specific architecture.

To fully understand a system's performance under reasonable-sized workload, users can rely on FS simulators. FS simulators are arguably the most complex simulation systems. Their complexity stems from the simulation of all the critical system's components, as well as the full software systems including the operating system (OS). The benefit of using FS simulators is that they provide more accurate estimation of the behaviors and component interactions for realistic workloads. In this category, we find the widely used Simics [19], Gem5 [26], SimOS [28], and others. These simulators are capable of full-scale system simulations with varying levels of detail. Naturally, their accuracy comes at the cost of simulation times; some simulations may take several

hundred times or even several thousand times longer than the time it takes to run the workload on a real hardware system [25]. Their features and performances vary and will be discussed in the subsequent sections.

Energy consumed by applications is becoming very important for not only embedded devices but also general-purpose systems with several processing cores. In order to evaluate issues related to power requirements of hardware subsystems, researchers rely on power estimation and power management tools. It must be noted that some hardware simulators provide power estimation models; however, we will place power modeling tools into a different category.

In the realm of hardware simulators, we must touch on another category of tools specifically designed to simulate accurately network processors and network subsystems. In this category, we will discuss network processor simulators such as NePSim [3]. For large computer systems, such as high performance computers, application performance is limited by the ability to deliver critical data to compute nodes. In addition, networks needed to interconnect processors consume energy, and it becomes necessary to understand these issues as we build larger and larger systems. Network simulation tools may be used for those studies.

Lastly, when available simulators and profiling tools are not adequate, users can use architectural tool-building frameworks and architectural tool-building libraries. These packages consist of a set of libraries specifically designed for building new simulators and subcomponent analyzers. In this category, we find the liberty simulation environment (LSE) [29], Red Hat's SID environment [31], SystemC, and others.

1.3. Application Profilers

Simulators are powerful tools that give insight into a device behavior under varying runtime circumstances. However, if a user is trying to understand an application's runtime behavior, which is a crucial step when trying to optimize the code, the user needs to rely on different class of tools. AP refers to the ability to measure an application's performance, diagnose potential problems, or otherwise log an application's runtime information. For example, a user may want to reduce an application's memory consumption, which is a common optimization required in embedded applications, and improve an application execution speed, which is a requirement for high-performance scientific applications, or the user simply needs to improve an application's reliability and correctness by tracking active and potential program errors.

For such tasks, application developers rely on debugging tools, profiling tools, and other performance libraries capable of delivering the needed information. These tools vary in their accuracy, profiling speed, and capabilities. Profiling generally involves a software tool called a profiler that analyzes the application. This is achieved either through analyzing the source code statically or at runtime using a method known as dynamic program translation.

Profiling tools can be further categorized into event-driven profiling, sampling profiling, and instrumented profiling. Event-driven profiling collects information about user-defined events, which may require hooks into the OS. When a desired event is triggered, the tool will collect program characteristic related to that event. Event-driven profiling tools have a tendency to rely on the OS to collect necessary information. Sampled profiling aims at collecting information at specified intervals or frequency. The goal is to collect enough samples to acquire a statistically accurate picture of the application. Note that some profiling libraries may be classified as sampling profilers although we must not confuse the libraries and tools that rely on the libraries for functionality. There are various commercial sampling profilers such as AMD CodeAnalyst [40], Intel VTune [43], and others.

Lastly, instrumentation profiling encompasses several subcategories that involve insertion or transformation of source code at strategic sections, usually driven and limited by the instrumenting tool. Instrumenting tools can be further categorized into compiler-assisted instrumentation (e.g., gprof [37]), binary translation, binary instrumentation (e.g., Pin [46], DynInst [49], ATOM [44]), and hybrids (e.g., Valgrind [55] and DynamoRIO [56]). We will discuss the details of each of these categories in the subsequent sections. It should be noted that many tools fit multiple categories; sometimes, this is due to evolution of tools based on need, and sometimes, this is due to the very nature of the profiling. However, we categorized the tools based on their primary functionality or capabilities.

1.4. Performance Libraries

Understanding application's runtime behavior is generally supplemented through AP tools. AP tools come in various forms such as debuggers, profilers, or libraries. From a user's perspective, a profiling tool is software that manipulates an application by injecting foreign code fragments at strategic locations of an application's source code or executable code. The injected code executes at the desired section and collects information on a number

of application-triggered events or other application-specific timing information. Similarly, profiling libraries refer to a subset of AP tools that offer specific application programming interfaces (APIs), which can be called from the client application. In a general sense, profiling libraries are a bridge between profiling tools and specialized hardware units called hardware performance counters, or simply hardware counters, used for hardware performance measurement. We will not list profiling libraries as part of hardware profilers because performance counters are, more often than not, used for software tuning. Implicitly, they provide information about the host hardware during an application execution, but their intent is not driven by hardware research. Another reason why profiling libraries are listed as application profilers is because they cannot distinguish, unless user corrections are applied, between application-triggered metrics and system noise during the application's execution.

2. APPLICATION PROFILING

2.1. Introduction

To our knowledge, the first paper on profiling was published in the early 1980s, gprof: a call graph execution profiler [37]. The need for gprof arose out of the necessity to adequately trace the time spent on specific procedures or subroutines. At that time, profilers were fairly simple and the tools only reported very limited information such as how many times a procedure was invoked. Gprof extended this functionality by collecting program timing information. At compile time, gprof inserts timers or counters, and during execution, the time spent within functions is aggregated. The end results, a collection of time spent within each function, may be analyzed off-line. The information provided by gprof is timing and function call counts with respect to subroutines as a percentage of an application's total executed time. Gprof was unique for its time (and still useful today) because it allowed the programmer to see where the majority of the execution time is spent. The idea of gprof is to allow programmers to tune individual program routines. Gprof is an example of a compiler-assisted profiling tool. Note that modern compilers ship with gprof and other analysis tools. Compiler-assisted profiling inserts profiling calls into the application during compilation process. The inserted profiling routines are invoked when the application executes. This means that the application will incur extra overhead. Gprof's development launched the research area of program profiling. Similar tools were

developed soon after (e.g., Parasight [39] and Quartz [42]). They are similar to gprof, but these tools targeted tuning of parallel applications.

Binary instrumentation tool research accelerated after the development of the ATOM tool [44]. Binary translation tools translate a binary code into a machine code. Their functionality is similar to a compiler. The difference is that unlike compilers their input is a precompiled binary image. As the name suggests, binary translators operate on a binary file, unlike compilers that operate on the source code. Another difference is that the translation tries to gather other information about the code during the translation processes using an interpreter. If a binary translation tool inserts, removes, or otherwise modifies code from the original binary image, then we can call that tool as a code manipulation tool. Binary translation comes in two forms: static binary translation and dynamic binary translation. Binary translators are usually used as optimization frameworks, for example, DIXIE [59] and UQBT [58], because of their ability to translate and modify compiled code. The limitation of static binary translation is the inability to accurately account for all the code because some code paths cannot be predicted statically. When higher accuracy is needed, we should utilize dynamic binary translation. Dynamic translation is slower than static, but dynamic translation is superior to static translation in terms of code coverage. We must note that dynamic translation is not always 100% accurate because it is possible that code paths may be dependent on a specific set of input parameters. While dynamic translators are better at coverage, or predicting taken code paths, they suffer in terms of performance. This is because they must translate blocks of source code on the fly and then execute the translated segments. The reason behind the performance degradation is that in dynamic translation, the translation is performed at runtime and program registers and its state needs to be preserved. Copying and preserving the program registers adds to the total cost of the application's wall-clock time. Since the dynamic translation needs to be fast, complex optimizations are not performed; thus, the translated code may be less efficient. However, the benefits of dynamic translation often outweigh these drawbacks: dynamic translators are more accurate than static because they have runtime knowledge of the taken code paths and this results in better optimizations or analysis for tools that utilize binary translators. Some speedup can be achieved by caching translated code and thus eliminate repeated translations of code segments. This is similar to the idea of CPU caches. Utilizing a code cache reuses frequently used translated blocks. Only portions of the code need to be kept in memory at any one time, thereby improving performance at the expense of memory requirements.

Binary instrumentation tools are different from other profiling tools because their main approach lies in injecting program executable with additional code. The inserted code is then passed on to various plug-in tools for additional analysis. Binary instrumentation similar to binary translation comes in two forms: static binary instrumentation and dynamic binary instrumentation. The trade-offs between the two mechanisms are also in terms of performance and accuracy. Static binary instrumentation is faster but less accurate, and dynamic binary instrumentation is slower but more accurate. It is important to note that ATOM and most other instrumenting tools are considered a tool-building system. This means that the underlying functionality of binary instrumentation tools allows for other plug-in tools to run on top of the framework. Development of ATOM has inspired other developers to build other more capable tools. We can consider these five instrumentation technique categories: manual, compiler-assisted, binary translations, runtime instrumentation, and hybrids.

In addition, programmers can manually instrument their code. Functions that will collect some type of runtime information are manually inserted into the application's source code. These vary from simple print statements to complex analysis routines specifically developed for debugging and profiling purposes.

For clarity purposes, however, we can group binary translation, binary instrumentation, and hybrids of these tools simply into runtime instrumentation tools. Runtime instrumentation is the predominant technique among modern profiling tools. It involves an external tool that supervises the application being instrumented. The instrumented code may be annotated or otherwise transformed into an intermediary representation. The plug-in tools that perform various types of analyses use annotated code or the intermediate representation. This also implies that the extent, accuracy, and efficiency of the plug-in tools are limited by the capabilities provided by the framework. The benefit of runtime instrumentation, particularly binary instrumentation, is the level of binary code detail that plug-in tools can take advantage of. Example frameworks in this area are Pin [46], Valgrind [55], DynamoRIO [56], DynInst [49], and others.

2.2. Instrumenting Tools

Instrumentation is a technique that injects analysis routines into the application code to either analyze or deliver the necessary metadata to other analysis tools. Instrumentation can be applied during various application development

cycles. During the early development cycles, instrumentation comes in the form of various print statements; this is known as manual instrumentation. For tuning and optimization purposes, manual instrumentation may invoke underlying hardware performance counters or OS events.

Compiler-assisted instrumented utilizes the compiler infrastructure to insert analysis routines, for example, instrumentation of function boundaries, to instrument the application's function call behavior.

Binary translation tools are a set of tools that reverse compile an application's binary into intermediate representation (IR) suitable for program analysis. The binary code is translated, usually at basic-block granularity, interpreted, and executed. The translated code may simply be augmented with code that measures desired properties and resynthesized (or recompiled) for execution. Notice that binary translation does not necessarily include any instrumentation to collect program statistics. The instrumentation in this sense refers to the necessity to control the client application by redirecting code back to the translator (i.e., every basic block of client application must be brought back under the translator's control).

Instrumentation at the lowest levels is applied on the application's executable binaries. Application's binary file is dissected block by block or instruction by instruction. The instruction stream is analyzed and passed to plug-in tools or interpreters for additional analysis.

Hybrids are tools that are also known as runtime code manipulation tools. We opted to list hybrid tools as a special category because some tools in this group are difficult to categorize. Hybrid tools apply binary translation and binary instrumentation. The translation happens in the framework's core and the instrumentation is left to the plug-in tools (e.g., Valgrind [55]) (Table 3.3).

2.2.1 Valgrind

Valgrind is a dynamic binary instrumentation framework that was initially designed for identifying memory leaks. Valgrind and other tools in this realm are also known as shadow value tools. That means that they shadow every register with another descriptive value. Valgrind belongs to a complex or heavyweight analysis tools in terms of both its capabilities and the complexities. In our taxonomy, Valgrind is an instrumenting profiler that utilizes a combination of binary translation and binary instrumentation. Referring to the chart in Fig. 3.1, Valgrind falls into the hybrid category. The basic Valgrind structure consists of a core tool and plug-in tools. The core tool is responsible for disassembling the client application's binary file into an

Table 3.3 Application Profiling Instrumenting Tools
Instrumenting Tools

Compiler-Assisted	Binary Translation	Binary Instrumentation	Hybrids/Runtime Code Manipulation
gprof [37]	Dynamite [60]	DynInst [49]	DynamoRIO [57]
Parasight [39]	UQBT [58]	Pin [46]	Valgrind [55]
Quartz [42]		ATOM [44]	
		Etch [52]	
		EEL [54]	

IR specific to Valgrind. The client code is partitioned into superblocks (SBs). An SB, consisting of one or more basic blocks, is a stream of approximately 50 instructions. The block is translated into an IR and passed on to the instrumentation tool. The instrumentation tool then analyzes every SB statement and inserts appropriate instrumented calls. When the tool is finished operating on the SB, it will return the instrumented SB back to the core tool. The core tool recompiles the instrumented SB into machine code and executes the SB on a synthetic CPU. This means that the client application never directly runs on the host processor. Because of this design, Valgrind is bound to a specific CPU and OS. Valgrind supports several combinations of CPUs and OS systems including AMD64, x86, ARM, and PowerPC 32/64 running predominately Linux/Unix systems.

Several widely used instrumentation tools come with Valgrind, while others are designed by researchers and users of Valgrind:

- Memcheck: Valgrind's default tool Memcheck enables the user to detect memory leaks during execution. Memcheck detects several common C and C++ errors. For example, it can detect accesses to restricted memory such as areas of heap that were deallocated, using undefined values, incorrectly freed memory blocks, or a mismatched number of allocation and free calls.
- Cachegrind: Cachegrind is Valgrind's default cache simulator. It can simulate a two-level cache hierarchy and an optional branch predictor. If the host machine has a three-level cache hierarchy, Cachegrind will simulate the first and third cache level. The Cachegrind tool comes with a third-party annotation tool that will annotate cache hit/miss statistics per source code line. It is a good tool for users who want to find potential memory performance bottlenecks in their programs.

- Callgrind: Callgrind is a profiling tool that records an application's function call history. It collects data relevant to the number of executed instructions and their relation to the called functions. Optionally, Callgrind can also simulate the cache behavior and branch prediction and relate that information to function call profile. Callgrind also comes with a third-party graphic visualization tool that helps visualize Callgrind's output.
- Helgrind: Helgrind is a thread error detection tool for applications written in C, C++, and Fortran.

  It supports portable operating system interface (POSIX) pthread primitives. Helgrind is capable of detecting several classes of error that are typically encountered in multithreaded programs. It can detect errors relating to the misuse of the POSIX API that can potentially lead to various undefined program behavior such as unlocking invalid mutexes, unlocking a unlocked mutex, thread exits still holding a lock, and destructions of uninitialized or still waited upon barriers. It can also detect error pertaining to an inconsistent lock ordering. This allows it to detect any potential deadlocks.
- Massif: Massif is a heap profiler tool that measures an application's heap memory usage. Profiling an application's heap may help reduce its dynamic memory footprint. As a result, reducing an application's memory footprint may help avoid exhausting a machine's swap space.
- DHAT: DHAT is a dynamic heap analysis tool similar to Massif. It helps identify memory leaks and analyze application allocation routines that allocate large amounts of memory but are not active for very long, allocation routines that allocate only short lived blocks or allocations that are not used or used incompletely.
- Lackey: Lackey is a Valgrind tool that performs various kinds of basic program measurements. Lackey can also produce very rudimentary traces that identify the instruction and memory load/store operations. These traces can then be used in a cache simulator (e.g., Cachegrind operates on a similar principle).
- Gleipnir: Gleipnir is a program profiling and tracing tool built as a third-party plug-in tool [61]. It combines several native Valgrind tools into a tracing–simulating environment. By taking advantage of Valgrind's internal debug symbol table parser, Gleipnir can trace memory accesses and relate each access to a specific program internal structure such as thread; program segment; function, local, global, and dynamic data structure; and scalar variables. Gleipnir's ability to collect fine-grained

memory traces and associate each access to source level data structures and elements of these structures makes it a good candidate tool for advanced cache memory simulation and analysis. The data provided by Gleipnir may be used by cache simulators to analyze accesses to data structure elements or by programmers to understand the relation between dynamic and static memory behavior. The goal of Gleipnir is to provide traces with rich information, which can aid in advanced analysis of memory behaviors. Gleipnir aims to bridge the gap between a program's dynamic and static memory behavior and its impact on application performance.

2.2.2 DynamoRIO

DynamoRIO [56] is a dynamic optimization and modification framework built as a revised version of Dynamo. It operates on a basic-block granularity and is suitable for various research areas: code modification, intrusion detection, profiling, statistical gathering, sandboxing, etc. It was originally developed for Windows OS but has been ported to a variety of Linux platforms. The key advantage of DynamoRIO is that it is fast and it is designed for runtime code manipulation and instrumentation. Similar to Valgrind, DynamoRIO is classified as a code manipulation framework and thus falls in the hybrid category in Fig. 3.1. Unlike other instrumentation tools, Dynamo does not emulate the incoming instruction stream of a client application but rather caches the instructions and executes them on the native target. DynamoRIO intercepts control transfers after every basic block because it operates on basic-block granularity. Performance is gained through various code block stitching techniques, for example, basic blocks that are accessed through a direct branch are stitched together so that no context switch, or other control transfer, needs to occur. Multiple code blocks are cached into a trace for faster execution. The framework employs an API for building DynamoRIO plug-in tools. Because DynamoRIO is a code optimization framework, it allows the client to access the cached code and perform client-driven optimizations.

In dynamic optimization frameworks, instruction representation is key to achieving fast execution performance. DynamoRIO represents instructions at several levels of granularity. At the lowest level, the instruction holds the instruction bytes, and at the highest level, the instruction is fully decoded at machine representation level. The level of detail is determined by the routine's API used by the plug-in tool. The levels of details can be automatically

and dynamically adjusted depending on later instrumentation and optimization needs.

The client tools operate through hooks that offer the ability to manipulate either basic blocks or traces. In DynamoRIO's terminology, a trace is a collection of basic blocks. Most plug-in tools operate on repeated executions of basic blocks also known as hot code. This makes sense because the potential optimization savings are likely to improve those regions of code. In addition, DynamoRIO supports adaptive optimization techniques. This means that the plug-in tools are able to reoptimize code instructions that were placed in the code cache and ready for execution.

Dynamic optimization frameworks such as DynamoRIO are designed to improve and optimize applications. As was demonstrated in Ref. [56], the framework improves on existing high-level compiler optimizations.

The following tools are built on top of the DynamoRIO framework:

- TaintTrace: TaintTrace [62] is a flow tracing tool for detecting security exploits.
- Dr. Memory: Dr. Memory [63] is a memory profiling tool similar to Valgrind's Memcheck. It can detect memory-related errors such as accesses to uninitialized memory, accesses to freed memory, improper allocation, and free ordering. Dr. Memory is available for both Windows and Linux OSs.
- Adept: Adept [64] is a dynamic execution profiling tool built on top of the DynamoRIO platform.

It profiles user-level code paths and records them. The goal is to capture the complete dynamic control flow, data dependencies, and memory references of the entire running program.

2.2.3 Pin

Pin [46] is a framework for dynamic binary program instrumentation that follows the model of the popular ATOM tool (which was designed for DEC Alpha-based systems, running DEC Unix), allowing the programmer to analyze programs at instruction level. Pin's model allows code injection into client's executable code. The difference between ATOM and Pin is that Pin dynamically inserts the code while the application is running, whereas ATOM required the application and the instrumentation code to be statically linked. This key feature of Pin allows it to attach itself to already running process, hence the name Pin. In terms of taxonomy, Pin is an instrumenting profiler that utilizes dynamic binary instrumentation. It is in many ways similar to Valgrind and other dynamic

binary instrumentation tools; however, Pin does not use an intermediate form to represent the instrumented instructions. The primary motivation of Pin is to have an easy to use, transparent, and efficient tool-building system.

Unlike Valgrind, Pin uses a copy and annotates IR, implying that every instruction is copied and annotated with metadata. This offers several benefits as well as drawbacks. The key components of a Pin system are the Pin virtual machine (VM) with just-in-time (JIT) compiler, the Pintools, and the code cache. Similar to other frameworks, a Pintool shares a client's address space, resulting in some skewing of address space; application addresses may be different when running with Pin compared to running without Pin. The code cache stores compiled code waiting to be launched by the dispatcher. Pin uses several code optimizations to make it more efficient.

For a set of plug-in tools, an almost necessary feature is its access to the compiler-generated client's symbol table (i.e., its debug information). Unlike Valgrind, Pin's debug granularity ends at the function level. This means that tracing plug-in tools such as Gleipnir can map instructions only to the function level. To obtain data-level symbols, a user must rely on debug parsers built into the plug-in tool.

Pin uses several instrumentation optimization techniques that improve the instrumentation speed. It is reported in Refs. [46,55] that Pin outperforms other similar tools for basic instrumentation. Pin's rich API is well documented and thus attractive to users interested in building Pin-based dynamic instrumentation. Pin comes with many examples; Pintools can provide data on basic blocks, instruction and memory traces, and cache statistics.

2.2.4 DynInst

DynInst [65] is a runtime instrumentation tool designed for code patching and program performance measurement. It expands on the design of ATOM, EEL, and Etch by allowing the instrumentation code to be inserted at runtime. This contrasts with the earlier static instrumentation tools that inserted the code statically at postcompile time. DynInst provides a machine-independent API designed as part of the Paradyn Parallel Performance Tools project. The benefit of DynInst is that instrumentation can be performed at arbitrary points without the need to predefine these points or to predefine the analysis code at these points.

The ability to defer instrumentation until runtime and the ability to insert arbitrary analysis routines make DynInst good for instrumenting large-scale scientific programs. The dynamic instrumentation interface is designed to be primarily used by higher-level visualization tools.

The DynInst approach consists of two manager classes that control instrumentation points and the collection of program performance data. DynInst uses a combination of tracing and sampling techniques. An internal agent, the metric manager, controls the collection of relevant performance metrics. The structures are periodically sampled and reported to higher-level tools. It also provides a template for a potential instrumentation perturbation cost. All instrumented applications incur performance perturbation because of the added code or intervention by the instrumentation tool. This means that performance gathering tools need to account for their overhead and adjust performance data accordingly.

The second agent, an instrumentation manager, identifies relevant points in the application to be instrumented. The instrumentation manager is responsible for the inserted analysis routines. The code fragments that are inserted are called trampolines. There are two kinds of trampolines: base and mini trampolines. A base trampoline facilitates the calling of mini trampolines, and there is one base trampoline active per instrumentation point. Trampolines are instruction sequences that are inserted at instrumentation points (e.g., beginning and end of function calls) that save and restore registers after the analysis codes complete data collection. DynInst comes with an API that enables tool developers to build other analysis routines or new performance measurement tools built on top of the DynInst platform.

There are several tools built around, on top of, or utilizing parts of the DynInst instrumentation framework:

- TAU: TAU [47] is a comprehensive profiling and tracing tool for analyzing parallel programs. By utilizing a combination of instrumentation and profiling techniques, TAU can report fine-grained application performance data. Applications can be profiled using various techniques using TAU's API. For example, users can use timing, event, and hardware counters in combination with application dynamic instrumentation. TAU comes with visualization tools for understanding and interpreting large amounts of data collected.
- Open SpeedShop: Open SpeedShop [50] is a Linux-based performance tool for evaluating performance of applications running on single-node and large-scale multinode systems. Open SpeedShop incorporates several performance gathering methodologies including sampling,

call-stack analysis, hardware performance counters, profiling message passing interface (MPI) libraries and input/output (I/O) libraries, and floating-point exception analysis. The tool is supplemented by a graphical user interface (GUI) for visual data inspection.

- Cobi: Cobi is a DynInst-based tool for static binary instrumentation. It leverages several static analysis techniques to reduce instrumentation overheads and metric dilation at the expense of instrumentation detail for parallel performance analysis.

2.3. Event-Driven and Sampling Tools

Sampling-based tools gather performance or other program metrics by collecting data at specified intervals. One can be fairly conservative with our categories of sampling-based tools as most of them rely on other types of libraries or instrumentation frameworks to operate. Sampling-based approaches generally involve interrupting running programs periodically and examining the program's state, retrieving hardware performance counter data, or executing instrumented analysis routines. The goal of sampling-based tools is to capture enough performance data at reasonable number of statistically meaningful intervals so that the resulting performance data distribution will resemble the client's full execution. Sampling-based approaches are sometimes known as statistical methods when referring to the data collected. Sampling-based tools acquire their performance data based on three sampling approaches: timer-based, event-based, and instruction-based. Diagram in Fig. 3.2 shows the relationships of sampling-based tools.

- Timer-based performance measurements: Timer-based and timing-based approaches are generally the basic forms of AP, where the sampling is based on built-in timers. Tools that use timers are able to obtain a general picture of execution times spent within an application. The amount of time spent by the application in each function may be derived from the sampled data. This allows the user to drill down into the specific program's function and eliminate possible bottlenecks.
- Event-based performance measurements: Event-based measurements sample information when predetermined events occur. Events can be either software or hardware events, for example, a user may be interested in the number of page faults encountered or the number of specific system calls. These events are trapped and counted by the underlying OS library primitives, thereby providing useful 16 information back to the tool and ultimately the user. Mechanisms that enable

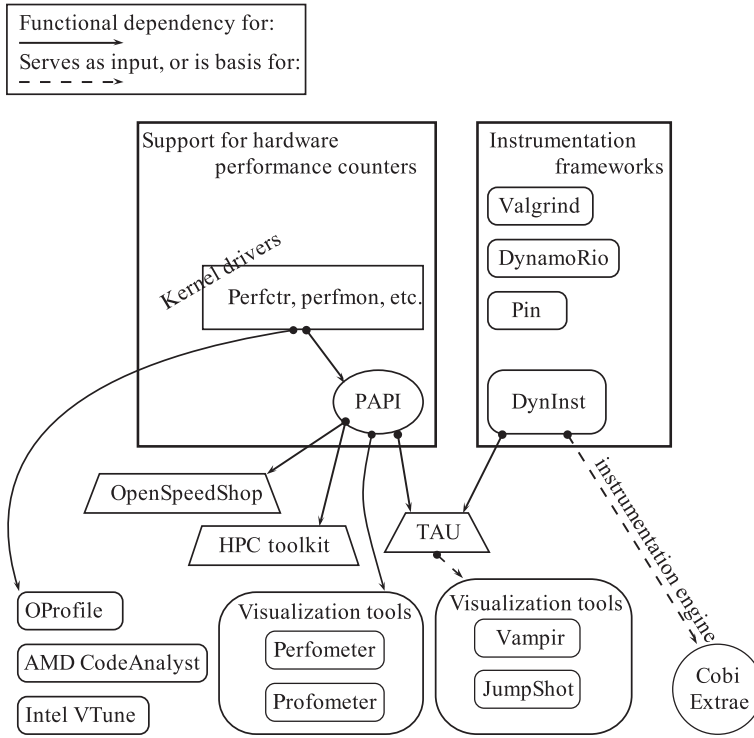

Figure 3.2 Sampling tools and their dependencies and extensions.

event-based profiling are generally the building blocks of many sampling-based tools.

- Instruction-based performance measurement: Arguably, the most accurate profiling representations are tools that use instruction-based sampling (IBS) approach. For example, AMD CodeAnalyst [40] uses the IBS method to interrupt a running program after a specified number of instructions and examine the state of hardware counters. The values obtained from the hardware counters can be used to reason about the program performance. The accuracy of instruction sampling depends on the sampling rate.

The basic components of sampling tools include the host architecture, software/hardware interfaces, and visualization tools. Most sampling tools use hardware performance counters and OS interfaces. We describe several sampling tools here.

2.3.1 OProfile

OProfile [38] is an open-source system-wide profiler for Linux systems. System-wide profilers are tools that operate in kernel space and are capable of profiling application and system-related events. OProfile uses a kernel driver and a daemon to sample events. Data collected on the events are aggregated into a file for postprocessing. The method by which OProfile collects information is through either hardware events or timing. In case the hardware performance counters are not available, OProfile resorts to using timers. The information is enough to account for time spent in individual functions; however, it is not sufficient to reason about application bottlenecks.

OProfile includes architecture-specific components, OProfile file system, a generic kernel driver, OProfile daemon, and postprocessing tools. Architecture-specific components are needed to use available hardware counters. The OProfile file system, oprofilefs, is used to aggregate information. The generic kernel driver is the data delivery management technique from the kernel to the user. The OProfile daemon is a user-space program that writes kernel data back to the disk. Graphic postprocessing tools provide user interfaces (GUIs) to correlate aggregated data to the source code.

It is important to note that to some extent, most open-source and commercial tools consist of these basic components, and virtually, all share the basic requirement of hardware performance counters or OS events, unless they rely on binary instrumentation.

2.3.2 Intel VTune

Intel VTune is a commercial system-wide profiler for Windows and Linux systems. Similar to OProfile, it uses timer and hardware event sampling technique to collect performance data that can be used by other analysis tools. The basic analysis techniques are timer analyses, which report the amount of time spent in individual functions, or specific code segments. Functions containing inefficiently written loops may be identified and optimized; however, the tool itself does not offer optimizations and it is left up to the user to find techniques for improving the code. Because modern architectures offer multiple cores, it is becoming increasingly important to fine-tune threaded applications. Intel VTune offers timing and CPU utilization information on application's threads. These data give programmers insights into how well their multithreaded designs are running. The information provided gives timing information for individual threads, time spent waiting for locks, and scheduling information. The

more advanced analysis techniques are enabled with hardware event sampling. Intel VTune can use the host architecture performance counters to record statistics. For example, using hardware counters, a user can sample the processor's cache behavior, usually last-level caches, and relate poor cache performance back to the source code statements. We must stress that to take advantage of these reports, a programmer must be knowledgeable of host hardware capabilities and have a good understanding of compiler and hardware interactions. As the name implies, Intel VTune is specifically designed for the Intel processors and the tool is tuned for Intel-specific compilers and libraries.

2.3.3 AMD CodeAnalyst

AMD CodeAnalyst is very similar to Intel VTune except that it targets AMD processors. Like other tools in this group, AMD CodeAnalyst requires underlying hardware counters to collect information about an application's behavior. The basic analysis is the timing-based approach where application's functions are broken down by the amount of time spent in individual functions. Users can drill down to individual code segments to find potential bottlenecks in the code and tune code to improve performance. For multithreaded programs, users can profile individual threads, including core utilization and affinity. Identifying poor memory localities is a useful feature for nonuniform memory access (NUMA) platforms. The analyses are not restricted to homogeneous systems (e.g., general-purpose processors only). With the increasing use of graphics processing units (GPUs) for scientific computing, it is becoming increasingly important to analyze the behavior of GPUs. AMD CodeAnalyst can display utilization of heterogeneous systems and relate the information back to the application. Performance bottlenecks in most applications are memory-related; thus, recent updates to analysis tools address data-centric visualization. For example, newer tools report on a CPU's cache line utilizations in an effort to measure the efficiency of data transfers from main memory to cache. AMD CodeAnalyst can be used to collect many useful data including instructions per cycle (IPC), memory access behavior, instruction and data cache utilization, translation lookaside buffer (TLB) misses, and control transfers. Most, and perhaps all, of these metrics are achieved through hardware performance counters. Interestingly for Linux-based systems, CodeAnalyst is integrated into OProfile (a system-wide event-based profiler for Linux described earlier).

2.3.4 HPCToolkit

HPCToolkit [45] is a set of performance measuring tools aimed at parallel programs. HPCToolkit relies on hardware counters to gather performance data and relates the collected data back to the calling context of the application's source code. HPCToolkit consists of several components that work together to stitch and analyze the collected data:

- hpcrun: hpcrun is the primary sampling profiler that executes an optimized binary. Hpcrun uses statistical sampling to collect performance metrics.
- hpcstruct: hpcstruct operates on the application's binary to recover any debug-relevant information later to be stitched with the collected metrics.
- hpcprof: hpcprof is the final analysis tool that correlates the information from hpcrun and hpcprof.
- hpcviewer: hpcviewer is the toolkit's GUI that helps visualize hpcprof's data.

HPCToolkit is a good example of a purely sampling-based tool. It uses sampling to be as minimally intrusive as possible and minimal execution overhead for profiling applications.

2.4. Performance Libraries

Performance libraries rely on hardware performance counters to collect performance data. Due to their scope, we have listed performance libraries as application profilers rather than hardware profilers. As we will explain at the end of this subsection, performance libraries have been used by several performance measuring tools to access hardware counters.

Despite their claims of nonintrusiveness, it was reported in Ref. [66] that performance counters still introduce some data perturbations since the counters may still count events that are caused by the instrumentation code. And the perturbation is proportional to the sampling rate. Users interested in using performance libraries should ensure that the measured application (i.e., the original code plus inserted measurement routines) resembles the native application (i.e., unmodified application code) as closely as possible.

The benefit of using performance counters is that these tools are the least intrusive and arguably the fastest for AP. Code manipulative tools, such as instrumenting tools, tend to skew the native application's memory image by cloning or interleaving the application's address space or adding and removing a substantial amount of instructions as part of their instrumentation framework.

Generally, libraries and tools that use hardware counters to measure performance require the use of kernel interfaces. Therefore, most tools are tied to specific OS kernels and available interfaces. The approach is to use a kernel interface to access hardware performance counters and use system libraries to facilitate the calling convention to those units. And third-party tools are used to visualize the collected information. Among commonly used open-source interfaces are perfmon and perfctr. Tools such as PAPI [36] utilize the perfctr kernel interface to access hardware units. Other tools such as TAU [47], HPCToolkit [45], and Open SpeedShop [50] all utilize the PAPI performance libraries.

- PAPI: PAPI [36] is a portable API, often referred to as an interface to performance counters. Since its development, PAPI has gained widespread acceptance and is maintained by an active community of open-source developers.

  PAPI's portable design offers high-level and low-level interfaces designed for machine independence and portability. The perfctr kernel module handles the Linux kernel hardware interface. The high- and low-level PAPI interfaces are tailored for both novice users and application engineers that require a quick turnaround time for sampling and benchmarking.

  PAPI offers several abstractions. Event sets are PAPI abstractions to count, add, or subtract sets of hardware counters without incurring additional system overhead. PAPI event sets offer users the ability to correlate different hardware sets back to the application source. This is useful to understand application-specific performance metrics.

  Overflow events are PAPI features aimed at tool writers. Threshold overflow allows the user to trigger specific signals an event when a specific counter exceeded a predefined amount. This allows the user to hash instructions that overflowed a specific event and relate it back to application symbol information.

  The number of hardware counters is limited and the data collection must wait until the application completes execution. PAPI offers multiplexing to alleviate that problem by subdividing counter usage over time. This could have adverse affects on the accuracy of reported performance data.

2.5. Debugging Tools

While most of the focus of this chapter is on profiling tools and performance libraries, it is important to keep in mind another category of tools that help

with program correctness. Virtually, everyone in the programming community is familiar with debugging tools. Programmers are usually confronted with either a compiler error or a logical error. Compiler errors tend to be syntactical in nature, that is, the programmer used a compiler unfriendly syntax. Logical errors are harder to find and they occur when a correctly compiled program errs during runtime. Debugging tools are used to identify logical errors in programs. Users can examine each code statement and logically traverse the program flow. There are a number of debuggers in use, and most integrated development environments (IDE) come with their own versions of program debuggers. For the common Unix/Linux user, gdb [48] will suffice.

3. HARDWARE PROFILING

3.1. Introduction

Simulation of hardware (or processor architecture) is a common way of evaluating new designs. This in turn accelerates hardware development because software models can be built from scratch within months rather than years that it takes to build physical hardware. Simulations allow the designers to explore a large number of variables and trade-offs. The main drawback of simulators is the level of detail that is examined. To obtain accurate results, simulators will have to be very detailed, and such simulators will be very complex in terms of both their design and the amount of time needed to complete a simulation. As modern systems are getting more complex with large number of processing cores, network-on-chip (NoC), large multilevel caches, faithfully simulating such systems is becoming prohibitive. Many researchers limit their studies to a subsystem, such as cache memories. In such cases, one needs only to simulate the interested subsystem in detail while abstracting other components.

Many architectural simulators are available for academic and commercial purposes. As stated previously, the accuracy of the data generated by the simulators depends on the level of detail simulated, the complexity of the simulation process, and the nature of benchmarks that can be simulated. Simulators may simulate single components, multiple components, or entire computer system capable of running FS including OSs. A paper describing architectural simulators for academic and classroom purposes is described in Ref. [67].

In this section, we expose the reader to a variety of simulation and modeling tools, but we will constrain our review to architectural simulator

based on their complexity and scope. We will introduce several simulators capable of simulating single component or FS. We will also treat simulation tools for modeling networks as well as modeling power consumption separately from architectural simulators. Figure 3.3 is a diagram that shows the various relationships between current and past tools. It also shows how various power modeling tools are used as interfaces with architectural simulators:

- Single-component simulators: Any simulator that simulates a single subsystem, regardless of accuracy or code complexity, is least complex among hardware simulators. In this category, we can find trace-driven tools such as the DineroIV cache simulator [4] or DRAMSim [2].
- Multiple-component simulators: Simulator tools that have the capability to simulate more than one subsystem are more complex than single-component simulators. An example of a simulator that can simulate

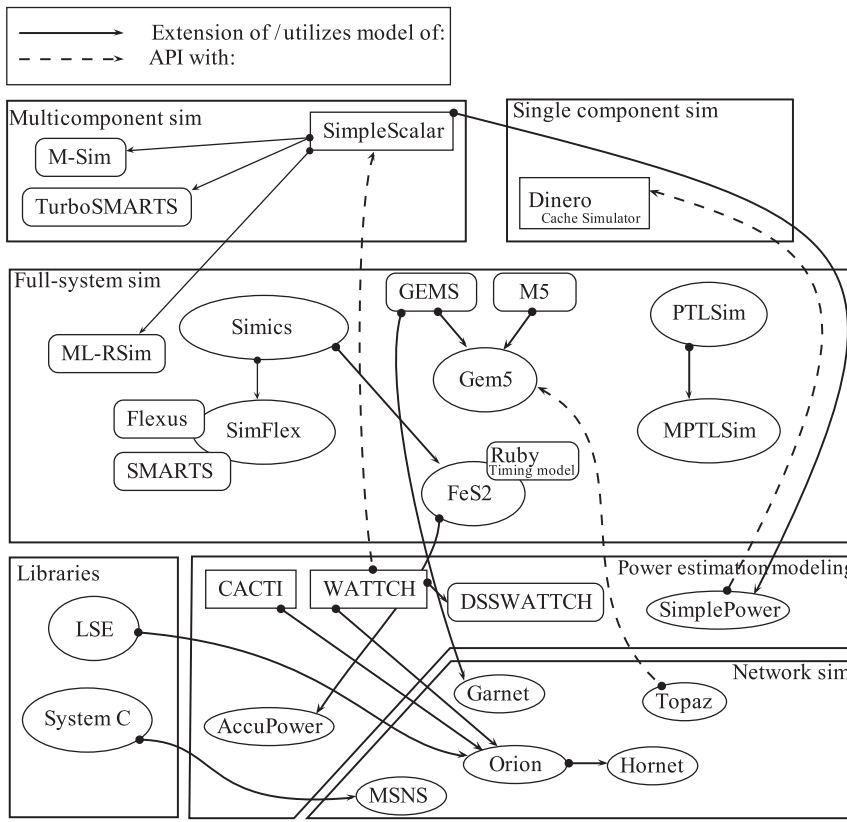

Figure 3.3 Hardware tools and their relationships and extensions.

multiple components is the widely used SimpleScalar [8] tool set that simulates a single CPU and several levels of memory systems.

- FS simulators: These are the most complex among architectural simulator as they simulate all subsystems, including multiple processing cores, interconnection buses, and multiple levels of memory hierarchy, and they permit simulation of realistic workloads under realistic execution environments. Note that the definition of an FS changes with time, since computing systems complexities in terms of the number of cores, complexity of each core, programming models for homogeneous and heterogeneous cores, memory subsystem, and interconnection subsystem are changing with time. Thus, today's FS simulators may not be able to simulate next-generation systems fully. In general, however, FS simulators simulate both hardware and software systems (including OSs and runtime systems).
- Network simulators: At the core of virtually every network router lies a network processor. They are designed to exploit packet-level parallelism by utilizing multiple fast execution cores. To study and research network processor complexity, power dissipation, and architectural alternatives, researchers rely on network simulators. Network simulators share many of the same features of other FS architectures, but their scope and design goals are specifically target network subsystem.
- Power estimation and modeling tools: Designing energy-efficient processors is a critical design pursued by current processor architects. With the increasing density of transistors integrated on a chip, the need to reduce power consumed by each component of the system becomes even more critical. To estimate the trade-offs between architectural alternatives in the power versus performance design space, researchers rely on power modeling and power estimation tools. They are similar to other simulators that focus on performance estimation, but power tools focus on estimating power consumed by each device when executing an application.

3.2. Single-Component Simulators

Although we categorize simulators that simulate a single component of a system as low-complexity simulators, they may require simulation of all the complexities of the component. Generally, these simulators are trace-driven: They receive input in a single file (e.g., a trace of instructions) and they simulate the component behavior for the provided input (e.g., if

a given memory address causes a cache hit or miss or if an instruction requires a specific functional unit). The most common example of such simulators is memory system simulators including those that simulate main memory systems (e.g., Ref. [2] used to study RAM (random access memory) behavior) or caches (e.g., DineroIV [4]). Other simple simulators are used for educational purposes such as the EasyCPU [6].

3.2.1 DineroIV

DineroIV is a trace-based uniprocessor cache simulator [4]. The availability of the source code makes it easy to modify and customize the simulator to model different cache configurations, albeit for a uniprocessor environment. DineroIV accepts address traces representing the addresses of instructions and data accessed when a program is executed and models if the referenced addresses can be found in (multilevel) cache or cause a miss. DineroIV permits experimentation with different cache organizations including different block size, associativity, and replacement policy. Trace-driven simulation is an attractive method to test architectural subcomponents because experiments for different configurations of the component can be evaluated without having to reexecute the application through an FS simulator. Variations to DineroIV are available that extend the simulator to model multicore systems; however, many of these variations are either unmaintained or difficult to use.

3.3. Multiple-Component Simulators

Medium-complexity simulators model multiple components and the interactions among the components, including a complete CPU with in-order or out-of-order execution pipelines, branch prediction and speculation, and memory subsystem. A prime example of such a system is the widely used SimpleScalar tool set [8]. It is aimed at architecture research although some academics deem SimpleScalar to be invaluable for teaching computer architecture courses. An extension known as ML-RSIM [10] is an execution-driven computer system simulating several subcomponents including an OS kernel. Other extension includes M-Sim [12], which extends SimpleScalar to model multithreaded architectures based on simultaneous multithreading (SMT).

3.3.1 SimpleScalar

SimpleScalar is a set of tools for computer architecture research and education. Developed in 1995 as part of the Wisconsin Multiscalar project, it has since

sparked many extensions and variants of the original tool. It runs precompiled binaries for the SimpleScalar architecture. This also implies that SimpleScalar is not an FS simulator but rather user-space single application simulator. SimpleScalar is capable of emulating Alpha, portable instruction set architecture (PISA) (MIPS like instructions), ARM, and x85 instruction sets. The simulator interface consists of the SimpleScalar ISA and POSIX system call emulations.

The available tools that come with SimpleScalar include sim-fast, sim-safe, sim-profile, sim-cache, sim-bpred, and sim-outorder:

- sim-fast is a fast functional simulator that ignores any microarchitectural pipelines.
- sim-safe is an instruction interpreter that checks for memory alignments; this is a good way to check for application bugs.
- sim-profile is an instruction interpreter and profiler. It can be used to measure application dynamic instruction counts and profiles of code and data segments.
- sim-cache is a memory simulator. This tool can simulate multiple levels of cache hierarchies.
- sim-bpred is a branch predictor simulator. It is intended to simulate different branch prediction schemes and measures miss prediction rates.
- sim-outorder is a detailed architectural simulator. It models a superscalar pipelined architecture with out-of-order execution of instructions, branch prediction, and speculative execution of instructions.

3.3.2 M-Sim

M-Sim is a multithreaded extension to SimpleScalar that models detailed individual key pipeline stages. M-Sim runs precompiled Alpha binaries and works on most systems that also run SimpleScalar. It extends SimpleScalar by providing a cycle-accurate model for thread context pipeline stages (reorder buffer, separate issue queue, and separate arithmetic and floating-point registers). M-Sim models a single SMT capable core (and not multicore systems), which means that some processor structures are shared while others remain private to each thread; details can be found in Ref. [12].

The look and feel of M-Sim is similar to SimpleScalar. The user runs the simulator as a stand-alone simulation that takes precompiled binaries compatible with M-Sim, which currently supports only Alpha APX ISA.

3.3.3 ML-RSIM

This is an execution-driven computer system simulator that combines detailed models of modern computer hardware, including I/O subsystems,

with a fully functional OS kernel. ML-RSIM's environment is based on RSIM, an execution-driven simulator for instruction-level parallelism (ILP) in shared memory multiprocessors and uniprocessor systems. It extends RSIM with additional features including I/O subsystem support and an OS. The goal behind ML-RSIM is to provide detailed hardware timing models so that users are able to explore OS and application interactions. ML-RSIM is capable of simulating OS code and memory-mapped access to I/O devices; thus, it is a suitable simulator for I/O-intensive interactions.

ML-RSIM implements the SPARC V8 instruction set. It includes cache and TLB models, and exception handling capabilities. The cache hierarchy is modeled as a two-level structure with support for cache coherency protocols. Load and store instructions to I/O subsystem are handled through an uncached buffer with support for store instruction combining. The memory controller supports MESI (modify, exclusive, shared, invalidate) snooping protocol with accurate modeling of queuing delays, bank contention, and dynamic random access memory (DRAM) timing. The I/O subsystem consists of a peripheral component interconnect (PCI) bridge, a real-time clock, and a number of small computer system interface (SCSI) adapters with hard disks. Unlike other FS simulators, ML-RSIM includes a detailed timing-accurate representation of various hardware components. ML-RSIM does not model any particular system or device, rather it implements detailed general device prototypes that can be used to assemble a range of real machines.

ML-RSIM uses a detailed representation of an OS kernel, Lamix kernel. The kernel is Unix-compatible, specifically designed to run on ML-RSIM and implements core kernel functionalities, primarily derived from NetBSD. Application linked for Lamix can (in most cases) run on Solaris. With a few exceptions, Lamix supports most of the major kernel functionalities such as signal handling, dynamic process termination, and virtual memory management.

3.3.4 ABSS

An augmentation-based SPARC simulator, or ABSS for short, is a multiprocessor simulator based on AugMINT, an augmented Mips interpreter. ABSS simulator can be either trace-driven or program-driven. We have described examples of trace-driven simulators, including the DineroIV, where only some abstracted features of an application (i.e., instruction or data address traces) are simulation. Program-driven simulators, on the other hand, simulate the execution of an actual application (e.g., a benchmark).

Program-driven simulations can be either interpretive simulations or execution-driven simulations. In interpretive simulations, the instructions are interpreted by the simulator one at a time, while in execution-driven simulations, the instructions are actually run on real hardware. ABSS is an execution-driven simulator that executes SPARC ISA.

ABSS consists of several components: a thread module, an augmenter, cycle-accurate libraries, memory system simulators, and the benchmark. Upon execution, the augmenter instruments the application and the cycle-accurate libraries. The thread module, libraries, the memory system simulator, and the benchmark are linked into a single executable. The augmenter then models each processor as a separate thread and in the event of a break (context switch) that the memory system must handle, the execution pauses, and the thread module handles the request, usually saving registers and reloading new ones. The goal behind ABSS is to allow the user to simulate timing-accurate SPARC multiprocessors.

3.3.5 HASE

HASE, hierarchical architecture design and simulation environment, and SimJava are educational tools used to design, test, and explore computer architecture components. Through abstraction, they facilitate the study of hardware and software designs on multiple levels. HASE offers a GUI for students trying to understand complex system interactions. The motivation for developing HASE was to develop a tool for rapid and flexible developing of new architectural ideas.

HASE is based in SIM++, a discrete-event simulation language. SIM++ describes the basic components and the user can link the components. HASE will then produce the initial code ready that forms the bases of the desired simulator. Since HASE is hierarchical, new components can be built as interconnected modules to core entities.

HASE offers a variety of simulations models intended for use for teaching and educational laboratory experiments. Each model must be used with HASE, a Java-based simulation environment. The simulator then produces a trace file that is later used as input into the graphic environment to represent interior workings of an architectural component. The following are few of the models available through HASE:

- Simple pipelined processor based on MIPS
- Processor with scoreboards (used for instruction scheduling)
- Processor with prediction
- Single instruction, multiple data (SIMD) array processors

- A two-level cache model
- Cache coherency protocols (snooping and directory)

3.4. FS Simulators

While lower-complexity systems described thus far are good at exploring and studying single components and component behavior under a smaller application load, for large-scale studies, we must employ FS simulators. In order to build large-scale system, a simulator must observe the scope and level of abstraction. The scope defines what the simulator is modeling, and the level defines the level of detail that is modeled. We can further refine the abstraction level by functionality and timing behavior. If we are trying to model a realistic workload, we must enlarge the scope to include an FS, including hardware and software systems. Each layer must be simulated in sufficient detail to run commercial systems and allow the researcher to tweak the hardware. Obviously, such systems are very complex and yet it is desirable that such systems complete simulations of realistic workloads in reasonable amount of time. For this reason, some FS simulators do not provide cycle-accurate simulations of the processor architecture. Some simulators permit the user to select between very detailed simulations of lower-level architectural components and abstractions of the architecture. The modular nature of some FS simulators allows users to include different ISAs, different microarchitectural designs, different memory system, or different OS kernels. Examples of FS simulator are the Simics simulator [19], Gem5 simulator [26], and SimOS [28].

3.4.1 Simics

'Simics is a widely used FS simulator. It simulates processors at the instruction level, and it is detailed enough to allow the booting of a variety of real OSs. Despite all of its complexity, it still offers performance levels to run realistic workloads. Simics supports a variety of processor architectures including UltraSPARC, Alpha, x86, x86-64, PowerPC, IPF Itanium, MIPS, and ARM. It can support Linux running on x86, PowerPC, and Alpha; Solaris on UltraSPARC; Tru64 on Alpha; and Windows 2000 on x86. Simics can be configured using command-level input. Since Simics simulates an FS, it includes device models capable of running real firmware and device drivers. For instance, Simics modeling the x86 architecture will correctly install and boot Windows XP OS.

Simics is unique in that it can connect several instances (viewed as nodes) and run distributed systems. A single Simics instance can simulate several nodes of a given instruction set architecture (ISA); but using multiple

instances of Simics, communicating through Simics Central, can simulate heterogeneous systems.

At the core of each Simics' simulation is Simics Central, which synchronizes the virtual time between different Simics' simulators and nodes. Simics is configured through an object-oriented paradigm where each object instance is an instance of a processor or device. Any new object is derived using Simics API, which is defined in the configuration. It also supports a runtime Python interpreter for executing Python scripts, which can then be triggered at interesting breakpoints. Devices are a key component of any Simics' simulation; therefore, Simics supports devices that enable OS and firmware to boot and run, respectively. A few of the devices supported by Simics are timers, floppy controllers, keyboard/mouse controllers, direct memory access (DMA) controllers, interrupt controllers, RAM controller, and various other crucial devices needed for FS simulation.

A crucial feature of Simics is its ability to interface with other simulators. Because Simics does not natively support cycle-accurate simulations, it allows interfaces to clock cycle-accurate models written in hardware description languages (such as Verilog).

Another notable feature of Simics is its hindsight feature. A system simulation can progress in two ways, and this opens a new range of testing capabilities. Simics is capable of taking simulation snapshots, and any future simulations can be fast-forwarded to the snapshot and continue execution beyond the snapshot. This is practical for running large simulations at interesting sections multiple times from the saved snapshot.

Applications of Simics are manifold since it is useful for a variety of tasks such as microprocessor design, memory studies, device development, OS emulation and development, and more. Originally, Simics was used primarily for processor designs. Previous trace-based simulators have some known limitations, which Simics tries to resolve. The notable limitations are the inability to simulate multiprocessor systems and its interaction with other devices such as memory and OS memory management and scheduling.

Due to its complexity, Simics requires time to learn all of its features. Moreover, the acquisition of Virtutech by Wind River System and academic licenses do not provide source code, and in many cases, only older OS and processor models are provided to the academic institutions. Yet, Simics is a valuable simulator for conducting research on computer systems architecture.

3.4.2 SimFlex

SimFlex is a simulation framework to support the simulation of uniprocessors, chip multiprocessors, and distributed shared memory systems; the project targets fast and flexible simulations of large-scale systems [21]. The core components of SimFlex are Flexus and SMARTS:

- Flexus: Flexus is the simulator framework, which relies on a set of well-defined components.
- SMARTS: SMARTS is a sampling method whose goal is to reduce the overall simulation time.

SimFlex is built around Simics and thus capable to running FS simulations. Unlike other modular component approaches to system design, SimFlex takes a compile-time component interconnect approach. This means that component interconnects can be optimized at compile time, reducing some runtime overheads. To further reduce simulation times, SimFlex applies SMARTS sampling methodology [25] instead of simulating every cycle.

The timing model around SimFlex is better than that provided natively in Simics. SimFlex operates on a stream of fetched instructions from Simics, allowing more accurate timing models for each instruction. Timing models for x86 and SPARC architectures, for both uniprocessor and multiprocessor configurations, are available with SimFlex.

SimFlex unique features are the notion of abstracting components through a layered approach while improving runtime performance. This is achieved through C++ template tricks around which the components are built, allowing compiler to optimize the overall system. Normally, simulation times of complex systems are reduced either by limiting the number of instructions simulated or by simulating a regularly sampled set of instructions. SimFlex takes the SMARTS [25] approach. SMARTS is a methodology to apply statistical sampling (similar to AP through hardware performance counters). SMARTS uses a coefficient of variation measure to obtain an instruction stream sample while maintaining accuracy of performance estimates. The system uses sampling to warm up components that are prone to have unpredictable state—this usually happens when simulation modes switch from highly accurate to simple models.

SimFlex is a good tool for modeling hardware where timing accuracy is critical.

3.4.3 Gem5 Simulator System

Gem5 simulator system is a combined effort of the previous work of GEMS (general execution-driven multiprocessor simulator) and M5 (a discrete

event-driven simulator). It is written primarily in C++ and to a lesser extent Python. It is open source licensed under a BSD-style license.

Gem5 is primarily built for research environments and provides an abundance of components that work out of box (or without modifications). The Gem5 is modular enough to permit the study of new architectures. Gem5 is designed using object-oriented methodology. Nearly all major components (CPUs, buses, caches, etc.) are designed as objects, internally known as SimObjects. They share similar configurations, initialization, statistics collection, and checkpointing behavior. Internally, the nature of C++ provides for a complex but flexible system description of CPU interconnects, cache hierarchies, multiprocessor networks, and other subsystem. The Gem5 environment relies on domain-specific languages (DSLs) for specialized tasks. For defining ISA's, it inherited an ISA DSL from M5, and for specifying cache coherency, it inherited a cache coherency DSL from GEMS. These languages allow users to represent design issues compactly.

The ISA DSL allows users to design new ISAs through a series of class templates covering a broad range of instructions and arithmetic operations. Cache coherency DSL represents protocols through a set of states, events, and transitions. These features allow Gem5 to implement different coherency protocols through the same underlying state transitions, thus reducing the programmer effort. Gem5 operates on several interchangeable CPU models; a simple CPU model supports a basic 4-stage pipeline, a functional model, an in-order CPU model, while a more advanced out-of-order CPU model with SMT support is also available. It features a flexible event-driven memory system to model complex multilevel cache hierarchies.

Gem5 supports two modes of simulation (or modes of execution): an FS mode that simulates a complete system with devices and an OS and a user-space mode where user programs are serviced with system call emulation (SE). The two modes vary in their support for Alpha, ARM, MIPS, PowerPC, SPARC, and x86-64 targets.

3.4.4 SimOS (Parallel SimOS)

Developed in the late 1990s at Stanford University, SimOS is an FS simulator with features to simulate the OS at different abstraction levels. The development goals were to design a simulator to study FS workloads. Initial version of SimOS was capable of booting and running IRIX (an SGI SVR4 Unix implementation). Later versions permitted modeling of DEC Alpha machines with enough detail to boot DEC Unix systems. Other version of SimOS allowed x86 extensions. SimOS interfaces allow one to include

different number of components. SimOS introduced the concept of dynamically adjustable simulation speed [28]. The simulation speed adjustment balances speed of execution with simulation detail. SimOS implements this through three different execution modes, defining different levels of detail. The three modes are the positioning mode (fastest mode) that offers enough functionality of boot operations and executes the basic OS operations; the rough characterization mode that offers a middle ground with more details but slightly slower than its predecessor; and the accurate mode that is the slowest but most detailed simulation mode. There are two different CPU models provided by SimOS: a basic single-issue pipeline model that provides the basic functionality and a more advanced superscalar dynamically scheduled processor model that supports privileged instructions, memory management unit executions, and exception handling.

The drawbacks of SimOS were addressed by its successor, Parallel SimOS. Parallel SimOS addresses SimOS' scalability issues. Thus, in many ways, Parallel SimOS is similar to the original SimOS, and it takes uses of multithreaded and shared memory capabilities of modern host processors to speedup simulations.

3.4.5 PTLsim

PTLsim is an x86-64 out-of-order superscalar microprocessor simulator [32]. One can think of PTLsim as a simulator and VM in one. It is capable of modeling a processor core at RTL (register transfer level) detail. It models the complete cache hierarchy and memory subsystems including any supportive devices with full-cycle accuracy. The instructions supported are x86- and x86-64-compatible including any modern streaming SIMD extension (SSE) instructions. PTLsim must run on the same host and target platform. Similar to SimOS, PTLsim supports multiple simulating modes.

PTLsim comes as a bare hardware simulator that runs in user space and a more advance version PTLsim/X, which integrates with the Xen hypervisor for full x86-64 simulations. PTLsim/X comes with full multithreading support, checkpointing, cycle-accurate virtual device timing models, and deterministic time dilation without the inherent simulation speed sacrifice. It is capable of booting various Linux distributions and industry standard heavy workloads.

C++ templated classes and libraries give PTLsim the flexibility to tweak and tune significant portions of the simulator for research purposes. It is licensed under a GNU free software license and thus is readily available for public use.

PTLsim employs a technique known as cosimulation, which means that the simulator runs directly on a host machine that supports the simulated instructions. This means that context switched between full-speed (native) simulation and simulated mode is transparent. Likewise, this provides an easy way for code verification. The drawback is that this limits the simulation to x86-based ISAs.

Cosimulation makes PTLsim extremely fast compared to other similar simulators, but other techniques to improve speed are also employed, particularly instruction vectorization. It comes with features to turn on or off various statistics to improve the simulation speed. Since PTLsim is a fast simulator, skipping streams of instructions used in sampling techniques is not necessary. The speed advantage that PTLsim makes is a very good candidate for HP under complex workloads, but because it requires the target platforms be compatible with simulated processors, PTLsim is not as flexible as other FS simulators described here.

Most simulators that were developed in the mid-2000s were designed to support the emergence of multicore architectures and SMT. MPTLsim [33] is an extension of PTLsim to permit SMT like threads.

3.4.6 FeS2

FeS2 is described as a timing-first, multiprocessor, x86 simulator built around Simics though many of its key features were also borrowed from other simulators (e.g., GEMS and PTLsim). It comes with all the standard features available with Simics, but it expands Simics' functionality with more accurate execution-driven timing model. The model includes cache hierarchy, branch predictors, and a superscalar out-of-order core. While the functionality is provided by Simics, the timing model is provided through Ruby, which was developed as part of the GEMS (now a part of Gem5) simulator.

The core is based on a microoperation (μop) model that decodes x86 instructions into a stream of RISC like μops.

FeS2 can be thought of as a separate simulator (technically a library) that wraps around Simics' functionality. The timing component is not FS-capable since it does not implement all the necessary devices required for FS simulation, such as SCSI controllers and disks, PCI and parallel bus interfaces, interrupt and DMA controllers, and temperature sensors (all of which are implemented by Simics).

3.4.7 TurboSMARTS

TurboSMARTS is a microarchitectural simulator that utilizes checkpointing to speed up overall simulation times [35]. In order to speedup simulation time,

simulators often use sampling methods. Sampling involves collecting a large number of brief simulation windows to represent an accurate image of the application's runtime behavior. The problem arises when the collection of snapshots drifts from the native execution. This normally occurs because of instruction data-flow dependencies; normally a branch depends on precomputed values. However, if the values are not present during a sampled event, then the branch might execute a different path than it would if the correct data were present. Therefore, it becomes important to warm the system so that branches execute as if the program flow never halted.

TurboSMARTS provides accurate warmed up states that are stored as checkpoints in order to speedup simulation. It is based on observations that only a small amount of microarchitectural states is accessed during sampling and that checkpointing is a fast and more accurate method than other statistical sampling alternatives.

TurboSMARTS is derived from SimpleScalar's sim-outorder simulator (a superscalar out-of-order detailed simulator). Because of sampling, the simulations times are only a fraction of those using full out-of-order (OoO) simulators, yet the error rates (in terms of CPI) are in the range of $\pm 3\%$.

3.5. Network Simulators

3.5.1 NePSim

NePSim is a network processor simulator with a power evaluation framework [3]. Network processors are a special type of processors specifically used as network routers. A network processor consists of a few simple cores that exploit the inherent data-level parallelism associated with networking codes. With the increasing clock frequency of these chips, power dissipation becomes an issue that must be considered. NePSim offers a simulation environment to explore the design space of speed and power.

NepSim includes cycle-accurate simulations, a formal verification engine, and a power estimator for network processor clusters that may consist of several components (memory controllers, I/O ports, packet buffers, and buses). It implements the Intel IXP1200 system consisting of a StrongARM processor, six microengines (MEs), standards memory interfaces, and high-speed buses.

The input to NepSim is a stream of network packets generated through a traffic generator module. The core processor is not crucial in controlling critical data paths; thus, it is not simulated. The core simulation is the ME core. NepSim simulates the instruction lookup, instruction decoding and source register address formation, reading operands from the source

register, arithmetic logic unit (ALU) operations, and writing results to destination registers. Memory timing information tries to resemble that of the Intel IXP1200s.

3.5.2 Orion

Orion is a power-performance simulator based on the LSE, a modularized microarchitectural simulator [29]. Guided by hierarchical modeling methodology described in Ref. [5], Orion abstracts a network model with building blocks (modules), which describe an interconnection between various parts of a network on chip system. Orion consists of two basic component classes: message transporting class and message processing class. Message-transporting classes consist of message sources and message sinks. Message processing classes consist of router buffers, crossbars, arbiters, and links.

Every module can be parameterized to represent various configurations at the design stage. The goal of Orion is to allow for an extensive range of architectural choices with relatively few reusable modules.

Power modeling in Orion is analytical based on components, which occupy the largest area (about 90%) such as first-in-first-out (FIFO) buffers, crossbars, and arbiters. The power model is based on a set of equations derived from a combination of cache access and cycle timing model tool (CACTI) [16] and WATTCH [18]. The goal is to provide reasonable power estimates through parameterized architectural-level power models. An updated version Orion 2.0 comes with some new features and an updated transistor technology database.

The hierarchical and modular composition of Orion allows us to explore different architectural compositions and workloads.

3.5.3 Garnet

Garnet is described as a cycle-accurate on-chip network model for an FS simulator [7]. The model utilizes the FS capabilities of the GEMS framework [68] modeling a five-stage pipelined router with a virtual channel (VC) flow control. Garnet enables researchers to evaluate system-level optimizations, new network processors simulated on an FS workload.

Garnet models a configurable classic five-stage VC router. The major components are the input buffers, route computation logic, virtual channel allocation (VCA), switch allocator, and a crossbar switch. Router's micro-architectural components consist of single-ported buffers and a single shared port into the crossbar from each input. Garnet's interaction between various memory systems (e.g., cache controllers) is handled by an interconnection

network interface. This means that CPU cache misses are broken into lower-level units and passed onto the interconnect interface. To model a variety of coherency protocols, Garnet implements a system-level point-to-point ordering mechanism, meaning that a message sent from two nodes to a single destination is received in the same order as they were sent. Network power model is implemented from Orion [5]. Garnet records per component events and records them using power counters; these are then used to compute the final energy usage of various router components. There are several input parameters that can be adjusted to fit any desired interconnect such as network type, model detail, simulation type (network only vs. FS), number of router pipeline stages, VC buffer size, and number of bytes per flit (flit is the smallest measurable unit). The models are validated against results in previously published papers.

3.5.4 Topaz

Topaz is an open-source interconnection network simulator for chip multiprocessors and supercomputers [9]. To accommodate the growing need to efficiently simulate large chip interconnects, Topaz offers an open-source multithreaded cycle-accurate network simulator. The ability to interface with GEMS [68] and Gem5 [26] FS simulation frameworks allows Topaz to simulate network interconnects within an FS environment. Topaz was designed for use for studying supercomputers (off-chip network) systems, system-on-chip, to chip-multiprocessor systems.

Topaz is based on a previous simulator (SICOSYS [69]), but for reusability and modularity, the tool is designed using object-oriented methodology and is implemented in C++. The simulation phase consists of several phases. In the build phase, the system will hierarchically outline the network topology according to a set of parameters. During the execution phase, every component is visited and simulated every cycle. Lastly, the print phase will output the collected records of the simulation. The results can be configured to measure various network parameters.

The general simulator breakdown can be described in terms of object constructors, components and flows, traffic patterns, and simulator. Object constructs interpret the input parameters and are responsible for building the required objects for simulation. Components and flows represent hardware to be simulated (functional network units). Traffic patterns are responsible for injecting packets into the network for stand-alone simulation use. The simulator is the main driver and it is responsible for initialization and configuration interpretation.

To build a simulation environment, the user can specify the following parameters:

- Simulation parameters: the general simulation definition that defines traffic pattern and distribution, applied load, message length, etc.
- Network interconnect: this defines the network type through network topology, dimensions, and interconnection delays.
- Router microarchitecture: this describes the router elements such as memories, switches, and multiplexers. Topaz also comes with a few out-of-the-box models that the user can choose and build. To improve its scalability and optimize its performance, Topaz is designed as multi-threaded application. Synchronization between individual threads is done using barriers. For GEMS and Gem5 integration, Topaz provides an API allowing the user to integrate new designs effortlessly.

3.5.5 MSNS: A Top-Down MPI-Style Hierarchical Simulation Framework for Network on Chip

MSNS is an MPI-style network simulator, which aims at accurately simulating packet delays for NoC systems. Based on SystemC, MSNS implements simulation at various granularities from high-level abstraction to low RTL [11]. MSNS uses top-down approach of all network layers and utilizes wait() functions to guarantee cycle accuracy of individual layers. This leads to the following benefits of MSNS: It can simulate traffic of practical systems more precisely by ensuring an accurate abstraction of application's data transmission rates, it can accurately estimate dynamic power consumption, and it can provide methods to evaluate performances of parallel algorithms and high-performance applications on network infrastructure.

MSNS employs several interface levels. RTL modeling is performed at the link layer and network layer. The interface layer is composed of the MPI library, RTL description, and high-level abstraction descriptions. The simulator consists of two components: MSNS generator and MSNS simulator. By utilizing user supplied input files, the MSNS generator generates the network and application properties and stores them as a configuration file. To ensure highest possible accuracy, MSNS generator calculates wire latencies and bit-error probabilities through its built-in wire models. These delays are enforced by a wait function. When the libraries (SystemC, MPI, and design modules) are loaded into the MSNS simulator, they are combined with the previously generated configuration files and the simulation initiates. To calculate power at each individual layer MSNS provides a power estimation for the whole network infrastructure at the architectural level.

3.5.6 Hornet

Hornet is a parallel, highly configurable, cycle-level multicore simulator [13] intended to simulate many-core NoC systems.

Hornet supports a variety of memory hierarchies interconnected through routing and VC allocation algorithms. It also has the ability to model a system's power and thermal aspect.

Hornet is capable of running in two modes: a network-only mode and a full-multicore mode running a built-in MIPS core simulator. The goal of Hornet is the capability to simulate 1000 core systems.

Hornet's basic router is modeled using ingress and egress buffers. Packets arrive in flits and compete for the crossbar. When a port has been assigned, they pass and exit using the egress buffer. Interconnect geometry can be configured with pairwise connections to form rings, multilayer meshes, and tori. Hornet supports oblivious, static, and adaptive routing. Oblivious and adaptive routing are configurable using routing tables. Similarly VCA is handled using tables. Hornet also allows for internode bidirectional linking. Links can be changed at every cycle depending on dynamic traffic needs.

To accurately represent power dissipation and thermal properties, Hornet uses dynamic power models based on Orion [5], combined with a leakage power model, and a thermal model using HotSpot [70].

To speedup simulation, Hornet implements a multithreaded strategy to offload simulation work to other cores. This is achieved by tiling each simulated processor core to a single thread. Cycle accuracy is globally visible for simulation correctness.

Hornet simulates tiles where each tile is an NoC router connected to other tiles via point-to-point links.

The traffic generators are either trace-driven injectors or cycle-level MIPS simulators.

A simple trace injector reads a trace file that contains traces annotated with timestamps, packet sizes, and other packet information describing each packet. The MIPS simulator can accept cross compiled MIPS binaries. The MIPS core can be configured with varying memory levels backed by a coherency protocol.

3.6. Power and Power Management Tools

Computer architecture research revolves around estimating trade-offs between alternative architectural modifications. The trade-offs typically involve performance, hardware complexity, and power budgets. In order

to estimate the power requirements of various design choices, researchers rely on power modeling and power management tools that offer validated models for power estimation.

3.6.1 CACTI

Perhaps the most widely used tool in this realm is the CACTI [16]. CACTI is an analytical model for a variety of cache components that contribute to the overall CPU memory subsystem. The earlier versions of CACTI modeled access and cycle times of on-chip caches. It supports both direct mapped and associative caches. CACTI is an extension to previous models that includes a tag array model, nonstep stage input slopes, rectangular stacking of memory subarrays, a transistor-level decoder model, column-multiplexed bitlines, cycle times and access times, and others. The analytical model in CACTI is validated through the HSPICE [71]. CACTI's access and cycle times are derived by estimating the delays in the following parameters: decoder, wordlines, bitlines, sense amplifiers, comparators, multiplexor drivers, and output drivers (data and valid signal output). Each delay is estimated separately, and the results are combined to estimate the final access and cycle times. Typically, the user describes cache parameters such as cache size, block size, and associativity, as well as process parameters. Describing the models of individual components is beyond the scope of this chapter; thus, the interested reader is referred to Ref. [16].

The tool evolved significantly over the past decade with enhancements to include an integrated approach: modeling access times, cycle times, area, aspect ratio, and power. As the fabrication technologies have changed over time, so have the models present within CACTI. CACTI's models have changed to reflect changes in feature sizes and technologies. CACTI now supports memories built using either SRAM or DRAM and even 3D packaging of DRAMs.

The latest CACTI tool focuses on the interconnect delays between cache subcomponents. The major extensions include the ability to model non-uniform cache access and the ability to model different types of wires. There is also a web-based GUI to facilitate a broader audience.

3.6.2 WATTCH

WATTCH is a framework for architectural-level power analysis and optimization [18]. The framework aims at facilitating power analysis and optimization early in the architectural design phase. Other tools offer highly

accurate power estimates only after the initial layout or floor planning. In contrast, WATTCH offers power estimates early in the design phase.

The power models in WATTCH are interfaced with SimpleScalar. WATTCH uses a modified version of sim-outorder (a superscalar out-of-order processor simulator) to collect their results. SimpleScalar provides simulation environment that keeps track of units that are involved in each cycle. WATTCH records the total energy consumed by each unit on each instruction and accumulates the total energy consumed by an application. WATTCH power model includes four subcategories:

- Array structures: Data and instruction caches, cache tag arrays, register files, alias tables, branch predictors, and portions of instruction window and load/store queues
- Fully associative content-addressable memories: Instruction window/reorder buffer wake-up logic, load/store order checks, and TLBs
- Combinational logic and wires: Functional units, instruction window selection logic, dependency check logic, and result buses
- Clocking: Clock buffers, clock wires, and capacitive loads

The models are similar to the early versions of CACTI tool with two key differences: WATTCH disregards the overall critical path but uses an expected critical path to determine a delay within a unit, and it only measures the capacitance of an entire unit to model power consumption rather than individual stages of a unit. WATTCH also offers models to measure varying conditional clocking styles. This means that WATTCH tracks the number of used ports and depending on the enabled option tracks full, low, or dynamic power consumptions.

Additional power estimation means adding additional overhead to sim-outorder; therefore, WATTCH performs at a 30% overhead well within a tolerable amount.

3.6.3 DSSWattch

DSSWattch [20] is a modification of WATTCH [18] framework. DSSWattch stands for Dynamic SimpleScalar; thus, DSSWattch is a tool for power estimation in dynamic SimpleScalar. The tool is simulates the PowerPC architecture. The tool operates on a new version of SimpleScalar that supports dynamic features for programs such as Jikes RVM Java VM. The additional features support actions such as dynamic compilation and dynamic memory mapping or programs that require dynamic linking.

The new features in DSSWattch are extended functionality of WATTCH through better register file modeling, floating-point capabilities, and support for the 32-bit mixed data-width PowerPC architecture.

Major differences between WATTCH and DSSWattch are the following:

- Operand harvesting to better serve the PowerPC architecture operand harvesting has been updated with new structures to accommodate the PowerPC instruction's ability to update special purpose registers as well as required additional operands.
- Handling of floating-point operands and population counting original WATTCH was unable to differentiate between floating point and integer registers. In order to estimate dynamic bus power usage, DSSWattch extends this functionality through the new operand harvesting data.
- Register file modeling: The original WATTCH tool only handled single integer register file of uniform word length. The PPC ISA requires both 32-bit integer and 64-bit floating-point register files. Therefore, DSSWattch extends this functionality by including multiple register files. Power consumed by these register files is computed separately and accumulated for overall power consumption.
- Differentiation of integer, floating-point, and address data widths: Along with the use of different register files, DSSWattch models different data widths for floating point and integer operations.

Since DSSWattch is an extension of the original WATTCH tool, the power models are based on WATTCH. The rename logic, instruction window, and the unified RUU (register update unit) reflect the microarchitectural state of the simulated out-of-order simulator.

3.6.4 SimplePower

SimplePower is a framework that utilizes several outstanding component-based simulators and combines them with power estimation models. It is an execution-driven, cycle-accurate, RTL-level power estimation tool.

SimplePower is based on a five-stage pipelined simulator with an integer ISA similar to SimpleScalar. The components of SimplePower are SimplePower core, RTL power estimation interface, technology-dependent switch capacitance tables, cache/bus simulator, and loader. At each cycle, the core simulates the instructions and calls corresponding power units. The power estimation modules are C routines for each unit. With new technologies, only the technology-dependent capacitance tables need be changed, while the core remains intact. The cache simulation is interfaced with

DineroIII [4] cache simulator and integrated with a memory energy model. The bus simulator snoops appropriate lines and utilizes an interconnect power model to compute switch capacitance of the on-chip buses. The capacitance tables are based on the design of the functional units. The units can be bit-dependent or bit-independent. Bit-independent functional units are units where operations of one bit do not depend on the value of other bits. Examples of bit-independent units are pipeline registers, logic unit in the ALU, latches, and buses. Bit-dependent functional units are units where the value of one bit depends on the value of other bits in the vector. The transitional matrix to compute capacitance values for bit-dependent functional units becomes very large to efficiently estimate the consumed power. SimplePower uses a combination of an analytical model and functional unit partitioning to estimate the power for such units. For example, the memory module is modeled analytically, while adders, subtracters, multipliers, register files, decoders, and multiplexers are broken into subcomponents whose power is computed as the sum of the individual subcomponents using lookup tables.

3.6.5 AccuPower

Similar to WATTCH and SimplePower, AccuPower [24] is a tool to estimate power consumption for a superscalar microprocessor. AccuPower describes itself as a true hardware-level and cycle-level microarchitectural simulator with the ability to estimate power dissipation of actual complementary metal-oxide semiconductor (CMOS) layouts of critical data path. The ability to obtain transition counts from higher-level blocks such as caches, issue queues, reorder buffers, and functional units at RTL allows AccuPower to accurately estimate switching activity.

The benefit of AccuPower over its predecessors is that AccuPower does not rely on the original SimpleScalar simulator, which combines several critical data paths into a single unit. The problem with other tools is that in many cases, these critical components contribute to more than half of overall power dissipation.

Several AccuPower features include the following:

- It uses a cycle-level simulation of all major data path and interconnection components (issue queue, register files, reorder buffers, load–store queue, and forward mechanisms).
- It provides detailed and accurate on-chip cache hierarchy simulations.
- AccuPower models CPU-internal interconnections in great detail, for example, explicit data transfers and forwarding and clock distribution network.

- It supports three major built-in models of widely used superscalar data paths.
- It includes facilities to collect power and performance estimates at both individual subcomponent level per bit and byte and the entire processor.
- AccuPower uses subcomponent power coefficient estimates using SPICE.

The major components of AccuPower are a microarchitectural simulator, the very large-scale integration (VLSI) layouts for major data paths and caches, and power estimation modules that use energy coefficients obtained from SPICE.

- The microarchitectural simulator is really a greatly enhanced version of the SimpleScalar simulator.[1] The enhancements include implementations for the issue queue, reorder buffer, rename tables, physical register files, and architectural register files. The microarchitectural simulator is modified to fit three widely used versions of superscalar processors. The out-of-order simulator now supports a true cycle-by-cycle instruction execution. The CPU cache structure now supports a true cycle aware cache pipeline. Caches are also bus aware due to the possibility that instruction and data caches can access L2 at the same time. The ultimate goal is to design a simulator that closely resembles the real-world cycle-by-cycle-accurate superscalar processor.
- In order to get accurate energy and power coefficients, the power models rely on models derived from VLSI layouts using SPICE. To obtain accurate energy dissipation, CMOS layouts for on-chip caches, issue queues, physical register files, architecture register files, and reorder buffers are used to obtain real power coefficients.
- The tool's execution is sped up using multithreading. Analysis routines are handled in a separate thread.

AccuPower can be used to obtain

1. raw data collection pertaining to (a) bit-level data-path activity and (b) occupancy of individual data-path resources as measured by the number of valid entries,
2. record accurate data-path component-level power estimation,
3. exploration of power reduction techniques,
4. exploration of alternative circuit-level techniques,
5. exploration of alternative data-path architectures.

[1] According to authors, only 10% of the original code is untouched.

AccuPower remains a great tool to estimate power dissipation for various superscalar data paths as well as explore new microarchitectural innovations.

4. CONCLUSIONS

The topic of application tuning, optimization, and profiling and hardware simulation is vast, and with over 30 years of research, there is an abundance of literature covering this area. Our aim was to expose relevant information and popular tools to the reader. In this chapter, we have covered a variety of popular tools pertaining to hardware and AP.

In the chapter's introduction, we introduced a tool taxonomy and outlined their differences in scope and functionality. Furthermore, we have categorized the AP tools based on their data gathering methodology and categorized HP tools based on their scope. In the chapter's subsections, we provided diagrams that show various tool interdependencies and their categories related to our taxonomy. We have discussed several AP methods for performance tuning such as utilizing hardware performance counters and binary instrumentation and translation. To introduce and elaborate each method and category, we collected tools ranging from hardware performance libraries to binary instrumentation, translation, and code manipulation frameworks.

In the area of hardware simulation, we have covered widely known tools used for simulation and architectural research (e.g., SimpleScalar and Simics). To expose the reader to the chronological picture, we have covered several historical tools that are still used as the basis of many newer tools (e.g., SimpleScalar and SimOS). Because the area of hardware simulation is still growing, our aim was to correlate hardware profiler components and scope in a manner that will allow the reader to explore further in each area. For example, we categorized hardware profilers based on their complexity related to the number of hardware components simulated. Due to the various, often distinct, areas of hardware simulation, we also covered tools used for network interconnection simulations and power consumption. Network and power consumption research is increasingly important due to the advent of multicore and multiprocess machines. Accurate network simulators are becoming a vital part for system simulation and research in high-performance computing and network research areas. Therefore, we have covered several tools in that area.

Finally, as the density of transistors per processor die increases and the CMOS manufacturing technology shrinks, power estimation is becoming

a critical part of future processor designs. Therefore, we have included some currently available tools for power estimation and modeling research. All tools that were picked in this chapter are summarized in Sections 5 and 6.

5. APPLICATION PROFILERS SUMMARY

Tool	Description
gprof	Compiler-assisted application profiler
Parasight	Parallel application profiler-based gprof
Quartz	A tool for tuning parallel program performance
ATOM	A tool for building static instrumentation analysis tools
Pin	A dynamic binary instrumentation tool for building analysis tools
DynInst	A dynamic binary instrumentation framework designed for code patching
Etch	A binary rewrite system for Windows32-based x86 executables
EEL	Machine- and system-independent executable editing library for building analysis tools
Valgrind	A dynamic binary instrumentation framework for code manipulation and error detection
DynamoRIO	Runtime code manipulation system for optimizations and error detection
Dynamite	A binary translation tool
UQBT	Static binary translation tool for RISC-, CISC-, and stack-based machines
OProfile	System-wide profiler for Linux (uses HW counters)
AMD CodeAnalyst	AMD chip-based performance analyzer
Intel VTune	Intel-based performance analysis tool
HPCToolkit	A parallel performance suite for measuring HPC applications
TAU	A portable profiling and tracing tool for parallel programs
Open SpeedShop	A open-source community-based multiplatform Linux-based performance analysis tool
VAMPIR	Visualization and Analysis tool for MPI Resources

PAPI	Portable application performance interface for hardware performance counters
perfmon	Kernel interface for hardware performance counters for performance analysis tools
perfctr	Linux-based hardware performance counter interface driver
gdb	The GNU debugger
DDT	An application debugger for large-scale parallel high-performance applications

6. HARDWARE PROFILERS SUMMARY

Tool	Tool Description
DineroIV	Trace-driven single-process cache simulator
DRAMSim	Open-source DDR memory system simulator
EasyCPU	Educational tool for teaching computer organization
SimpleScalar	A superscalar simulator
SimpleScalar–Alpha	The SimpleScalar simulator ported to Alpha/Linux
SimplePower	Execution-driven data-path energy estimation tool based on SimpleScalar
ML_RSIM	Detailed execution-driven simulator running a Unix-compatible operating system
M-Sim	A multithreaded extension to SimpleScalar
ABSS	An augmentation-based SPARC simulator
AugMINT	An augmented MIPS interpreter
HASE	Hierarchical architecture design and simulation environment
Simics	An FS simulator
SimFlex	A system simulator that targets fast, accurate, and flexible simulation of large-scale systems
Flexus	The SimFlex framework component for FS simulation
Gem5	A collaborative simulation infrastructure targeting FS simulation

SimOS	A machine simulation environment designed for uni- and multiprocess machines
PTLsim	Open-source, FS, timing simulator. With SMP support, OoO core and AMD64 architecture
MPTLsim	Multithreaded version of PTLsim
FeS2	Timing-first, multiprocessor, x86 simulator, implemented as a module for Simics, uses PTLsim for decoding of μops
TurboSMARTSsim	A fast and accurate timing simulator
NepSim	A network processor simulator with power evaluation framework
Orion	Power-performance simulator based on LSE for NoC systems
Garnet	Network model within an FS simulator
Topaz	Open-source network interconnect simulator within an FS environment
MSNS	MPI-style network simulator targeting accurate packet delay measurement of NoC systems
Hornet	A parallel, highly configurable, cycle-level multicore simulator for many-core simulations
CACTI	Cache access and cycle time power estimation
WATTCH	A framework for architectural-level power analysis and optimization
DSSWATCH	Dynamic power analysis framework based on WATTCH
SimplePower	Execution-driven data-path energy estimation tool based on SimpleScalar
AccuPower	A hardware-level power estimation simulator
CGEN	Red Hat's CPU tools generator
LSE	Liberty simulation environment, framework for architectural component design

REFERENCES

[1] M.D. Hill, J.R. Larus, A.R. Lebeck, M. Talluri, D.A. Wood, Wisconsin architectural research tool set, SIGARCH Comput. Archit. News 21 (1993) 8–10.
[2] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, B. Jacob, DRAMsim: a memory system simulator, SIGARCH Comput. Archit. News 33 (4) (2005) 100–107.

[3] Y. Luo, J. Yang, L.N. Bhuyan, L. Zhao, NePSim: a network processor simulator with a power evaluation framework, IEEE Micro 24 (5) (2004) 34–44.
[4] M.D. Hill, J. Edler, DineroIV Trace-Driven Uniprocessor Cache Simulator. http://www.cs.wisc.edu/~markhill/DineroIV, 1997.
[5] X. Zhu, S. Malik, A hierarchical modeling framework for on-chip communication architectures [soc], in: IEEE/ACM International Conference on Computer Aided Design, 2002, ICCAD 2002, November, 2002, pp. 663–670.
[6] Holon Inst. Technology, Easy CPU H.I.T. http://www.hit.ac.il/EasyCPU/, 2008.
[7] N. Agarwal, T. Krishna, L.-S. Peh, N.K. Jha, Garnet: a detailed on-chip network model inside a full-system simulator, in: IEEE International Symposium on Performance Analysis of Systems and Software, 2009, ISPASS 2009, April, 2009, pp. 33–42.
[8] T. Austin, E. Larson, D. Ernst, SimpleScalar: an infrastructure for computer system modeling, Computer 35 (2002) 59–67.
[9] P. Abad, P. Prieto, L.G. Menezo, A. Colaso, V. Puente, J.-A. Gregorio, Topaz: an open-source interconnection network simulator for chip multiprocessors and supercomputers, in: 2012 Sixth IEEE/ACM International Symposium on Networks on Chip (NoCS), May, 2012, pp. 99–106.
[10] L. Schaelicke, M. Parker, ML-Rsim. http://www.cs.utah.edu/lambert/mlrsim/index.php, 2005.
[11] Z. Li, X. Ling, J. Hu, Msns: A top-down mpi-style hierarchical simulation framework for network-on-chip, in: WRI International Conference on Communications and Mobile Computing, 2009, CMC '09, vol. 2, January, 2009, pp. 609–614.
[12] J.J. Sharkey, D. Ponomarev, K. Ghose, Abstract M-SIM: a flexible, multi-threaded architectural simulation environment. http://www.cs.binghamton.edu/jsharke/m-sim/documentation/msim_tr.pdf, 2006.
[13] P. Ren, M. Lis, M.H. Cho, K.S. Shim, C.W. Fletcher, O. Khan, N. Zheng, S. Devadas, Hornet: a cycle-level multicore simulator, IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 31 (6) (2012) 890–903.
[14] M. Flynn, D. Sunada, D. Glasco, ABSS v2.0: a SPARC simulator, Technical Report, 1998.
[15] A.-T. Nguyen, M. Michael, A. Sharma, J. Torrella, The Augmint multiprocessor simulation toolkit for Intel x86 architectures, in: Proceedings of the 1996 International Conference on Computer Design, VLSI in Computers and Processors, ICCD '96, IEEE Computer Society, Washington, DC, 1996, p. 486.
[16] S.J.E. Wilton, N.P. Jouppi, CACTI: an enhanced cache access and cycle time model, IEEE J. Solid-State Circuits 31 (1996) 677–688.
[17] P.S. Coe, F.W. Howell, R.N. Ibbett, L.M. Williams, A hierarchical computer architecture design and simulation environment, ACM Trans. Model. Comput. Simul. 8 (1998) 431–446.
[18] V.T. Brooks, M. Martonosi, Wattch: a framework for architectural-level power analysis and optimizations, in: Proceedings of the 27th International Symposium on Computer Architecture, June 2000, 2000, pp. 83–94.
[19] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, B. Werner, Simics: a full system simulation platform, Computer 35 (2002) 50–58.
[20] J. Dinan, E. Moss, Dsswattch: power estimations in dynamic simplescalar, Technical Report, 2004.
[21] N. Hardavellas, S. Somogyi, T.F. Wenisch, E. Wunderlich, S. Chen, J. Kim, B. Falsafi, J.C. Hoe, A.G. Nowatzyk, Simflex: a fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture, SIGMETRICS Perform. Eval. Rev. 31 (2004) 31–35.
[22] W. Ye, N. Vijaykrishnan, M. Kandemir, M.J. Irwin, The Design and Use of SimplePower: A Cycle-Accurate Energy Estimation Tool, ACM, California, USA, 2000, pp. 340–345.

[23] Flexus 4.1.0, http://parsa.epfl.ch/simflex, 2005.
[24] D. Ponomarev, G. Kucuk, K. Ghose, AccuPower: an accurate power estimation tool for super-scalar microprocessors, in: Proceedings of the Conference on Design, Automation and Test in Europe Conference and Exhibition, 2002, pp. 124–129.
[25] R.E. Wunderlich, T.F. Wenisch, B. Falsafi, J.C. Hoe, SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling, SIGARCH Comput. Archit. News 31 (2) (2003) 84–97.
[26] N. Binkert, B. Beckmann, G. Black, S.K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D.R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M.D. Hill, D.A. Wood, The gem5 simulator, SIGARCH Comput. Archit. News 39 (2011) 1–7.
[27] CGEN, The CPU Tools Generator, 2009, www.sourceware.org/cgen.
[28] M. Rosenblum, S.A. Herrod, E. Witchel, A. Gupta, Complete computer system simulation: the SimOS approach, IEEE Concurrency 3 (1995) 34–43.
[29] M. Vachharajani, N. Vachharajani, D.A. Penry, A. Blo Jason, S. Malik, D.I. August, The liberty simulation environment: a deliberate approach to high-level system modeling, ACM Trans. Comput. Syst. 24 (2006) 211–249.
[30] R. Lantz, Parallel SimOS: Scalability and performance for large system simulation, Ph.D. Thesis, Stanford University, 2007.
[31] Red Hat Inc. Red hat's sid. http://sourceware.org/sid/sid-guide.pdf, 2001.
[32] M.T. Yourst, PTLsim: a cycle accurate full system x86-64 microarchitectural simulator, in: ISPASS, IEEE Computer Society, San Jose, CA, 2007, pp. 23–34.
[33] H. Zeng, M. Yourst, K. Ghose, D. Ponomarev, MPTLsim: a cycle-accurate, full-system simulator for x86-64 multicore architectures with coherent caches, SIGARCH Comput. Archit. News 37 (2) (2009) 2–9.
[34] C.J. Mauer, M.D. Hill, D.A. Wood, Full-system timing-first simulation, SIGMETRICS Perform. Eval. Rev. 30 (1) (2002) 108–116.
[35] T.F. Wenisch, R.E. Wunderlich, B. Falsafi, J.C. Hoe. TurboSMARTS: accurate microarchitecture simulation sampling in minutes, Technical Report, SIGMETRICS Performance Evaluation Review, 2005.
[36] P.J. Mucci, S. Browne, C. Deane, G. Ho, PAPI: a portable interface to hardware performance counters, in: Proceedings of the Department of Defense HPCMP Users Group Conference, 1999, pp. 7–10.
[37] S.L. Graham, P.B. Kessler, M.K. McKusick, gprof: a call graph execution profiler (with retrospective), in: Best of PLDI, 1982, pp. 49–57.
[38] J. Levon, P. Elie, OProfile. http://oprofile.sourceforge.net, 2003.
[39] Z. Aral, Ilya Gertner, Parasight: a high-level debugger/profiler architecture for shared-memory multiprocessor, in: Proceedings of the 2nd International Conference on Supercomputing, ICS '88, ACM, New York, NY, 1988, pp. 131–139.
[40] Advance Micro Devices, AMD CodeAnalyst Performance Analyzer. http://developer.amd.com/tools/CodeAnalyst, April, 2012.
[41] S. Jarp, R. Jurga, A. Nowak, Perfmon2: a leap forward in performance monitoring, J. Phys. Conf. Ser. 119 (4) (2008) 042017.
[42] T.E. Anderson, E.D. Lazowska, Quartz: a tool for tuning parallel program performance, SIGMETRICS Perform. Eval. Rev. 18 (1990) 115–125.
[43] Intel, Intel VTune. http://software.intel.com/en-us/intel-vtune-amplifier-xe, 2013.
[44] A. Srivastava, A. Eustace, Atom: A System for Building Customized Program Analysis Tools, ACM, New York, NY, 1994, pp. 196–205.
[45] N. Tallent, J. Mellor-Crummey, L. Adhianto, M. Fagan, M. Krentel, HPCToolkit: performance tools for scientific computing, J. Phys. Conf. Ser. 125 (1) (2008) 012088.
[46] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, K. Hazelwood, Pin: building customized program analysis tools with

dynamic instrumentation, in: Programming Language Design and Implementation, ACM Press, New York, USA, 2005, pp. 190–200.
[47] S.S. Shende, A.D. Malony, The Tau parallel performance system, Int. J. High Perform. Comput. Appl. 20 (2006) 287–311.
[48] GDB, GDB: the GNU project debugger. http://www.sourceware.org/gdb.com, 2013.
[49] B. Buck, J.K. Hollingsworth, An API for runtime code patching, Int. J. High Perform. Comput. Appl. 14 (2000) 317–329.
[50] M. Schulz, J. Galarowicz, W. Hachfeld, Open SpeedShop: open source performance analysis for Linux clusters, in: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, SC '06, ACM, New York, NY, 2006.
[51] Allinea, Allinea DDT. http://www.allinea.com/products/ddt, 2002.
[52] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, B. Chen, Instrumentation and optimization of Win32/Intel executables using etch, in: Proceedings of the USENIX Windows NT Workshop, 1997, pp. 1–7.
[53] ZIH TU Dresden, VampirTrace. http://www.tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/forschung/software_werkzeuge_zur_unterstuetzung_von_programmierung_und_optimierung/vampirtrace, 2007.
[54] J.R. Larus, E. Schnarr, EEL: machine-independent executable editing, in: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation, PLDI '95, ACM, New York, NY, 1995, pp. 291–300.
[55] N. Nethercote, J. Seward, Valgrind: a framework for heavyweight dynamic binary instrumentation, SIGPLAN Not. 42 (2007) 89–100.
[56] D.L. Bruening, Efficient, transparent and comprehensive runtime code manipulation, Technical Report, 2004.
[57] D. Bruening, T. Garnett, S. Amarasinghe, An infrastructure for adaptive dynamic optimization, in: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, CGO '03, IEEE Computer Society, Washington, DC, 2003, pp. 265–275.
[58] C. Cifuentes, M. Van Emmerik, UQBT: adaptable binary translation at low cost, Computer 33 (3) (2000) 60–66.
[59] M. Fernández, R. Espasa, Dixie: a retargetable binary instrumentation tool, in: Proceedings of the Workshop on Binary Translation, 1999.
[60] J. Souloglou, A, Rawsthorne, Dynamite: a framework for dynamic retargetable binary translation. Technical Report, The University of Manchester, March 1997.
[61] T. Janjusic, K. Kavi, B. Potter, International Conference on Computational Science, ICCS 2011 Gleipnir: A Memory Analysis Tool, Procedia Computer Science, vol. 4, 2011, pp. 2058–2067.
[62] W. Cheng, Q. Zhao, B. Yu, S. Hiroshige, Tainttrace: efficient flow tracing with dynamic binary rewriting, in: Proceedings of the 11th IEEE Symposium on Computers and Communications, ISCC '06, IEEE Computer Society, Washington, DC, 2006, pp. 749–754.
[63] D. Bruening, Q. Zhao, Practical memory checking with Dr. Memory, in: Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '11, IEEE Computer Society, Washington, DC, 2011, pp. 213–223.
[64] Q. Zhao, J.E. Sim, W.F. Wong, L. Rudolph, DEP: detailed execution profile, in: PACT '06: Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques, ACM Press, Seattle, Washington, USA, 2006, pp. 154–163.
[65] J. Hollingsworth, B.P. Miller, J. Cargille, Dynamic Program Instrumentation for Scalable Performance Tools, IEEE Computer Society, Los Alamitos, CA, USA, 1994, pp. 841–850.

[66] W. Korn, P.J. Teller, G. Castillo, Just how accurate are performance counters? in: IEEE International Conference on Performance, Computing, and Communications, April 2001, 2001, pp. 303–310.
[67] B. Nikolic, Z. Radivojevic, J. Djordjevic, V. Milutinovic, A survey and evaluation of simulators suitable for teaching courses in computer architecture and organization, IEEE Trans. Educ. 52 (4) (2009) 449–458.
[68] M.M.K. Martin, D.J. Sorin, B.M. Beckmann, M.R. Marty, M. Xu, A.R. Alameldeen, K.E. Moore, M.D. Hill, D.A. Wood, Multifacet's general execution-driven multiprocessor simulator (gems) toolset, SIGARCH Comput. Archit. News 33 (2005) 92–99.
[69] V. Puente, J.A. Gregorio, R. Beivide, Sicosys: an integrated framework for studying interconnection network performance in multiprocessor systems, in: Proceedings of the 10th Euromicro Workshop on Parallel, Distributed and Network-Based Processing, 2002, pp. 15–22.
[70] K. Skadron, M.R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, D. Tarjan, Temperature-aware microarchitecture, in: Proceedings of the 30th Annual International Symposium on Computer Architecture, June, 2003, pp. 2–13.
[71] T. Wada, S. Rajan, S.A. Przybylski, An analytical access time model for on-chip cache memories, IEEE J. Solid-State Circuits 27 (8) (1992) 1147–1156.

ABOUT THE AUTHORS

Dr. Tomislav Janjusic received his Bachelor of Science in Computer Science from Henderson State University in 2006 and a Ph.D. in Computer Science and Engineering in 2013 from the University of North Texas. Dr. Janjusic joined Oak Ridge National Laboratory in Oak Ridge Tennessee as a postdoctoral research associate in July 2013. His main focus is on application performance analysis and communication library development. Dr. Janjusic's research area is in computer systems with emphasis on computer architecture, memory allocation techniques, cache system performance, and parallel computing.

Dr. Krishna Kavi is currently a Professor of Computer Science and Engineering and the Director of the NSF Industry/University Cooperative Research Center for Net-Centric Software and Systems at the University of North Texas. During 2001–2009, he served as the Chair of the department. Previously, he was on the faculty of University of Alabama in Huntsville and the University of Texas at Arlington. His research is primarily on Computer Systems Architecture including multithreaded and multicore processors, cache memories, and hardware-assisted memory managers. He also conducted research in the area of formal methods, parallel processing, and real-time systems. He published more than 150 technical papers in these areas. He received his Ph.D. from Southern Methodist University in Dallas Texas and a BS in EE from the Indian Institute of Science in Bangalore, India.