Semi-automatic Validation of Cycle-Accurate Simulation Infrastructures: The Case for gem5-x86

Juan M. Cebrian[1], Adrián Barredo, Helena Caminal, Miquel Moretó, Marc Casas, Mateo Valero

Barcelona Supercomputing Center (BSC), Barcelona, Spain

Abstract

Since the early 70s, simulation infrastructures have been a keystone in computer architecture research, providing a fast and reliable way to prototype and evaluate ideas for future computing systems. There are different types of simulators, from most detailed (cycle-accurate) to time-based/functional and analytical modeling. Increasing accuracy translates into several orders of magnitude in terms of simulation speed. Yet, a question remains open: are the results derived from the simulation infrastructure representative of a real machine?

Validation of these infrastructures is complex and costly, usually performed upon release. However, most simulators do not provide the appropriate means to verify or validate new architectural models. In this paper, we introduce a semi-automatic validation framework based on real-hardware performance counter information. The framework provides two levels of abstraction: a) a high level definition of the processor behavior (Top-Down model) and b) detailed per-structure and per-pipeline-stage usage breakdown to pinpoint simulator issues. We used this framework to validate the latest available gem5-x86 simulation environment, and found several sources of error that alter the expected behavior of the simulated processor, which we were later to document and correct.

*Juan M. Cebrian

Email addresses: jm.cebriangonzalez@gmail.com (Juan M. Cebrian), adrian.barredo@bsc.es (Adrián Barredo), helena.caminal@bsc.es (Helena Caminal), miquel.moreto@bsc.es (Miquel Moretó), marc.casas@bsc.es (Marc Casas), mateo.valero@bsc.es (Mateo Valero)

Keywords: gem5, Simulator, Validation, SIMD, SSE, x86

1. Introduction

In the last decade, the complexity of modern computer systems has reached unprecedented levels. Prototyping any system component is exceedingly expensive, forcing architects to rely on simulation to model and evaluate new ideas. Simulators produce quantitative estimates in a safe, timely and cost-effective manner. As a matter of fact, architecture researchers have increasingly relied on simulators over the years. Yi *et al.* [1] have classified the performance evaluation methods for papers appearing in the International Symposium on Computer Architecture (ISCA) in six selected years. In the conference's inaugural year (1973), only two papers out of 28 (7.1%) were simulation-based, but that number steadily increased to 27.9% in 1985, 71.9% in 1993, 80% in 1997, and finally to 88% and 87% in 2001 and 2004, respectively. Modern simulators model multiple ISAs (instruction set architecture), CPU (central processing unit) types, interconnect topologies, and other devices with different levels of detail. Complete infrastructures are able to boot unmodified operating systems (OS), and even run interactive graphical workloads. To add further complexity to these systems, architects use a wide variety of methodologies, benchmarks and datasets on top of the simulated infrastructure.

Current simulation speeds often limit the scope and depth of the work that can be performed. Simulation infrastructures can be classified based on the level of detail of the simulation as follows: cycle-level, time-based/functional and analytical modeling, from more to less accurate. Cycle-level accuracy is necessary when working with very specific hardware features that require a high level of detail. Timing simulation and analytical modeling are key enabling techniques to explore the design space and to enable software development on unavailable hardware. Fast simulations of many-core processors at system level are critical, due to the need to emulate the behavior and communication of tens to hundreds

of cores. Hybrid methodologies, like interval simulation, provide some balance between detailed cycle-accurate and time-based/functional simulation. This allows applications with long execution times to be modeled much faster, while still providing the necessary level of detail to observe core, memory, interconnect and system interactions.

In this paper we focus on cycle-level simulators, complex pieces of software that typically take hundreds of thousands of lines of code. Cycle-accurate simulators need to model each of the processor components as accurately as possible, updating state elements at every clock cycle. As a result, insightful information can be retrieved when running realistic applications and datasets. This is key to quantitatively estimate the usage and behavior of specific structures for given sizes, and to design features or configurations in the micro-architecture. Cycle-accurate simulators also enable the evaluation of new hardware features and instructions, providing binary compatibility and enabling performance and energy studies, especially in hardware/software co-design. Unfortunately, this level of accuracy comes at a price: long simulation times and high development/-validation costs. Single-core cycle-accurate simulators can execute around 0.01 to 0.3 million instructions per second (MIPS), leading to simulation times of several days for a couple of minutes of application time.

In essence, while simulation is indubitably one of the most important tools available to computer architecture researchers, designers have to face trade-offs between performance, accuracy and flexibility that inherently lead to a certain degree of experimental error. As a result, several important challenges arise regarding the simulation process that need to be addressed:

Verification, validation and calibration. Implementation and testing of a simulation infrastructure so that it accurately models a state-of-the-art processor is a tedious and laborious process. Formal verification (using specification languages) is usually reserved to simple embedded models, leaving ad-hoc techniques for complex processor models. Simulators are usually verified and validated before release by the developers and, upon release, by other institu-

tions. Black and Shen [2] classified the different sources of simulation error into: a) Modeling errors: erroneous description of the desired functionality. b) Specification errors: the developer is unaware of the internal functionality being modeled. c) Abstraction errors: the developer fails to implement certain details of the system being modeled. Modeling errors can be corrected, but, as years go by, the slow and infrequent updates often result in simulators that model unrealistic, buggy, or even obsolete computer architectures. Many researchers just take simulators "as provided", and do not spend time checking the correctness of the infrastructure built with new tool-chains, running new benchmarks, or validating a modified simulator to represent a system which was not originally intended to model. They do not consider either if abstraction or specification errors, which may not affect overall application performance, produce unacceptable errors in their specific area of research. There is a real necessity to have a simple infrastructure to ensure that the cumulative errors do not lead to unreliable results over the years. There are also another two additional "sources of error" that are worth mentioning.

Input datasets. Long simulation times may be an issue when using cycle-accurate infrastructures. One common "solution" is to use small input datasets to reduce simulation time. However, doing so may lead to different conclusions than studies performed using more realistic input sets, above all if simulated resources are not scaled accordingly. This is certainly true when simulating multi/many core architectures in which synchronization heavily depends on workload distribution. Basically, not only do we need a simulator that models the target architecture in detail, it also must be driven by a realistic workload.

Lack of features. Computer architecture advances quickly and simulation infrastructures have difficulties to match all the architectural features introduced by different companies. This is usually due to their complexity and the lack of permanent developers/resources. While users may expect missing some modern processor features on the simulation infrastructure, they usually underestimate the impact it could have on their proposals (e.g., SIMD, prefetching, write

buffers, etc.). Certain features can be critical for the performance of specific applications or per-structure behavior, and the lack of them can be a real issue. Researchers may propose some architectural changes to solve an issue that is no longer existent, or that can affect applications designed with modern features in mind. It is important to revisit simulator implementation/validation for state-of-the-art processors that are representative of modern systems and ensure they are able to capture the behavior of emerging workloads.

In this paper we address some of the above-mentioned issues, providing researchers with a semi-automatic infrastructure to analyze the resource requirements of different input workloads and to ease the validation process of cycle-accurate simulation infrastructures against real hardware. Our methodology is based on performance counter information and the Top-Down Model approach [3]. Performance counter information is used, in essence, to define the processor behavior for a given workload and to detect hardware bottlenecks. Per-structure and pipeline performance counters are used to pinpoint the sources of error in the simulation infrastructure. We also provide a case study for gem5-x86, describing the bugs found and possible fixes, showing the final behavior. The "fixed" simulator improves both the accuracy of the results and the simulation speed for a relatively up-to-date revision of gem5-x86 (new μops (Micro-operations) and reduced simulation stalls due to different sources of error). The framework will be made available upon publication for the research community [4].

The paper is organized as follows. Section 2 reviews the related work on simulation environments and validation alternatives. Section 3 describes the proposed validation framework. Section 4 describes the infrastructure and methodology used in the article. Section 5 presents a case study performed on gem5-x86. Finally, Section 6 summarizes the main conclusions and future work.

2. Related Work

Simulation infrastructures have become very popular since the late 90s, in a great effort from both the industry and the research community to improve the quality, accuracy and development/evaluation time of novel ideas.

Cycle-level simulators can be seen as high-level abstractions of RTL (register-transfer level) designs implemented over several thousands of lines of code. These complex systems model each of the processor components as accurately as needed, updating state elements at every clock cycle, so that insightful information can be retrieved when running realistic applications and datasets. M5 [5], gemsOpal [6], PTLsim [7] or MARSS [8] are examples of simulation infrastructures that can achieve cycle-accuracy.

Time-based simulation provides a good trade-off between accuracy and simulation speed, processing most of the instructions from the application in a simplified way. MASE is an example of time-based simulation, which combines timing and functional modeling [9]. MASE was built on top of SimpleScalar, one of the first trace-driven simulators available for the research community [10]. Asim or RSIM are other examples of the initial efforts to provide time-based and functional simulation infrastructures [11, 12]. More recently, Kang *et al.* [13] conducted an interesting survey on time-based and functional simulation infrastructures, extending the work done by Yi *et al.* [1], and classifying time-based methods into different categories:

- CPI-Based simulation (k-CPI). This methodology assumes that each instruction takes k cycles to execute them. A simplified model can assume that each instruction requires one cycle to execute (1-CPI model). A more complex model can provide a different CPI value for each instruction or instruction group based on the processor's specification datasheet (datasheet model). k-CPI models can be easily built on top of functional simulators without sacrificing the simulation speed. Commercial processor simulators such as Imperas OVP [14] or Arm FastModels [15] use a k-CPI-based simulation approach. While k-CPI models are easy to understand

and fast to simulate, the accuracy of the results is far from real hardware measurements, especially for corner cases, since this model neglects the complex behavior of modern microprocessor architectures.

- Sampled simulation. Cycle-accurate simulation is performed over a subset of the instruction stream divided into "sampling units". The selection of the sampling units can be done randomly [16], periodically [17], or based on phase analysis [18, 19, 20]. The main issue of this approach is to guarantee that the sampling units properly represent computational and memory patterns of the whole application. Statistical methods can be applied to ensure this. Phase-sampling can be also handled at an application level, providing pre-processed sampling units for complete applications (e.g., SimPoint [21] and SMARTS [17]).

- Statistical simulation. This strategy speeds up architectural simulation by building synthetic traces or benchmarks that are representative for long-running benchmarks [22, 23]. The synthetic trace is built based on the results obtained in two collection phases. The first one collects information about instruction count, types and dependencies. The second phase collects micro-architectural dependent statistics (e.g., cache, branch).

- FPGA-accelerated simulation. As the name suggests, it builds timing models onto field-programmable gate-arrays (FPGAs) [24, 25, 26]. These models demand additional hardware and development time to synthesize the model into hardware, so it is common to offload only performance-critical parts of software-based techniques, rather than performing the whole simulation on the FPGA.

- Hybrid simulation. This methodology tries to obtain the best features from cycle-accurate and functional simulation by dynamically switching between them, while keeping the processor-centric state synchronized. HySim [27, 28] divides the code into processor-specific functions (executed in the cycle-accurate simulator) and target-independent functions, exe-

cuted by the host. MARSS [8] or GEMsOpal [6] support seamless dynamic switching between the cycle-accurate simulation mode and the native x86 emulation mode of QEMU/Simics. gem5 supports KVM-mode, where code execution can be migrated to native hardware for execution and sent back to the simulator, significantly improving the performance [29].

- Control-sensitive simulation. A specific case of sampled simulation that works at a basic block level. A basic block is defined as the code between two branch instructions. Key statistics per basic block can be obtained by running multiple times at different contexts using a timing-accurate simulator (e.g., [30]) or a static worst-case execution time analysis framework [31, 32] such as OTAWA [33] and Absint aiT [34].

- Trace-driven simulation. Timed/functional simulation that uses traces obtained in a real system as inputs. It provides a fast and reliable model for design space exploration, but limited support for architectural changes. Examples of this methodology include CMP$im [35], Graphite [36], Sniper [37], ZSim [38] or MUSA [39].

Finally, analytical simulation/modeling uses mathematical formulas to model the performance of the architecture. Mechanistic models (or white-box models) [40, 41, 42], construct a model based on the mechanics of the target architecture. On the other hand, empirical models (or black-box models) [43], utilize a parameterized performance model, trained using machine learning or regression analysis, without any specific knowledge about the micro-architecture of the target processor. TQSIM uses analytical models based on sampled simulation over disruptive events, such as cache misses and branch misprediction [13].

Regarding validation of simulation infrastructures, each simulation framework has been validated against real architectures of the same period. However, regression tests for simulator modifications when modifying the source code are usually not present. In addition, hardware evolves quickly, and changing the simulator parameters may lead the model to misbehave when compared

with newer hardware, so additional validation methods are required. Zhang *et al.* [44] propose a new methodology for validating simplified simulation models, which focuses on the trends of metric values across benchmarks and architectures, instead of errors of absolute metric values. However, the validation of a simplified model requires a detailed cycle-accurate one. Nyew *et al.* [45] use domain-specific information to effectively capture a potential mismatch between the assumed architecture model and its simulator. They show how a simulator-generated event trace can be fed into an automatically generated verification program, used to verify that the simulator obeys the invariants. Techniques to extract simulator behavior from traces and present the results to the user are also reported.

The works that show similarities to ours were performed by Gutierrez *et al.* [46], Butko *et al.* [47], Akram *et al.* [48] and Walker *et al.* [49]. In the first work, the authors validated the gem5 simulator against a Versatile Express TC2 board (Gutierrez) and a Snowball SKYS9500-ULP-C01 board (Butko), not only in terms of runtime, but also by adding statistics about L2 caches. In addition, Gutierrez *et al.* provided L1 data and instruction cache, TLB (Translation Lookaside Buffer.) together with branch predictor statistics. Their evaluation showed 20% accuracy on average for most of them. They link most of this error to modeling similar but not identical components. Gibson *et al.* [50] also showed the importance of simulator validation to improve accuracy and correctness against real hardware. Akram *et al.* performed an evaluation of different x86 simulation infrastructures, reporting an 80% IPC deviation in gem5 when compared with a modern Haswell-like processor. In contrast to previous works, which are designed for a specific platform, we provide a semi-automatic framework to perform a more detailed evaluation of key system components and pipeline stages (both in terms of instructions and running/stalled cycles).

To conclude, Walker *et al.* designed GemStone, a tool which uses hierarchical clustering, correlation analysis, and regression techniques to identify sources of error without requiring detailed CPU specifications. This allows existing models to be improved, new models to be developed, validation of simulator

changes, and the testing of model suitability for specific use-cases. They also automate the process of characterising hardware platforms, identifying sources of error in gem5 models and quantifying the effect of errors on the performance. This is the closest related work to our validation framework. Their work is fully automated, but lacks the level of detail that our framework can achieve. For example, it is pointed out that they discovered an issue with the branch predictor. This information is also provided by the Top-Down model in our framework. However, other sources of error, such as the one discovered by our framework regarding additional cycles when fetching instructions, or the wrong mapping of instructions to ALUs have not been detected by their tool. We believe this is due to the fact that they have tried to find an overall model focusing on "performance", while giving less importance to per/structure detailed functionality. This may hide specific modeling issues that are hidden by overall model behavior. We provide a high abstraction layer (based on the Top-Down model) to summarize processor behavior for a given workload, together with detailed information as expected from a cycle-level simulation infrastructure. We also propose some fixes, workarounds and solutions to improve the quality of the works based on the gem5-x86 infrastructure.

3. Validation Framework

The validation framework presented in this paper consists of a set of python and bash scripts to semi-automatize the process of profiling, gathering, processing and visualizing statistics. This helps to detect errors in cycle-accurate simulation infrastructures. The same framework can be used to check real-system behavior with different input sizes and extrapolate the expected behavior in the simulation infrastructure [4].

Figure 1 shows an overview of the validation framework. The base of the framework is a python dictionary (base dictionary), composed by a set of unique keys that map to database entries or simple formulas (multi-level addition, subtraction, multiplication or division) with elements of the database. The current

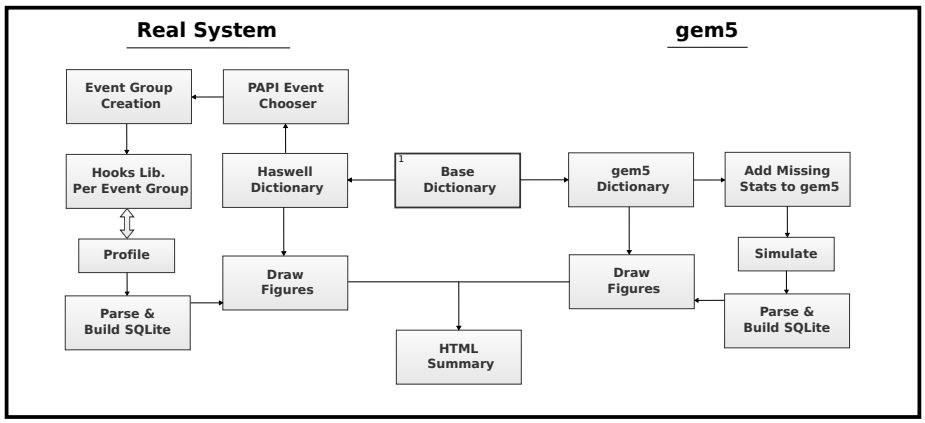

Figure 1: Overview of the Validation Framework.

base dictionary defines keys for all the required information to build the Top-Down model, as well as pipeline cycle/stall activity and per-stage instruction count; reasons for dispatch/execute stalls; usage statistics for L1, L2, L3 caches, TLB and branch predictor (left column in Tables 1 and 2).

Users need to create a dictionary for each pair architecture/simulator they want to validate (e.g., Haswell Dictionary and gem5 Dictionary in Figure 1). The selected keys will be mapped to specific performance counters or simulator statistics using those dictionaries. Input values will be obtained in the profiling/simulation steps and parsed into a SQLite database. Table 1 shows a complete example for an Intel i7-4600U Haswell processor. For example, the *TOTAL CYCLES* key will search for the database entry *perf::CYCLES*. In addition, the key *L2 READ ACCESSES* maps to a formula that adds the L1D and L1I read misses *@(+,perf::PERF COUNT HW CACHE L1D:READ:MISS,perf::PERF COUNT HW CACHE L1I:READ:MISS)*. Formulas can be specified as @(OPERATOR, key1, key2 ... keyN). Each key can be, at the same time, a formula with the same format or an entry from the database. On the other hand, if there is no performance counter or stat available in either platform, the value "none" should be specified. This abstraction layer, built on top of both the simulator statistics and the real hardware performance counter interfaces (e.g., PAPI,

Table 1: Haswell Dictionary. Core Performance Counters.

Dictionary Keys	Associated Database Entry or Formula (Performance Counter or Combination of Counters)
#Top-Down Model	
CLOCKS	CPU_CLK_UNHALTED
SLOTS	4
DECODED_NONE	IDQ_UOPS_NOT_DELIVERED:CORE
ISSUED_ANY	UOPS_ISSUED:ANY
RETIRE_SLOTS	UOPS_RETIRED:RETIRE_SLOTS
RECOVERY_CYCLES	INT_MISC:RECOVERY_CYCLES
#Pipeline-Stalls-Per-Stage	
TOTAL_CYCLES	perf::CYCLES
STALL_CYCLES_IFETCH	ICACHE:IFETCH_STALL
STALL_CYCLES_DECODE	IDQ:EMPTY
STALL_CYCLES_DISPATH	RESOURCE_STALLS:ANY
STALL_CYCLES_ISSUE	UOPS_ISSUED:STALL_CYCLES
STALL_CYCLES_EXECUTE	UOPS_EXECUTED:STALL_CYCLES
STALL_CYCLES_RETIRE	UOPS_RETIRED:STALL_CYCLES
#Instructions-Per-Stage	
MACRO_INST_TOTAL	perf::INSTRUCTIONS
MICRO_INST_ISSUED	UOPS_ISSUED:ALL
MICRO_INST_EXECUTED	UOPS_EXECUTED:CORE
MICRO_INST_RETIRED	UOPS_RETIRED:ALL
MACRO_INST_RETIRED	INSTRUCTIONS_RETIRED
#Cycle Activity	
CYCLE_ACTIVITY:CYCLES_NO_EXECUTE	CYCLE_ACTIVITY:CYCLES_NO_EXECUTE
CYCLE_ACTIVITY:STALLS_L1D_PENDING	CYCLE_ACTIVITY:STALLS_L1D_PENDING
CYCLE_ACTIVITY:STALLS_L2_PENDING	CYCLE_ACTIVITY:STALLS_L2_PENDING
CYCLE_ACTIVITY:STALLS_LDM_PENDING	CYCLE_ACTIVITY:STALLS_LDM_PENDING
CYCLE_ACTIVITY:CYCLES_L1D_PENDING	CYCLE_ACTIVITY:CYCLES_L1D_PENDING
CYCLE_ACTIVITY:CYCLES_L2_PENDING	CYCLE_ACTIVITY:CYCLES_L2_PENDING
#Dispatch Resource Stalls	
RESOURCE_STALLS:ANY	RESOURCE_STALLS:ANY
RESOURCE_STALLS:RS	RESOURCE_STALLS:RS
RESOURCE_STALLS:SB	RESOURCE_STALLS:SB
RESOURCE_STALLS:ROB	RESOURCE_STALLS:ROB
#TLB	
DTLB_READ_HITS	perf::PERF_COUNT_HW_CACHE_DTLB:READ:ACCESS
DTLB_READ_MISSES	perf::PERF_COUNT_HW_CACHE_DTLB:READ:MISS
DTLB_WRITE_HITS	perf::PERF_COUNT_HW_CACHE_DTLB:WRITE:ACCESS
DTLB_WRITE_MISSES	perf::PERF_COUNT_HW_CACHE_DTLB:WRITE:MISS
DTLB_TOTAL_HITS	@(+,perf::PERF_COUNT_HW_CACHE_DTLB:READ:ACCESS,perf::PERF_COUNT_HW_CACHE_DTLB:WRITE:ACCESS)
DTLB_TOTAL_MISSES	@(+,perf::PERF_COUNT_HW_CACHE_DTLB:READ:MISS,perf::PERF_COUNT_HW_CACHE_DTLB:WRITE:MISS)
ITLB_READ_HITS	perf::PERF_COUNT_HW_CACHE_ITLB:READ:ACCESS
ITLB_READ_MISSES	perf::PERF_COUNT_HW_CACHE_ITLB:READ:MISS
#Branch Predictor	
BR_INST_TOTAL	PAPI_BR_INS
BR_INST_UNCOND	PAPI_BR_UCN
BR_INST_COND	PAPI_BR_CN
BR_INST_COND_TAKEN	PAPI_BR_TKN
BR_INST_COND_NOT_TAKEN	PAPI_BR_NTK
BR_INST_COND_MISSPREDICT	PAPI_BR_MSP
BR_INST_COND_CORRECT	PAPI_BR_PRC
#Other	
CONTEXT-SWITCHES	perf::CONTEXT-SWITCHES

Table 2: Haswell Dictionary. Cache Performance Counters.

Dictionary Keys	Associated Database Entry or Formula (Performance Counter or Combination of Counters)
	#L1 cache
L1D_READ_HITS	perf::PERF_COUNT_HW_CACHE_L1D:READ:ACCESS
L1D_READ_MISSES	perf::PERF_COUNT_HW_CACHE_L1D:READ:MISS
L1D_WRITE_HITS	perf::PERF_COUNT_HW_CACHE_L1D:WRITE:ACCESS
L1D_WRITE_MISSES	PAPI_L1_STM
L1I_READ_HITS	perf::PERF_COUNT_HW_CACHE_L1I:READ:ACCESS
L1I_READ_HITS	none
L1I_READ_MISSES	perf::PERF_COUNT_HW_CACHE_L1I:READ:MISS
	#L2 cache
L2D_READ_ACCESSES	perf::PERF_COUNT_HW_CACHE_L1D:READ:MISS
L2D_READ_MISSES	none
L2D_WRITE_ACCESSES	PAPI_L1_STM
L2D_ACCESSES_TOTAL	PAPI_L2_DCA
L2_READ_TOTAL	PAPI_L2_TCR
L2_WRITE_TOTAL	PAPI_L2_TCW
L2_READ_ACCESSES	@(+,perf::PERF_COUNT_HW_CACHE_L1D:READ:MISS,perf::PERF_COUNT_HW_CACHE_L1I:READ:MISS)
L2_READ_MISSES	PAPI_L2_LDM
L2_WRITE_ACCESSES	PAPI_L1_STM
L2_WRITE_MISSES	PAPI_L2_STM
	#L3 cache
L3_ACCESSES_TOTAL	PAPI_L3_TCA
L3_MISSES_TOTAL	PAPI_L3_TCM

perfmon2, etc.), provides the SQLite parsing scripts and the drawing scripts with a list of unique keys that are be translated to the specific architectural or simulator values we want to compare.

The next step is to run the benchmarks/applications in the simulated/real infrastructures and produce the inputs for the parsing scripts to build the SQLite databases. This is trivial for gem5, since all the necessary statistics can be produced simultaneously once the simulator is modified to count them. However, the maximum number of performance counters we can simultaneously work with in real hardware is usually very limited (around four/five for the evaluated Haswell processor). In addition, there are performance counters that cannot be measured at the same time, since they share the same physical register. There are two options to solve this issue: a) multiplexing or b) running several times with different groups of events that can be counted simultaneously. The proposed infrastructure explores the second option.

Currently, performance counter groups are created manually, using the tool

"papi_event_chooser" provided by the PAPI library [51]. This tool checks whether a set of performance counters can be measured simultaneously. Once we have the ones we are interested in divided into groups, we have several ways to measure them, either from within the application or from the command shell (e.g., perf, likwid, etc.). We decided to measure only the "region of interest" (ROI) of the selected benchmarks, that is, after initialization and before writing the outputs to disk/screen.

The ParVec benchmark suite inherited the "hooks" library from PARSEC, a dummy library which is called by the benchmarks when they start, before entering the ROI, after the ROI and before finishing. We modified these dummy functions to measure performance counter information within the ROI using PAPI. Since we do not wish to parametrize the function calls to the library, we have scripts that build one library for each group of events (16 in our case study). We then use *LD_PRELOAD* before running the benchmark to link to the different libraries and measure the counters in each group. The framework provides an example script to perform the profiling process, using CPUSET (cset) to shield the cores that will run the application from other system processes, minimizing interference in the measurements.

After the profiling phase, the framework uses a set of python scripts (one for the CPU and another one for gem5), to build a SQLite database with all the performance counter and statistics information retrieved. These scripts also compute derivative statistics from the raw data, like the Top-Down model, runtime information, trimmed mean, mean, standard deviation, minimum, maximum, etc. This information is stored into different tables of the database.

Finally, the framework contains a python drawing script that builds figures using the Pychart framework, and places them in a (basic) HTML web interface (to be improved in the future). This script takes keywords from the architecture dictionary to retrieve information from the database, and automatically adjusts margins and label information to produce some basic SVG and PDF output images. All the figures shown in this paper are produced automatically with these scripts. Examples will be provided upon release to show how to use this

drawing infrastructure, since we believe it is not the point of this paper to be a user manual.

4. Experimental Methodology

This section describes the Top-Down model, presented by Yasim *et al.* in 2014 [3], along with information about the evaluation environment: applications, hardware platform and simulation infrastructure.

Top-Down Model. This methodology is described by the authors as "a practical method to quickly identify true bottlenecks in out-of-order processors". It can be seen as a multi-level summary of the hundreds of performance events available in modern processors, to quickly and accurately identify dominant performance bottlenecks. Among the many levels of detail available in this methodology, we will focus on the first level, namely Top-Level breakdown. This level divides the status of the issued μops into four categories: frontend-bound, backend-bound, retiring or bad-speculation. Retiring accounts for μops finishing normally and leaving the pipeline, while bad-speculation represents those μops squashed due to a mispredicted branch. If μops are neither allocated resources nor squashed, then it means they are stalled. Frontend-bound represents the ratio of μops stalled in fetch/decode, while backend-bound covers ready-to-issue μops that cannot continue along the pipeline due to resource unavailability. Backend-bound μops can be categorized into CPU or memory bound, depending on the resources causing these stalls. This model has been adopted by Intel's profiling production tool "VTune". The required performance counters are already featured in in-production systems (just eight new events in the PMU on Intel's Ivy Bridge and later models).

Real Hardware. The evaluated platform is an Intel Core i7-4600U CPU, a Haswell system with two physical cores and four logical threads. This platform is labeled "Haswell" in our evaluation section. The CPU runs at 2.1GHz and has a pipeline depth of 14 to 19 stages. Caches are organized as: 32KB+32KB

of 8-way associative L1D/I, 256KB of unified 8-way associative L2 and 4MB of 16-way associative L3. In order to ensure that the simulated system and the real hardware are running the same code, we use the "chroot" command to run a virtualized copy of the software system. This virtual system is Ubuntu 16.04.1 and is stored into an image file that will be later used by gem5. Application binaries are built inside this disk image on the real system and used later by gem5 without re-compiling. In essence, both systems run on the same system image, binaries and libraries with Linux kernel version 4.9.4. In order to isolate our measurements from other system processes, benchmarks are executed using CPUSET. This tool shields certain cores and binds the program to those cores preventing other processes from running on them. PAPI version 5.5.0 is used to retrieve performance counter information from the region of interest of the applications. We evaluate the simsmall, simlarge and native input sets. Applications are run 100 times (10 times for native) for each of the 16 event groups containing the selected performance counters, and a 0.3 trimmed mean is computed to remove outliers.

gem5. Results in this paper are based on the gem5 development branch dated Nov 2018 with minor modifications. The simulator was extended to account for TLB statistics, issue breakdown cycle stalls and additional statistics required for the Top-Down model, including cycles running with L1 misses, stalled with L1 cache miss, etc. The Haswell i7-4600U has been modeled as closely as possible. Structure sizes (branch predictor, L1 caches, BTB), pipeline widths and ALU latencies are based on several sources, including Intel documentation [52], and Agner Fog's work [53]. A summary of the configuration is shown in Table 3.

Applications. Benchmarking is the standard method to conduct scientific experiments in computer science research. Benchmark suites gather together a set of benchmarks or kernels which are a representation of applications of interest. There are well-known benchmarks suites (i.e., Rodinia, PARSEC, Parboil, SHOC, Antutu, Linpack, EEMBC, ParMiBench, etc.), that try to be as "complete" as possible. However, as technology moves forward, benchmarks should

Table 3: gem5 configuration based on i7-4600U specs.

Parameter	Value
Timebuffer latencies (e.g, fetchtodecode)	1
Branch Target Buffer	1-Way, 2048-Entry
Branch Predictor	Bimode. Global:8K-Entry Choice:8K-Entry
FU Pools (x2)	1x Int.Alu only — 3x Int. FP and SIMD ALU
FetchQueueSize	32-Entry
FetchBufferSize	16B
DecodeBuffer (new structure)	56-μops
LQEntries	72
SQEntries	42
LFS/SSITSize	1024
numPhysInt/Float Regs	168
numIQEntries	60
numROBEntries	192
Fetch / Decode / Rename Width	4
Dispatch / Issue / Commit Width	8
Squash Width	8 (changed to 192 in evaluation)
L1-I cache — Latency (cycles)	32KB, 8-Way — 1 (to emulate μops cache)
L1-D cache — Latency (cycles)	32KB, 8-Way — 4 (access)
L2 cache — Latency (cycles)	256KB, 8-Way — 12 cycles
L3 cache — Latency (cycles)	4MB, 16-Way — 36 cycles

be extended to cover new hardware features, or else researchers may end up over or underestimating the impact of their contributions.

In addition, none of the previously mentioned suites offers support for a critical feature that is of rising importance in modern HPC (high performance computing) systems, SIMD/Vector features. Cebrian *et al.* [54] discuss the effects of SIMD implementations on microprocessor resource utilization and energy efficiency when compared with regular scalar implementations. In their work the authors show the benefits of including SIMD-enabled benchmarks when evaluating new architectural features. Intel already supports 512-bit vector registers, and Arm released SVE (Scalable Vector Extension) [55], that is able to scale up to 2048-bit vector registers. We believe that SIMD features cannot be ignored, and thus this paper uses the benchmarks in ParVec [56].

The i7-4600U CPU offers support for SSE and AVX2 instructions, but gem5 only supports SSE instructions, so the evaluation will only cover scalar and SSE

codes. All the applications and libraries are built using GCC version 6.2, with the -O2 flag to prevent automatic vectorization, focusing on manually vectorized code sections. In addition, at the end of the results section we provide a Top-Level breakdown overview of 12 of the Splash 3.0 [57] benchmarks for the fixed simulator and the Haswell system. We consider the ParVec benchmarks as a "training" set for the simulator to find sources of error. The Splash 3.0 is later employed as a "test" or "validation" set to validate our discoveries and fixes. However, to perform an in-depth validation study of any simulation framework, microbenchmark-based validation could be a better alternative (e.g., [58]). We leave this alternative for future work.

5. Case Study: gem5-x86

This section shows a case study on how to detect sources of error in simulation infrastructures using outputs provided by the validation framework. Additional information provided by the framework, such as TLB statistics, caches, execution stall breakdown, etc., is not shown due to space limitations. Figures show results for five of the ParVec benchmarks: blackscholes (BS), swaptions (SW), fluidanimate (FL), streamcluster (ST) and canneal (CN). Stacked bars represent the three infrastructures being compared: gem5-official (GO), Haswell (HW) and gem5-fixes (GF).

5.1. Top-Down Overview

As discussed in previous sections, the Top-Level breakdown (Figure 2) is generated by the framework, providing developers with a summary of the dominant system performance bottlenecks. Users must then analyze the evaluated benchmarks on the different platforms, looking for sources of error in the frontend, backend or mis-speculation. It can be depicted that blackscholes-scalar, streamcluster-scalar, swaptions-SSE and canneal-SSE have similar bottlenecks in the official gem5 as they do in the real hardware. However, there are major discrepancies in the rest of the benchmarks. Mis-speculated paths on fluidanimate show clear differences. Backend/retiring stages on the rest of the

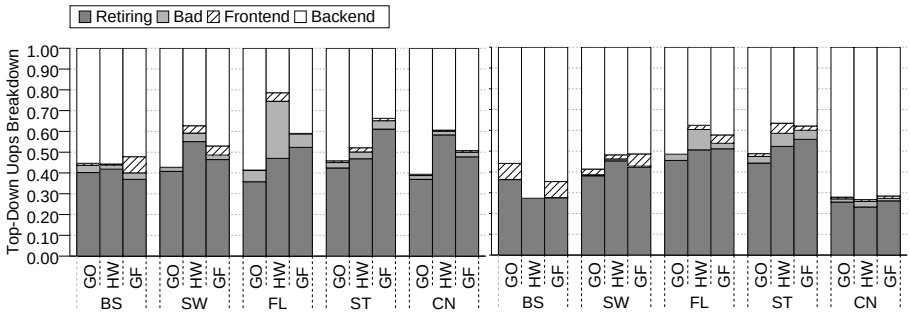

Figure 2: Top-Down model. Top-Level overview for simsmall for five benchmarks and gem5-official (GO), Haswell (HW) and gem5-fixes (GF). Left scalar, right SSE.

benchmarks also show significant variability between the simulated and real hardware. In addition, frontend bottlenecks seemed lower in the simulator than in real hardware for most cases. This translates into different hardware resource requirements that may affect conclusions obtained in research proposals.

5.2. Pipeline Overview

The Top-Level breakdown shows discrepancies in both frontend, backend and mis-speculated paths. Lower level information is required to find out the sources of error and determine if these errors can be corrected. However, before looking for errors we need to make sure that both the real hardware and the simulation infrastructure are running the exact same code. As discussed in the methodology section (Section 4), both systems run on the exact same system image, sharing binaries and libraries. However, the OS may be scheduling some background process in the simulator that may alter the output statistics. OS interference can be an issue when running gem5 in full system simulation. System checkpoints are taken during the boot process, so that the system can interact with the simulator using a scripting infrastructure. In a modern OS, the boot process is performed in parallel ("systemd" handles the boot in parallel in our case). We took precautions when integrating the system services that handle gem5 scripts in "systemd", so that they run in "oneshot" mode. In that mode, the service blocks on a start operation until the first process exits (i.e., the gem5

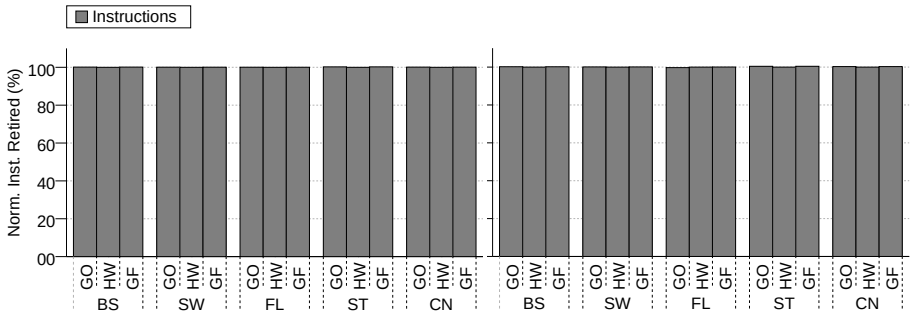

Figure 3: Normalized decoded/retired instructions for simsmall with five benchmarks using gem5-official (GO), gem5-fixes (GF) normalized to Haswell (HW). Left scalar, right SSE.

script that runs the desired application). However, the simulation could be corrupted due to a bug and actually run more/less instructions producing a different output. The proposed validation framework can help us determine if there is a significant OS interference or data corruption. In other words, we can compare the total number of decoded/retired instructions on both systems, and see if there is an OS overhead/corruption on any of them. Indeed, by asking the framework about the "MACRO INST RETIRED" dictionary key to both Haswell and gem5 databases, it generates Figure 3. This figure shows the normalized decoded/retired macro instructions on both Haswell and gem5 for the simsmall input. The amount of instructions is virtually the same in all cases, so OS interference is minimal in the tests performed. Similar results were obtained for the simlarge input set.

Mis-speculated paths. gem5 implements several branch predictor models, including g-share, bimode, tournament, etc. However, specific details of the branch predictor on Haswell are unknown. The evaluation performed in this paper uses a bimodal branch predictor with similar sizes to those of the Haswell system. Bimode has a slightly higher hit ratio than tournament in the evaluated benchmarks, but the difference is minimal. Different hit ratios of the branch predictor would lead to differences in the total number of instructions executed from wrong execution paths. The proposed validation framework can provide useful infor-

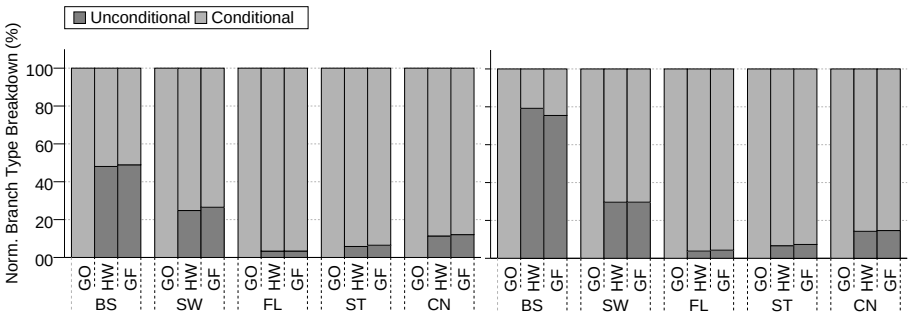

Figure 4: Normalized branch type breakdown for simsmall with five benchmarks using gem5-official (GO), gem5-fixes (GF) normalized to Haswell (HW). Left scalar, right SSE.

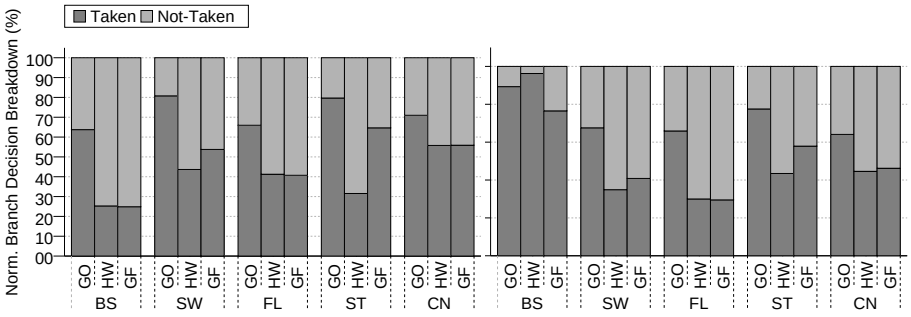

Figure 5: Normalized branch decision breakdown for simsmall with five benchmarks using gem5-official (GO), gem5-fixes (GF) normalized to Haswell (HW). Left scalar, right SSE.

mation about the branch predictor behavior, including the total conditional and unconditional branch instructions (Figure 4), taken/not-taken branches (Figure 5) and hit/miss ratios (Figure 6).

The information provided by these figures allow us to detect a bug in the gem5 infrastructure. More specifically, branch instructions are not labeled as conditional-unconditional in the x86 decoder, while they are properly labeled for other decoders (e.g., the Arm decoder). This causes unconditional branches to wait until commit to provide the correct address information, and update the BTB/branch prediction accordingly. However, for unconditional branches, the correct address information is available at decode stage. When branches are properly flagged, the BTB is updated at decode stage, and instructions

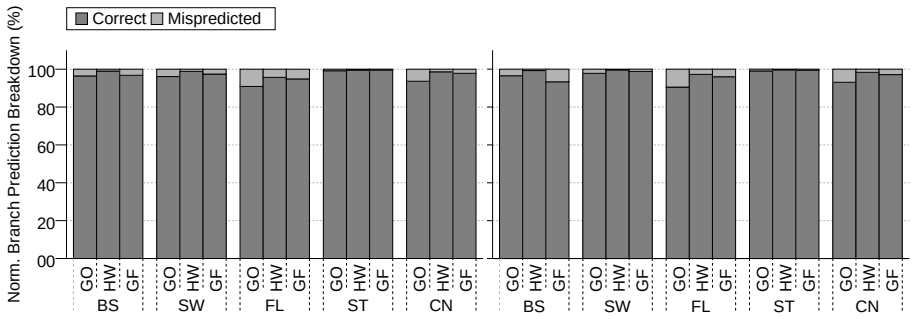

Figure 6: Normalized branch decision breakdown for simsmall with five benchmarks using gem5-official (GO), gem5-fixes (GF) normalized to Haswell (HW). Left scalar, right SSE.

from a potential "mis-predicted" branch can be detected much earlier. This issue was especially problematic in applications with many system calls. In this case, the BTB will point to the last system call and the last return address, and will wait until committing to correct the prediction, modeling an incorrect behavior. This will not alter the output of the application, since instructions will not commit, but may trigger internal simulator asserts on different pipeline stages that should otherwise never happen. However, while this fix helps to better model the correct branch behavior of the architecture, it does not fix the issue with mis-speculated paths on the Top-Level breakdown. A possible explanation for this second source of error will be discussed in the next section.

Another interesting result is the behavior of the streamcluster kernel. While both Haswell and gem5-fixes claim to have almost perfect accurate predictions, Haswell decides to take conditional branches around 35% of the time, and gem5-fixes around 67%. This information seems contradictory, since both predictors cannot have the same accuracy making different predictions. Our best guess is that taken/not-taken information in gem5 refers to the BTB, rather than the branch predictor. The BTB has much lower accuracy when using the simsmall input, since it does not have time to warm up.

Frontend. The next step is to determine why the frontend fraction in the Top-Down for real hardware and gem5 do not match. This can be achieved by

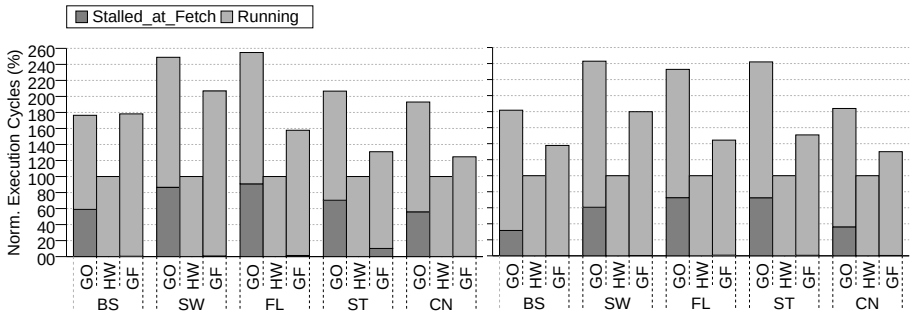

Figure 7: Normalized fetch stalls for simsmall with five benchmarks using gem5-official (GO), gem5-fixes (GF) normalized to Haswell (HW). Left scalar, right SSE.

comparing the behaviour of the fetch and decode stages between gem5 and Haswell in our framework. Figure 7 shows both normalized running cycles and cycles stalled at fetch stage, as provided by our framework, for the different benchmarks to the Haswell platform. The official gem5 has significantly higher (several orders of magnitude) fetch stalls than the Haswell platform. This not only translates into slower simulation times and unrealistic application performance, but also into misleading resource requirements at later pipeline stages. After a detailed analysis of the simulator, we discovered a bug in the packet buffer that receives packets from the memory module. There are several memory modules available in gem5, including a simple memory module and a detailed memory one (Ruby). Ruby deals with cache coherence and cache/bus latencies, and specifies the exact cycle when data will be available to the processor in each data packet. However, the packet buffer adds an additional processing cycle, assuming packets come from the simple memory module (probably to account for the data transfer latency). This causes additional blocking cycles at fetch stage when using Ruby to model the memory subsystem, since it already considers network latencies and does not need extra cycles. In fact, authors make a comment in the code saying "@todo Revisit the +1", which is actually wrong when using Ruby. To solve this issue, packets that go through the Ruby module are marked, and those packets skip the +1. After fixing this error, fetch

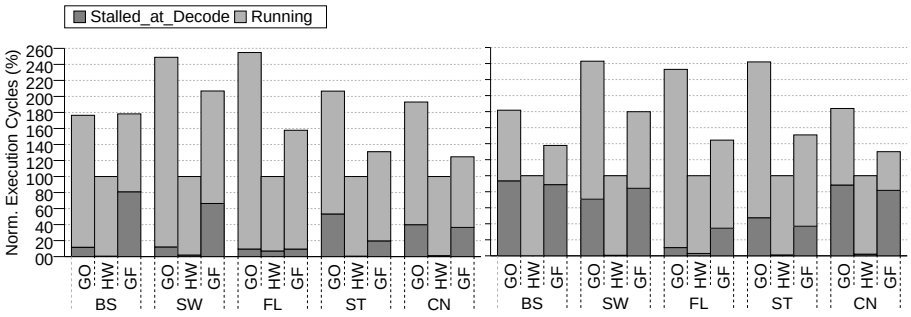

Figure 8: Normalized decode stalls for simsmall with five benchmarks using gem5-official (GO), gem5-fixes (GF) normalized to Haswell (HW). Left scalar, right SSE.

stalls are considerably reduced for all benchmarks. In addition, the simulator shows around 10x more L1I cache misses than the Haswell system, most likely responsible from the remaining fetch stalls in gem5 after the fix. Cache miss information is provided by the infrastructure but not shown here due to space limitations.

Fixing this issue moved the problem to decode stage (Figure 8, as provided by our framework). gem5 has been designed to simulate different pipeline depths by adding latency between the different pipeline stages. That is, if the fetch stage is divided into three segments in the real hardware, gem5 can be configured to add a 3-cycle latency between fetch and decode. Our configuration file tried to emulate a Silvermont 6-stage frontend, since we could not find detailed information about the Haswell frontend ([59]). We added a 3-cycle latency between fetch-decode and decode-dispatch. However, increasing these latencies can lead to a significant increment in decode stalls. Indeed, while the simulator will block decode stage completely if it cannot perform the decoding, a segmented architecture may be able to handle early stages of decoding, but never increasing this specific performance counter. In addition, while this way of modeling pipeline depth is supposed to let more instructions from a mis-speculated branch into the pipeline, it seems to actually have the opposite effect. Since early stages of the pipeline are blocked, the amount of instructions

from mis-predicted paths that reach the retire stage is lower than in the real hardware. This fact, combined with the shorter pipeline length, can explain the lower number of "bad speculation" cycles from fluidanimate, shown in the Top-Level breakdown overview (Section 5.1). This happens despite the similar branch hit ratios shown in Figure 6. To check this hypothesis we computed the ratio of μops from different stages that reach retire. For Haswell, 27% of fluidanimate μops that issue never make it to retire, while this number is much lower in gem5, both with 3+3 cycle (11.4%) and 1+1 cycle (11.5%). This behavior has also been reported in the "gem5-users" mail list, where developers usually recommend to set fetch-to-decode and decode-to-rename latencies to one cycle to prevent unexpected side-effects.

After applying this change to the configuration files, some benchmarks still showed a significant number of decode stalls (e.g., blackscholes more than 60%). However, these stalls are not reflected in the Top-Down model. Indeed, the Top-Down model only accounts as frontend stalls those fetch/decode stalls that happen when the backend is ready. This leads us to conclude that decode stalls are caused by later pipeline stages.

Backend. This section analyzes different pipeline stages from the processor's backend, namely: dispatch (rename in gem5), issue, execution and retire (write-back in gem5) as provided by our validation framework. Issue, execution and writeback are handled simultaneously by gem5 in a single unit, while Haswell provides separate counters for each stage.

After applying the fixes discussed in the frontend section, dispatch (Figure 9) showed around half stall cycles as decode in gem5. A breakdown of sources of stall cycles at dispatch and execute stages is provided by our framework, but not shown here for the sake of clarity. This breakdown shows that most stalls are caused by a lack of reservation stations (instruction queue in gem5). Our next improvement to the simulator was to add a decode buffer between decode and rename, to prevent back and forth blocking between rename and decode. This structure is available in Haswell as well. The decode buffer however had

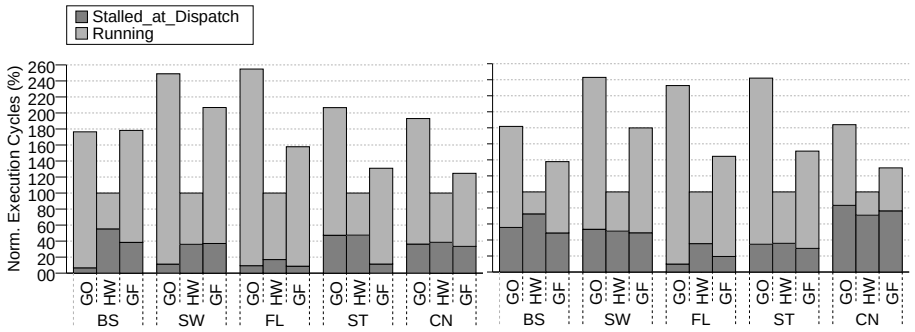

Figure 9: Normalized dispatch stalls for simsmall with five benchmarks using gem5-official (GO), gem5-fixes (GF) normalized to Haswell (HW). Left scalar, right SSE.

little to none effect on overall decode stalls, but it halved in some cases the frontend stalls in the Top-Down model.

Finally, after carefully checking several simulator traces, we found the main bottleneck between gem5's frontend and backend. Whenever the execution stage detects a branch misprediction, it sends a signal to the commit stage to squash the contents of the pipeline, ROB and other structures. One cycle after, commit sends a signal to all stages to perform the squash. At this point, commit begins to squash the ROB, in groups of "squash width" instructions per cycle (set by default to the same width as commit, i.e., 8). If commit squash takes more than one cycle, all other pipeline stages (except fetch) will be blocked for as long as commit is squashing. This can take up to 192/8 cycles in our case (N cycles). The issue-execution-writeback (IEW) stage is notified to unblock one cycle after commit finishes squashing. Similarly, rename is notified to unblock one cycle after IEW finishes unblocking (X cycles). Finally, decode is notified to unblock when rename is finished unblocking (Y cycles), and the processes will take another Z cycles.

This process creates a backwards-bubble in the pipeline of 1+N+X+Y+Z cycles, that adds to the forward bubble of filling the pipeline. The most common case is that X, Y and Z equal to one, since widths of all backend stages are usually the same. The backward bubble will be thus of 4 cycles + N (for

N>1). This design decision is especially harmful for benchmarks with high IPC, since they make more use of the available ROB entries, which need to be slowly squashed on a miss prediction. We are uncertain about the equivalent architectural parameter in Haswell to "squash width", but we do not believe that this process will be blocking the pipeline. Since the pipeline depth of Haswell is 14 to 19 stages, this width can be adjusted to ensure that the ROB will be ready by the time new instructions reach the backend. This will prevent the real system from creating this backwards bubble, since the frontend will start producing instructions right away. We believe that this "blocking" design and the associated backwards bubbles are a key limiting factor in the simulation of certain benchmarks, like blackscholes. To minimize what we believe to be a source of error, we set our "squash width" to the number of ROB entries, so that N equals one cycle and the cascade block never happens.

Another interesting finding derived from the validation framework has to do with the translation of macro to micro instructions. Figure 10 shows the amount of μops generated in the decode stage. There is an increment that ranges between 20 and 80% in the amount of μops issued by gem5 when running scalar codes. Moreover, this increment reaches 240% when running SSE codes. This μop increment is slightly lower for gem5-fixes in scalar codes. As it will be discussed in Section 5.4, scalar code produced by x86_64 compilers is in reality SSE code limited to the lowest lane. This code is handled slightly better by our SSE decoder than it is in the official gem5. All those extra μops need to traverse the pipeline and use resources. This means that, even if we model the resources to match a specific architecture (reservation stations, ROB entries, stage widths, etc.), the resources consumed by the simulator can vary considerably depending on the application.

For SSE codes there is another great concern. While Intel specifies that SSE and AVX instructions are translated into a single μops in most cases, gem5 translates SSE instructions into as many μops as an equivalent 64-bit scalar implementation. The reason to do this is the lack of vector registers in the simulator (128-bit registers in SSE). For example, for an SSE load of single-precision

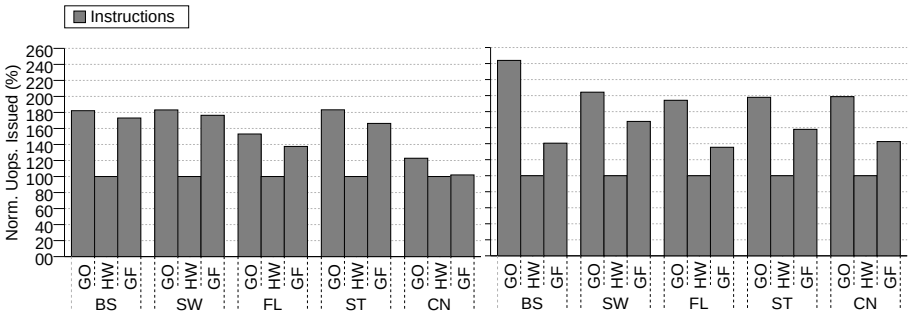

Figure 10: Normalized μops issued for simsmall with five benchmarks using gem5-official (GO), gem5-fixes (GF) normalized to Haswell (HW). Left scalar, right SSE.

floating point values, gem5 performs two 64-bit load operations. This explains the huge increment on μops count. We fixed this issue by introducing proper vector registers and vector instructions that translate into a single μinstruction to simulate SSE. Still, gem5-fixes maintains the overhead from μops derived from some scalar and SSE macro instructions. More specifically, arithmetic or logical operations that use the "memory addressing mode", for example "add r1,r2,[rsp + 10]", will translate into two gem5 μops, "load r3, [rsp + 10]; add r1,r2,r3" (rsp: register stack pointer). We still need to design a synthetic test that uses both memory and register address modes. This will allow us to study the differences in real hardware and how they can translate into the simulator. Performing an "up-to-date" translation of all x86 instructions for the Haswell case seems out of the scope of this paper. Furthermore, increasing the amount of resources to cope with this increment of instructions does not seem reasonable, since it will affect applications that do not have a huge μop deviation from the real hardware (e.g., canneal).

For issue, execute and retire stages, the general shape of gem5-fixes is similar for most benchmarks and stages. Although gem5 models all three stages as one, variability between stages exists since Haswell has different instruction count per stage. This behavior has been discussed in the Intel forums by John McCalpin, and initially reported by Agner Fog when discussing "micro-op fusion" in [53].

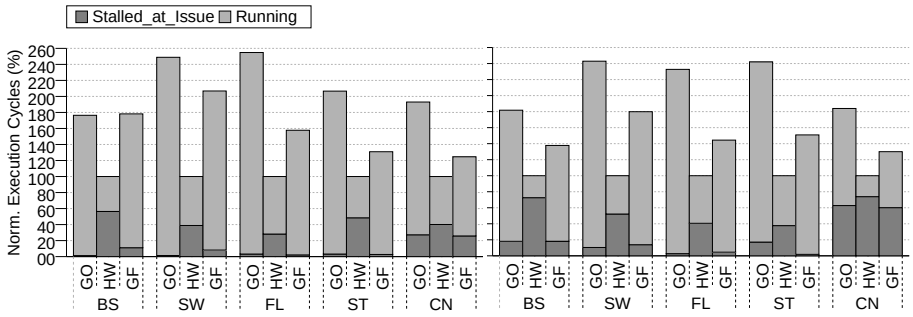

Figure 11: Normalized issue stalls for simsmall with five benchmarks using gem5-official (GO), gem5-fixes (GF) normalized to Haswell (HW). Left scalar, right SSE.

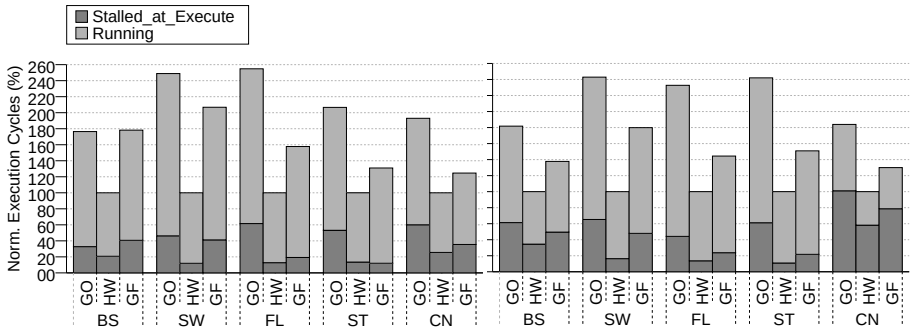

Figure 12: Normalized execute stalls for simsmall with five benchmarks using gem5-official (GO), gem5-fixes (GF) normalized to Haswell (HW). Left scalar, right SSE.

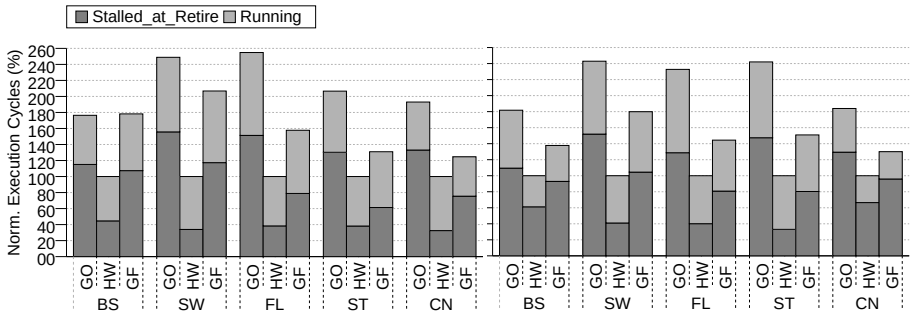

Figure 13: Normalized retire stalls for simsmall with five benchmarks using gem5 official (GO) gem5 fixes (GF) normalized to Haswell (HW). Left scalar, right SSE.

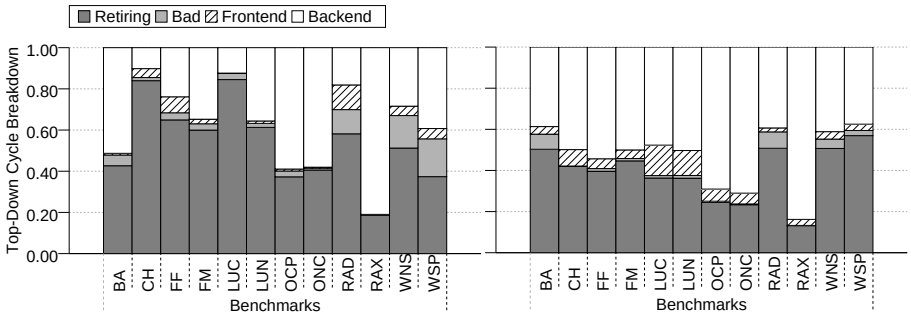

Figure 14: Top-Down model. Top-Level overview for the Splash 3 benchmark suite using the default input. Left Haswell. Right gem5-fixes

In addition, they also mention that counters like "μops dispatched" (to the execution ports), can be much larger than either μops issued or μops retired due to instruction retries. This supports our initial guess that the Haswell pipeline implementation is not blocking previous stages, but discarding/retrying when a specific stage is not ready.

5.3. Validating with Additional Applications

The evaluation and improvements on gem5 were performed using five benchmarks from the ParVec suite. We have considered these ten benchmarks (scalar and SSE) as a "training set" for the simulator, but it does not guarantee that the discovered sources of error translate to other benchmarks. The "fixed" simulator improves both the accuracy of the results and the simulation speed. For scalar codes on ParVec, simulation speed is improved by around 35% on average, since we are significantly reducing the amount of pipeline stalls. Therefore, running the application faster reduces simulation time. However, there may be cases where fixing the simulator actually slows down simulation speed. Improvements for SSE codes are even better (43%), especially since we implement proper vector registers, reducing the amount of instructions that the simulator has to execute. This section shows Top-Level of the Top-Down model with Splash 3 applications, that can be considered our "test or validation set" (Figure 14).

As it happened with the ParVec benchmarks, the main differences in the

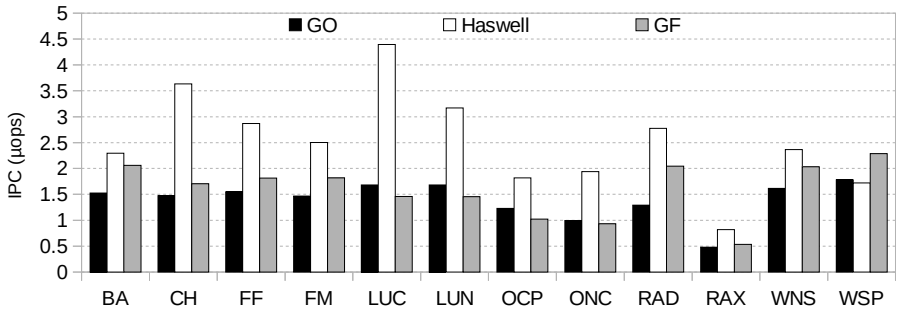

Figure 15: μops per cycle for the Splash 3 benchmark suite for gem5-original (GO), Haswell and gem5-fixes (GF).

Top-Level breakdown between the real hardware and gem5 come from applications with many mis-speculation and those with high IPC (Figure 15). Further improvements to the simulator are needed in order to capture the behavior of modern processors.

Finally, Figures 15 and 16 show the μops per cycle for all the evaluated benchmarks and ISAs (scalar and SSE) on both Haswell and gem5. Most computer architecture papers rely on the IPC metric to "validate" their simulation infrastructure. Overall, our fixed gem5 achieves higher IPC than the original gem5. However, there are cases where the IPC (e.g., blackscholes from ParVec) is reduced when fixes are applied. This can be counter-intuitive, since the error detected by the framework makes gem5 slower, not faster. This is actually due to the wrong allocation of ALU resources that will be discussed in the next section. However, as we have seen, IPC, or even the Top-Level overview alone are not good enough to detect sources of error. Positive and negative sources of error can neglect each other, and it may seem like the simulator is close to the real hardware, while in fact is behaving erratically.

5.4. Instruction Type Breakdown

Another issue we found with the simulator has to do with the instruction classification. gem5 ALUs are implemented as pools (namely FU pools) that can handle specific instruction types. The instruction types handled by different

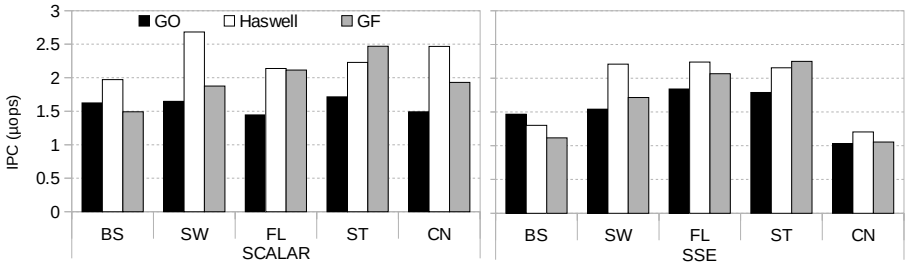

Figure 16: μops per cycle for five benchmarks of the ParVec benchmark suite using the simsmall input for gem5-original (GO), Haswell and gem5-fixes (GF).

pools are not exclusive. Developers can have, for example, two pools, one for integer additions only, and another one for any type of floating point and integer operations. This is an excellent way to emulate the behavior of Intel's ALU ports. In this way, users can set the latency for each instruction type in each FU pool.

Besides, and in order to build x86 scalar code, both GCC and ICC compilers use SSE instructions that operate on the lower register lane (e.g., MOVSS, ADDSS, etc.). We believe this is part of the x86_64 standard, and the only way we could find to avoid this is to set the compiler flag "-m32". However, using this flag produced an extremely low-quality 386 code.

SSE codes are passed through the SSE decoder in gem5, but none of the SSE instructions has the instruction type set in the official gem5. This makes all the generated SSE μops to fall back to default types (IntAlu, FloatAdd, MemRead and MemWrite). These default instruction types may have a different latency than the one intended to simulate (e.g., a division, or a square root). This issue is not present on the Arm decoder, where instruction types are properly set. We have fixed this issue in our SSE decoder, and now instruction types are set properly, using their specific FU pool.

Evidently, properly setting multiplication, division and square root instruction types made gem5-fixes slower than the official gem5. The overall slowdown will depend on high-latency instructions present in the critical path of execution. We have also included specific types for double precision division and

square root operations, since latencies are different from those of single-precision floating point. This kind of information is not available through performance counters, but can be retrieved using a binary emulation software such as Intel SDE (Software Development Emulator). We believe it would be interesting to include this information in future revisions of the validation framework.

5.5. Extensibility of the Framework

Extending the framework can be a necessary step to support new architectures or to include additional performance counter information required to validate our simulation environment.

In order to include a new architecture, users must create a dictionary file tailored for that specific architecture. This requires a certain level of expertise. The user must locate the appropriate performance counter (if available) for the target architecture. A detailed definition of what is expected from each performance counter that is being mapped can be found in the base dictionary file. The user must find the corresponding performance counter in the architecture documentation. Alternatively, a list of native performance counters can be obtained from PAPI running the "papi_native_avail" command.

PAPI eases this work by hiding some performance counter names. Indeed, some performance counters are mapped by PAPI developers to specific names for each architecture, so one can just point at the PAPI name (e.g., PAPI_BR_INS for total branches) and that would map to the specific performance counter in each architecture.

On the other hand, if a user wants to include additional performance counters to the framework, this can be easily done by defining new wrappers in the dictionary files for the architectures to compare. However, in order to add more complicated features (e.g., complex formulas or new tables in the results database) the user is required to modify the python scripts that build the database and create the figures, using the ones provided as an example. There may be cases where the architecture does not have a specific performance counter required by the framework. A simple "none" in the dictionary file will

suffice to skip it.

Moreover, our validation framework can be extended to work with any simulation environment. We use gem5 as a case-study, but the use of this simulator is not mandatory. The only requirement is an output database from the simulation environment with pairs of values, event-mnemonic and event-value, so that they can be accessed by the framework based on the dictionary definition.

5.6. A Different use for the Framework: Input Dataset Analysis

Long simulation times are usually a limiting factor in cycle-accurate infrastructures. It is common that researchers use small input datasets to reduce simulation time. However, reducing the input sizes may have undesired side effects, especially if simulated resources are not scaled accordingly. For example, a small input set may fit in the L2 or L3 cache, and have better performance than a realistic input set that does not fit in any level of the cache hierarchy. Ideally, researchers should be aware of the impact of using small input sets on system resources. This knowledge is crucial to achieve conclusions that can be extrapolated to realistic input sets.

The proposed validation framework can be used to check the effects of different input sizes on processor resource utilization. We have tested three input sets from ParVec (simsmall, simlarge and native) on both scalar and SSE codes running with a single thread. Figure 17 shows that blackscholes and swaptions behave similarly for all three input sets. However, fluidanimate, streamcluster and canneal show different performance bottlenecks depending on the input size. Simlarge is a good compromise between runtime and resource utilization, except for fluidanimate, where simsmall actually behaves closer to the native input than simlarge.

Another interesting finding is the variation of resource requirements between the scalar and SSE codes. This fact has already been reported by Cebrian *et al.* [54]. Vectorized applications should be considered (in addition to scalar codes) in hardware research proposals. Otherwise, researchers may end up under/over estimating the impact their proposal may have in this type of applica-

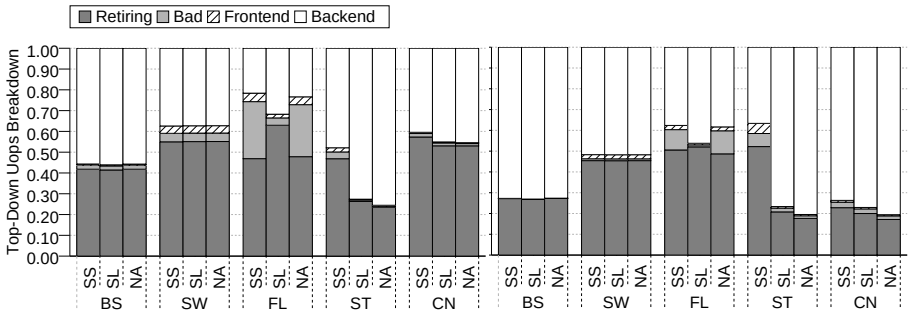

Figure 17: Top-Down model. Top-Level overview for five benchmarks and simsmall (SS), simlarge (SL) and native (NA) running on Haswell. Left scalar, right SSE.

tions.

6. Conclusions and Future Work

Verification, validation and calibration of simulation infrastructures is key to ensure the quality of computer architecture proposals. Engineers long for the simulated hardware to behave like the real architecture, or at least to have the same bottlenecks. There are many sources of error in simulation infrastructures, including: a) Modeling errors: erroneous description of the desired functionality. b) Specification errors: the developer is unaware of the internal functionality being modeled. c) Abstraction errors: the developer fails to implement certain details of the system being modeled. d) Input dataset errors: reducing input size may alter the resources required by the modeled core e) Lack of features: important features available in modern systems which are not modeled in the simulator. However, finding and solving these sources of error is a complex and costly process that not all research institutions can afford.

IPC or execution time values are not usually useful units to allow rigorous independent replication of results in simulation methodologies. In this paper we introduce a semi-automatic framework to ease the validation process [4]. The framework extracts real-hardware performance counter information and compares it against simulator statistics. We provide two levels of abstraction: a) a

high level definition of the processor behavior (Top-Down model) and b) detailed per-structure and per-pipeline-stage usage breakdown to pinpoint simulator issues. The same methodology can be used to analyze the effects of reducing the input size of a specific application on the processor behavior. To show the usefulness of the framework, we validate the gem5-x86 simulation environment.

The framework allows us to quickly discover simulator issues for baseline configurations. More specifically, issues with Ruby cache latencies, latencies between stages, μop translation, branch labeling and ROB stalls that create simulation stalls. To the best of our knowledge, none of the identified issues using the validation framework has been solved by Jan. 2019. The "fixed" simulator improves both the accuracy of the results and the simulation speed, mainly due to the reduction on pipeline stalls on the applications being simulated. However, there can be cases where fixing the simulator actually slows down simulation speed. There are still issues that need to be solved in order to improve the accuracy of the simulator. As part of our future work we would like to improve the validation framework providing an instruction type breakdown. We would also like to add support for Arm platforms, and modify the simulator to better match the simulated platform.

7. Acknowledgements

This work has been partially supported by the Spanish Government (Severo Ochoa grants SEV2015-0493, SEV-2011-00067), the Spanish Ministry of Science and Innovation (contract TIN2015-65316-P), Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272), the RoMoL ERC Advanced Grant (GA 321253), the European HiPEAC Network of Excellence and the Mont-Blanc project (EU-FP7-610402 and EU-H2020-779877). A. Barredo has been supported by the Spanish Government under Formación del Personal Investigador fellowship number BES-2017-080635. M. Moreto and M. Casas have been partially supported by the Spanish Ministry of Economy, Industry and Competitiveness under Ramon y Cajal fellowship numbers RYC-2016-21104 and

RYC-2017-23269.

References

[1] J. J. Yi, L. Eeckhout, D. J. Lilja, B. Calder, L. K. John, J. E. Smith, The future of simulation: A field of dreams?, Computer 39 (11) (2006) 22–29. doi:10.1109/MC.2006.404.

[2] B. Black, J. P. Shen, Calibration of microprocessor performance models, Computer 31 (5) (1998) 59–65. doi:10.1109/2.675637.

[3] A. Yasin, A Top-Down method for performance analysis and counters architecture, ISPASS 2014 - IEEE International Symposium on Performance Analysis of Systems and Software (2014) 35–44doi:10.1109/ISPASS.2014.6844459.

[4] J. M. Cebrian, et al., Semi-automatic Validation of Cycle-Accurate Simulation Infrastructures. (2020).
URL https://sites.google.com/view/validationframework/

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

[6] 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 Computer Architecture News 33 (2005) 2005.

[7] M. T. Yourst, PTLsim: A cycle accurate full system x86-64 microarchitectural simulator, ISPASS 2007: IEEE International Symposium on Performance Analysis of Systems and Software (2007) 23–34doi:10.1109/ISPASS.2007.363733.

[8] A. Patel, F. Afram, S. Chen, K. Ghose, MARSS: A Full System Simulator for Multicore x86 CPUs, Dac'11 (2011) 1050–1055doi:10.1145/2024724.2024954.

[9] E. Larson, S. Chatterjee, T. Austin, MASE: A novel infrastructure for detailed microarchitectural modeling, 2001 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2001 (2001) 1–9doi:10.1109/ISPASS.2001.990668.

[10] D. Burger, T. M. Austin, The SimpleScalar tool set, version 2.0, ACM SIGARCH Computer Architecture News 25 (2) (1997) 13–25. doi:10.1145/268806.268810.

[11] J. Emer, P. Ahuja, E. Borch, A. Klauser, S. Manne, S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, T. Juan, Asim: a performance model framework, Computer 35 (2) (2002) 68–76. doi:10.1109/2.982918.

[12] C. J. Hughes, V. S. Pai, P. Ranganathan, S. V. Adve, Rsim: Simulating shared-memory multiprocessors with ILP processors, Computer 35 (2). doi:10.1109/2.982915.

[13] S. haeng Kang, D. Yoo, S. Ha, TQSIM: A fast cycle-approximate processor simulator based on QEMU, Journal of Systems Architecture 67 (2015) 33–47. doi:10.1016/j.sysarc.2016.04.012.

[14] I. S. Limited, OVP Processor Modeling Guide, no. December, 2013.

[15] N. Romdan, ARM FastModels–Virtual Platforms for Embedded Software Development, Information Quarterly Magazine 7 (4) (2008) 33–36.

[16] T. M. Conte, M. A. Hirsch, K. N. Menezes, Reducing state loss for effective trace sampling of superscalar processors, Computer Design: VLSI in Computers and Processors, 1996. ICCD '96. Proceedings., 1996 IEEE International Conference on (1996) 468–477doi:10.1109/ICCD.1996.563595.

[17] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, J. C. Hoe, SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling, 30th Annual International Symposium on Computer Architecture, 2003. Proceedings. (2003) 84–95doi:10.1109/ISCA.2003.1206991.

[18] T. Sherwood, E. Perelman, G. Hamerly, B. Calder, Automatically characterizing large scale program behavior, ACM SIGOPS Operating Systems Review 36 (5) (2002) 45. doi:10.1145/635508.605403.

[19] T. E. Carlson, W. Heirman, K. Van Craeynest, L. Eeckhout, Barrierpoint: Sampled simulation of multi-threaded applications, in: 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2014, pp. 2–12.

[20] T. Grass, A. Rico, M. Casas, M. Moreto, E. Ayguadé, Taskpoint: Sampled simulation of task-based programs, in: 2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2016, pp. 296–306.

[21] J. J. Yi, S. V. Kodakara, R. Sendag, D. J. Lilja, D. M. Hawkins, Characterizing and comparing prevailing simulation techniques, High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on (2005) 266–277doi:10.1109/HPCA.2005.8.

[22] S. Nussbaum, J. Smith, Modeling superscalar processors via statistical simulation, Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques (2001) 15–24doi:10.1109/PACT.2001.953284.

[23] L. Eeckhout, S. Nussbaum, J. E. Smith, K. De Bosschere, Statistical Simulation: Adding Efficiency to the Computer Designer's Toolbox, IEEE Micro 23 (5) (2003) 26–38. doi:10.1109/MM.2003.1240210.

[24] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, H. Angepat, FPGA-accelerated simulation technologies

(FAST): Fast, full-system, cycle-accurate simulators, Proceedings of the Annual International Symposium on Microarchitecture, MICRO (2007) 249–261doi:10.1109/MICRO.2007.36.

[25] J. Wawrzynek, D. Patterson, M. Oskin, S. L. Lu, C. Kozyrakis, J. C. Hoe, D. Chiou, K. Asanović, RAMP: Research accelerator for multiple processors, IEEE Micro 27 (2) (2007) 46–57. doi:10.1109/MM.2007.39.

[26] M. Pellauer, M. Adler, M. Kinsy, A. Parashar, J. Emer, HAsim: FPGA-Based High-Detail Multicore Simulation Using Time-Division Multiplexing, Hpca'11 (2011) 406–417.

[27] S. Kraemer, L. Gao, J. Weinstock, R. Leupers, G. Ascheid, H. Meyr, HySim: a fast simulation framework for embedded software development, Proceedings of the 5th IEEE/ACM international conference on Hardware/-software codesign and system synthesis - CODES+ISSS'07 (2007) 75–80doi:http://doi.acm.org/10.1145/1289816.1289837.

[28] L. G. Murillo, J. Eusse, J. Jovic, S. Yakoushkin, R. Leupers, G. Ascheid, Synchronization for hybrid MPSoC full-system simulation, Proceedings of the 49th Annual Design Automation Conference on - DAC '12 (2012) 121doi:10.1145/2228360.2228383.

[29] N. Binkert, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. M. D. Hill, D. A. D. A. Wood, B. Beckmann, G. Black, S. K. S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, A. Basil, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. M. D. Hill, D. A. D. A. Wood, The gem5 Simulator, Computer Architecture News 39 (2) (2011) 1. doi:10.1145/2024716.2024718.

[30] R. Plyaskin, A. Herkersdorf, Context-aware compiled simulation of out-of-order processor behavior based on atomic traces, 2011 IEEE/IFIP 19th International Conference on VLSI and System-on-Chip, VLSI-SoC 2011 (2011) 386–391doi:10.1109/VLSISoC.2011.6081615.

[31] S. Stattelmann, S. Ottlik, A. Viehl, O. Bringmann, W. Rosenstiel, Combining instruction set simulation and WCET analysis for embedded software performance estimation, 7th IEEE International Symposium on Industrial Embedded Systems (SIES) (2012) 295–298doi:10.1109/SIES.2012.6356600.

[32] S. Ottlik, S. Stattelmann, A. Viehl, W. Rosenstiel, O. Bringmann, Context-sensitive Timing Simulation of Binary Embedded Software, Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES) (2014) 14:1—-14:10doi:10.1145/2656106.2656117.

[33] C. H, S. P, OTAWA, a framework for experimenting WCET computations, 3rd European Congress on Embedded Real-Time (January) (2006) 1–8.

[34] C. Ferdinand, R. Heckmann, aiT: Worst case execution time prediction by static program analysis, Building the Information Society (2004) 377–383.

[35] A. Jaleel, R. S. Cohn, C.-K. Luk, B. Jacob, CMP$im: A Pin-Based On-The-Fly Multi-Core Cache Simulator, MoBs'08 (2008) 28–36.

[36] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, A. Agarwal, Graphite: A distributed parallel simulator for multicores, Hpca'10 (January) (2010) 1–12. doi:10.1109/HPCA.2010.5416635.

[37] T. E. Carlson, W. Heirmant, L. Eeckhout, Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation, 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC) (September) (2011) 1–12. doi:10.1145/2063384.2063454.

[38] D. Sanchez, C. Kozyrakis, ZSim: Fast and accurate microarchitectural simulation of thousand-core systems, Proceedings of the International Symposium on Computer Architecture (2013) 475–486doi:10.1145/2485922.2485963.

[39] T. Grass, C. Allande, A. Armejach, A. Rico, E. Ayguadé, J. Labarta, M. Valero, M. Casas, M. Moreto, Musa: A multi-level simulation approach for next-generation hpc machines, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2016.

[40] T. Karkhanis, J. Smith, A first-order superscalar processor model, Isca (2004) 338–349doi:10.1109/ISCA.2004.1310786.

[41] S. Eyerman, L. Eeckhout, K. D. Bosschere, Efficient Design Space Exploration of High Performance Embedded Out-of-Order Processors, Proceedings of the Design Automation & Test in Europe Conference 1 (2006) 351–356. doi:10.1109/DATE.2006.243735.

[42] D. Genbrugge, S. Eyerman, L. Eeckhout, Interval simulation: Raising the level of abstraction in architectural simulation, Hpca (2010) 1–12doi:10.1109/HPCA.2010.5416636.

[43] E. İpek, S. A. McKee, R. Caruana, B. R. de Supinski, M. Schulz, Efficiently Exploring Architectural Design Spaces via Predictive Modeling, in: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XII, ACM, New York, NY, USA, 2006, pp. 195–206. doi:10.1145/1168857.1168882.

[44] S. Zhang, A. Wright, D. Sanchez, Arvind, Validating Simplified Processor Models in Architectural StudiesarXiv:1610.02094.

[45] H. M. Nyew, N. Onder, S. Onder, Z. Wang, Verifying micro-architecture simulators using event traces, Proceedings of the 28th ACM international conference on Supercomputing - ICS '14 1 (c) (2014) 323–332. doi:10.1145/2597652.2597680.

[46] A. Gutierrez, J. Pusdesris, R. G. Dreslinski, T. Mudge, C. Sudanthi, C. D. Emmons, M. Hayenga, N. Paver, Sources of error in full-system simulation,

ISPASS 2014 - IEEE International Symposium on Performance Analysis of Systems and Software (2014) 13–22doi:10.1109/ISPASS.2014.6844457.

[47] A. Butko, R. Garibotti, L. Ost, G. Sassatelli, Accuracy evaluation of GEM5 simulator system, ReCoSoC 2012 - 7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip, Proceedingsdoi:10.1109/ReCoSoC.2012.6322869.

[48] A. Akram, L. Sawalha, A comparison of x86 computer architecture simulators, in: Technical Report, Western Michigan University ScholarWorks, 2016.

[49] M. Walker, S. Bischoff, S. Diestelhorst, G. Merrett, B. Al-Hashimi, Hardware-validated cpu performance and energy modelling, in: 2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2018, pp. 44–53. doi:10.1109/ISPASS.2018.00013.

[50] J. Gibson, R. Kunz, D. Ofelt, M. Horowitz, J. Hennessy, M. Heinrich, FLASH vs. (Simulated) FLASH, ACM SIGOPS Operating Systems Review 34 (5) (2000) 49–58. doi:10.1145/384264.379000.

[51] P. J. Mucci, S. Browne, C. Deane, G. Ho, Papi: A portable interface to hardware performance counters, in: In Proceedings of the Department of Defense HPCMP Users Group Conference, 1999, pp. 7–10.

[52] P. Hammarlund, 4th Generation Intel Core Processor, codenamed Haswell, 2013 IEEE Hot Chips 25 Symposium, HCS 2013doi:10.1109/HOTCHIPS.2013.7478321.

[53] A. Fog, A. Fog, Microarchitecture and Instruction tables, Optimization. URL http://www.agner.org/optimize/

[54] J. M. Cebrian, M. Jahre, L. Natvig, ParVec: vectorizing the PARSEC benchmark suite, Computing 97 (11) (2015) 1077–1100. doi:10.1007/s00607-015-0444-y.

[55] ARM, Scalable Vector Extension (2016).
URL https://community.arm.com/groups/processors/blog/2016/08/22/technology-update-the-scalable-vector-extension-sve-for-the-armv8-a-architecture

[56] NTNU, ParVec (2014).
URL http://www.ntnu.edu/ime/eecs/parvec

[57] C. Sakalis, C. Leonardsson, S. Kaxiras, A. Ros, Splash-3: A properly synchronized benchmark suite for contemporary research, in: I. C. Society (Ed.), International Symposium on Performance Analysis of Systems and Software (ISPASS), IEEE Computer Society, 2016, pp. 101–111.

[58] J.-E. Jo, G.-H. Lee, H. Jang, J. Lee, M. Ajdari, J. Kim, Diagsim: Systematically diagnosing simulators for healthy simulations, ACM Trans. Archit. Code Optim. 15 (1). doi:10.1145/3177959.

[59] Intel, Silvermont Pipeline (2015).
URL https://en.wikichip.org/wiki/intel/microarchitectures/silvermont