2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), and 2015 IEEE 12th International Conf on Embedded Software and Systems (ICESS)

SiNUCA: A Validated Micro-Architecture Simulator

Marco A. Z. Alves, Matthias Diener,
Francis B. Moreira, Philippe O. A. Navaux
Informatics Institute – Federal University of Rio Grande do Sul
Email: {mazalves, mdiener, fbmoreira, navaux}@inf.ufrgs.br

Carlos Villavieja
Google
Email: villavieja@google.com

***Abstract*—In order to observe and understand the architectural behavior of applications and evaluate new techniques, computer architects often use simulation tools. Several cycle-accurate simulators have been proposed to simulate the operation of the processor on the micro-architectural level. However, an important step before adopting a simulator is its validation, in order to determine how accurate the simulator is compared to a real machine. This validation step is often neglected with the argument that only the industry possesses the implementation details of the architectural components. The lack of publicly available micro-benchmarks that are capable of providing insights about the processor implementation is another barrier. In this paper, we present the validation of a new cycle-accurate, trace-driven simulator, SiNUCA. To perform the validation, we introduce a new set of micro-benchmarks to evaluate the performance of architectural components. SiNUCA provides a controlled environment to simulate the micro-architecture inside the cores, the cache memory sub-system with multi-banked caches, a *NoC* interconnection and a detailed memory controller. Using our micro-benchmarks, we present a simulation validation comparing the performance of real Core 2 Duo and Sandy-Bridge processors, achieving an average performance error of less than 9%.**

I. Introduction

Research on high-performance computing architectures depends on accurate and flexible simulation to enable the development of future generations of computer systems. Such research depends on simulators that provide detailed models of each subsystem, such as pipeline stages, functional units, caches, memory controllers and interconnections. Few existing public micro-architecture simulators have been verified for single core architectures [1]. However, with the latest advances in Chip Multiprocessor (CMP), Simultaneous Multi-Threading (SMT), branch prediction, memory disambiguation prediction [2], Non-Uniform Cache Architecture (NUCA) [3], Network-on-Chip (NoC) [4] and other mechanisms, the validation process of the simulators has not been continued. For multi-core architectures, mechanisms such as cache coherence, interconnection networks and memory controllers affect performance and increases the difficulty of validation. The use of these shared resources and their performance impact can vary significantly, depending on their implementation.

When validating a simulator, errors simulating full applications with mixed operations can lead to misleading conclusions, since errors from different components can cancel each other out. For this reason, micro-benchmarks that stress different hardware components independently can be used to correlate the implementation with real processors. The micro-benchmarks must be implemented in such a way that the performance of each component is evaluated separately, in order to isolate its impact. Furthermore, the operation of specific components in the processor is not published for intellectual property reasons. As a possible way to observe the behavior at a finer granularity, micro-benchmarks can be used as well.

For these reasons, writing an accurate and validated simulator for modern micro-architectures is a difficult task, and few publicly available simulators are validated for modern architectures. With these problems in mind, we developed the *Simulator of Non-Uniform Cache Architectures (SiNUCA)*, a performance validated, trace-driven, cycle accurate simulator for the x86 architecture. The simulator is capable of running single and multi-threaded applications and multiple workloads with a high level of detail of all the pipeline components. Additionally, we implemented a large set of micro-benchmarks to correlate the simulator results (performance and other statistics) with two existing x86 platforms. Due to its performance validation, SiNUCA enables simulation of new micro-architecture features with a high fidelity.

SiNUCA has the following main features:

Detailed Processor Model: SiNUCA implements architectural components, such as processor pipeline, memory and interconnections with a high level of detail. We also model parallel architectures such as multi-core and multi-processor systems. We used publicly available information for the implementation of the simulator. Where this information was not available, we used micro-benchmarks in order to observe the behavior of a real machine.

Validated with High Accuracy: We introduce a set of micro-benchmarks that cover the main performance relevant components: the control unit, dependency handling, execution units, as well as memory load and store operations on different memory levels. SiNUCA is validated with these micro-benchmarks in addition to real workloads. The simulation statistics are compared to real machines. The micro-benchmarks have an average difference of 9% comparing the Instructions per Cycle (IPC) when simulating machines for the Core 2 Duo and Sandy Bridge architectures.

Support for Emerging Techniques: SiNUCA is able to model several state-of-the-art technologies, such as NUCA, Non-Uniform Memory Access (NUMA), NoC and DDR3 memory controllers. Besides traditional micro-architecture enhancements such as cache pre-fetchers, branch predictors and others, the support for new technologies is important for accurate simulation of future processors.

Flexibility: Another important feature to support computer architecture research is the ease of implementing or extending features. This is enabled by SiNUCA with a modular architecture, which provides a direct access to the operational details of all the components. Other simulators are limited by metalanguages that do not expose all the functionalities of the micro-architecture, making it more difficult to model new mechanisms and modify existing ones.

II. Related Work

In this section, we analyze related computer architecture simulators and compare them to our proposed simulator.

Desikan et al. [1] validate *sim-alpha*, a simulator based on the *sim-out-order* version of SimpleScalar [5]. In this work, the authors aim to simulate the Alpha 21264 processor, using all available documentation. They use micro-benchmarks in order to scrutinize every aspect of the architecture, and are able to achieve an average error of 2% for a set of micro-benchmarks, and an error of 18% for the SPEC-CPU2000 applications. The authors identify the memory model as the main source of errors. They show that commonly used

978-1-4799-8937-9/15 $31.00 © 2015 IEEE
DOI 10.1109/HPCC-CSS-ICESS.2015.166

simulators, such as SimpleScalar, might contain modeling errors and become less reliable to test certain new features. We use a similar validation process for SiNUCA, making separate evaluations for specific components inside the processor by using micro-benchmarks. We extend the control and memory micro-benchmarks to evaluate modern branch predictors and multiple cache levels.

Virtutech SimICS [6] is a full-system, functional simulator which measures execution time based on the number of instructions executed multiplied by a fixed IPC, and the number of stall cycles caused by the latency of all components. In Weaver et al. [7], the SESC simulator [8] is compared to Dynamic Binary Instrumentation (DBI) using QEMU [9]. The authors show that in general, cycle-accurate simulators generate inaccurate results over a long execution time, due to lack of correctness in architectural details. They are able to obtain similar results in an order of magnitude shorter time with DBI. The paper also lists the flaws in cycle-accurate simulators. They cite speed, obscurity, source code forks, generalization, validation, documentation and lack of operating system influence as the major factors when considering the use of a simulator. Regarding these issues, SiNUCA solves several issues with code modularity, use of traced instructions and the validation.

Gem5 [10] is a combination of two simulators: M5 [11] and the General Execution-driven Multiprocessor Simulator (GEMS) [12]. The validation of the Gem5, modeling a simple embedded system [13], shows that errors can vary from 0.5% to 16% for the applications from the SPLASH-2 and ALPBench suites. However, for small synthetic benchmarks with tiny input sizes, the error varies from 3.7% to 35.9%. The authors conclude that the Dynamic Random Access Memory (DRAM) model is inaccurate.

PTLsim [14] is a cycle accurate, full-system x86-64 microprocessor simulator and virtual machine. It is integrated with the Xen hypervisor in order to provide full-system capabilities. Multi2Sim [15] is an event-driven simulator based on the premise of simulating multi-threaded systems. COTSon [16] is a simulator developed to provide fast and accurate evaluation of current and future computing systems, covering the full software stack and complete hardware models. Similar to SimICS, COTSon also abstracts processor microarchitecture details, prohibiting development of novelty at this level. MARSSx86 [17] is a cycle-accurate simulator based on PTLSim. Although it simulates all architectural details, it does not ensure accuracy, as can be seen in their comparison with the SPEC-CPU2006 workloads, where errors of up to 60% were obtained, with an average error of 21%.

Table I summarizes the main characteristics of a large set of well-known computer architecture simulators. Full-system simulation enables processor designers to evaluate OS improvements or its impact on the final performance. However, OS simulation can introduce noise during the evaluations, requiring several simulation repetitions (higher simulation time) in order to obtain reliable results with a reduced OS influence. Detailed micro-architectural simulation is required to evaluate most of the state-of-the-art component proposals.

TABLE I
COMPARISON OF COMPUTER ARCHITECTURE SIMULATORS.

Name	Full-system	Micro-arch.	Extension flexibility	NoC model	NUCA support	Memory controller
SimAlpha	No	Yes	High	No	No	No
SimICS	Yes	No	Low	No	No	No
SESC	No	Yes	High	Detailed	Yes	Extension
GEMS	No	Yes	Low	Simple	Yes	No
M5	Yes	Yes	High	Simple	Yes	No
Gem5	Yes	Yes	Low	Simple	Yes	No
PTLsim	Yes	Yes	High	No	No	Extension
Multi2Sim	No	Yes	High	Detailed	Yes	No
COTSon	Yes	No	Low	Detailed	Yes	No
MARSSx86	Yes	Yes	High	Detailed	No	No
SiNUCA	No	Yes	High	Detailed	Yes	Detailed

We consider that SimICS, GEMS, Gem5 and COTSon are not easy to extend because they have private source code, need metalanguages to modify the architecture or require modifications of multiple simulation levels in order to perform micro-architectural changes. Regarding NoC modeling, different detail levels can be observed among the simulators. SimAlpha, SimICS and PTLsim do not natively support it. GEMS, M5 and Gem5 model only the interconnection latency without modeling traffic contention. Considering NUCA, we classified the simulators as having support if they model at least multi-banked caches (static NUCA [3]).

The memory controller is becoming more important in modern processors because of its integration inside the processor chip. If a simulator only simulates a fixed latency for every DRAM request, we classify it as not capable of modeling a memory controller. Although SESC and PTLsim do not support memory controller modeling natively, extensions were proposed that overcome this deficiency.

III. THE SINUCA SIMULATOR

We developed SiNUCA, a trace-driven simulator, which executes traces generated on a real machine with a real workload without the influence from the Operating System (OS) or other processes. The traces are simulated in a cycle-accurate way, where each component is modeled to execute its operations on a clock cycle basis. SiNUCA currently focuses on the x86_32 and x86_64 architectures, but extending support for other Instruction Set Architectures (ISAs) is a simple task.

A. System Model

The main components of SiNUCA are illustrated in Figure 1. The description of the components is presented below:

Memory Package: Every memory operation inside the simulator is encapsulated within this component.

Opcode Package: The instructions inside the simulator trace and front-end are encapsulated within this component.

MicroOp Package: After decoding the Opcode Package, the micro-operations are encapsulated within this component.

Token: Every communication between two memory components, such as cache memories and memory controllers, needs to request a token from the final package destination before the communication starts. Tokens avoid deadlocks during package transfers that require more than one hop to reach their final destination.

Processor: This component is responsible for executing the Opcodes. It consists of the fetch, decode, rename, dispatch, execute and commit stages. Currently, an Out-of-Order (OoO) processor is modeled. The processor consists of 6 main stages, each stage can be configured to take multiple cycles to complete. Although the processor could be simulated with fewer stages, we implemented these stages separately in order to increase the flexibility and ease simulator extensions.

Branch Predictor: It is responsible for implementing any branch predictor mechanism with its Branch Target Buffer (BTB) and other structures. It returns to the processor if the branch was predicted correctly or not, together with the information about the stage in which the branch will be solved. Thus, the fetch stage will stall until the mispredicted branch reaches the informed stage. During each prediction, the prediction correctness information is also updated

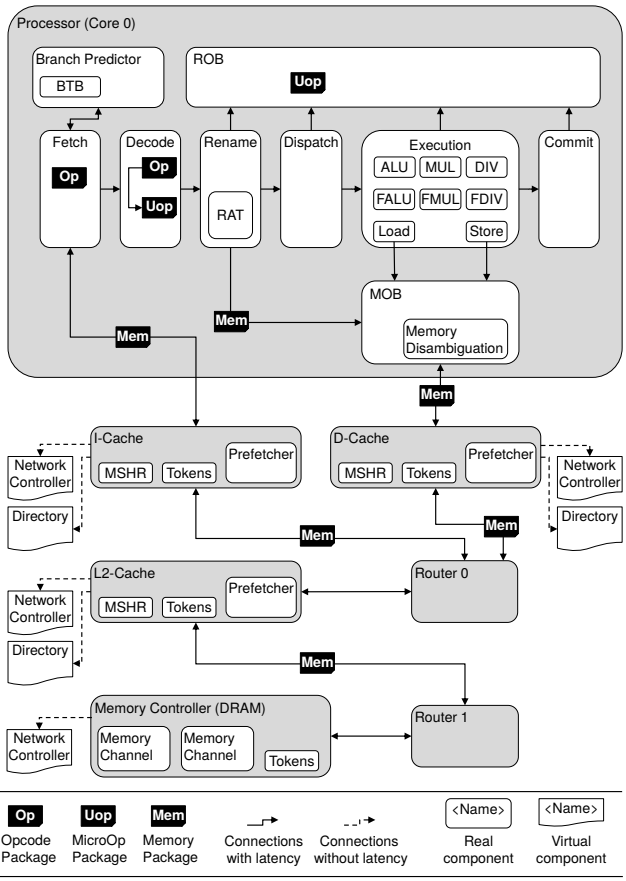

Fig. 1. SiNUCA architecture with its main components and connections, modeling an Intel Core 2 Duo architecture.

inside the predictor structures. Currently, the Per-address Adaptive branch prediction using per-Set PHT (PAs), Global Adaptive branch prediction using per-Set PHT (GAs), Per-address Adaptive branch prediction using one Global PHT (PAg) and Global Adaptive branch prediction using one Global PHT (GAg) branch predictor mechanisms are available in the simulator.

Cache Memory: This component is responsible for modeling instruction and data caches, both single and multi-banked models (static NUCA), implementing a Miss-Status Handling Registers (MSHR) internally per bank. This component only stores the tag array of the cache, reducing the memory usage and the trace size.

Pre-fetcher: The memory packages serviced by the cache memory are sent to the pre-fetcher, so that the memory addresses required by the application in the future can be predicted and fetched. Currently, a *stride pre-fetcher* and a *stream pre-fetcher* can be simulated.

Memory Controller: This component implements the memory controller, formed by multiple channels, each channel with multiple banks. A memory request that misses in all the cache levels will be sent to this component, which will schedule the request to the memory banks. The memory controllers also support NUMA modeling.

Memory Channel: Inside the memory controller, multiple memory channels can be instantiated. Each channel may be connected to one or more banks, each bank with its own read and write buffers.

Router: This component models the contention and delays imposed by the interconnections. It implements a general NoC router that automatically delivers packages using the routing information inside each package.

Network Controller: This controller is used to generate a communication graph with the routes available in the modeled architecture. All packages that need to be transmitted have a pointer to the routing information contained in the routing table. The routing information describes which path to take for each intermediary component (routing input/output ports).

Directory: This component models the MOESI cache coherence protocol, which is accessed directly by the caches for every memory operation. It is responsible for creating locks on cache lines to control concurrent operations, to change the cache line status (for example, after writes or when propagating invalidations), to generate write-backs and to inform the caches about hits or misses. During read requests, the directory creates a read lock to that memory address, thus only read accesses can be provided to that address in any cache in the system. During write operations, the directory locks the cache line. In this way, no other operation can be performed by other caches on the same address until the lock is released.

The network controller and the directory are virtual components, which model lookup tables but do not generate any latency on accesses to them. All other components include a detailed implementation and the connections between them generate latency operations when communicating with other components.

IV. Micro-Benchmarks

Many architectural details of modern processors are not publicly available. For this reason, micro-benchmarks are needed to evaluate and estimate the behavior of the processor components. We developed a suite of micro-benchmarks to isolate and validate specific parts of SiNUCA. These benchmarks are presented in this section. Our micro-benchmarks are inspired by the SimAlpha validation [1]. Four categories of benchmarks are defined: *Control*, *Dependency*, *Execution* and *Memory*. These categories stress different stages of the pipeline and evaluate different components separately. Table II presents the micro-architecture details that each benchmark stresses.

1) *Control* benchmarks are designed to stress the execution flow.

Control Conditional implements a simple if-then-else construct in a loop that is repeatedly executed, and alternates between taking and not taking the conditional branch.

Control Switch tests indirect jumps with a *switch* construct formed by 10 case statements within a loop. Each case statement is taken $n/10$ times on consecutive loop iterations before moving to the next case statement, where n is the total number of repetitions of the loop.

Control Complex mixes *if-else* and *switch* constructs in order to create a hard to predict branch behavior.

Control Random implements an if-then-else construct in a loop that is repeatedly executed. For every iteration, a Linear Feedback Shift Register function decides whether the conditional branch is taken.

Control Small BBL evaluates the number of simultaneous in-flight branches inside the processor by executing a loop with only the control variable being incremented inside the loop.

2) *Dependency* benchmarks stress the forwarding of dependencies between instructions. They evaluate dependency chains of 1–6 instructions. Each instruction waits for the execution of the previous one due to the dependency, evaluating the data forwarding time.

3) *Execution* benchmarks stress the functional units: **int-add**, **int-multiply**, **int-division**, **fp-add**, **fp-multiply**, and **fp-division**. All execution benchmarks execute 32 independent operations inside a loop, with a low number of memory operations, control hazards and data dependencies, allowing a close-to-ideal throughput.

TABLE II
MICRO-BENCHMARKS LIST AND MICRO-ARCHITECTURAL DETAILS STRESSED BY EACH BENCHMARK.

		Control					Dependency						Execution						Mem. Load Dependent												Mem. Load Independent												Mem. Store Independent											
		Complex	Conditional	Random	Small_bbl	Switch	Chain-1	Chain-2	Chain-3	Chain-4	Chain-5	Chain-6	FP-add	FP-div	FP-mul	INT-add	INT-div	INT-mul	00016kb	00032kb	00064kb	00128kb	00256kb	00512kb	01024kb	02048kb	04096kb	08192kb	16384kb	32768kb	00016kb	00032kb	00064kb	00128kb	00256kb	00512kb	01024kb	02048kb	04096kb	08192kb	16384kb	32768kb	00016kb	00032kb	00064kb	00128kb	00256kb	00512kb	01024kb	02048kb	04096kb	08192kb	16384kb	32768kb
Characteristic	Branch Predictor	•	•	•	•	•																																														
	Branch Miss Penalty			•																																															
	In-flight Branches				•																																															
	Register File						•	•	•	•	•	•																																								
	Functional Units												•	•	•	•	•	•																																		
	L1 Cache																		•	•											•	•												•	•								
	L2 Cache																				•	•	•										•	•	•												•	•	•					
	LLC																							•	•	•	•	•	•							•	•	•	•	•	•							•	•	•	•	•	•
	DRAM																												•													•											•
	Prefetcher																		•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•

4) *Memory* benchmarks stress the cache and memory hierarchy.

Load Dependent executes a loop that walks a linked list, waiting for each load to complete before starting the next. Twelve different linked list sizes are used in the evaluation.

Load Independent repeatedly executes 32 parallel independent loads from an array and sumps up their values in a scalar variable. Twelve array sizes were evaluated in order to stress different cache levels.

Store Independent repeatedly performs 32 parallel independent stores of a fixed scalar, iterating over all the positions of an array. The same twelve array sizes as for the load independent are evaluated.

V. EVALUATION

To evaluate and validate SiNUCA, we performed extensive experiments with a large set of benchmarks. This section presents the methodology of the experiments, validation results and a discussion of the differences between the simulator and the real machine.

A. Evaluation Methodology

In order to reduce the influence from the operating system, the experiments in the real machine were repeated 100 times. For each execution, we set a static affinity between the thread and the processor core to reduce the overhead of migration. We equalized the micro-benchmarks execution time to 3 seconds on the real machine. This ensures that the amount of work performed is significant, while keeping the simulation time reasonable for the full execution. To obtain the performance counters (cycles and instructions), we used the `perf` tool included in Linux.

TABLE III
PARAMETERS TO MODEL THE CORE 2 DUO PROCESSOR.

Processor Cores: 2 cores @ 1.8 GHz, 65 nm; 4-wide Out-of-Order; 16 B fetch block size; 1 branch per fetch; 8 parallel in-flight branches; 12 stages (2-fetch, 2-decode, 2-rename, 2-dispatch, 2-commit stages); 18-entry fetch buffer, 24-entry decode buffer; 96-entry ROB; MOB entries: 24-read, 16-write; 1-load and 1-store units (1-1 cycle); 3-alu, 1-mul. and 1-div. int. units (1-4-26 cycle); 1-alu, 1-mul. and 1-div. fp. units (3-5-5 cycle);
Branch Predictor: 4 K-entry 4-way set-assoc., LRU policy BTB; Two-Level PAs 2-bits; 16 K-entry PBHT; 128 lines and 1024 sets SPHT;
L1 Data + Inst. Cache: 32 KB, 8-way, 64 B line size; LRU policy; 1-cycle; MSHR: 8-request, 10-write-back, 1-prefetch; Stride prefetch: 1-degree, 16-strides table;
L2 Cache: Shared 2 MB; 8-way, 64 B line size; LRU policy; 4-cycle; MOESI coherence protocol; MSHR: 2-request, 6-write-back, 2-prefetch; Stream prefetch: 2-degree, 16-dist., 64-streams;
Memory Controller: Off-chip DRAM controller; 2-channels, 4 burst length; DDR2 667 MHz; 2.8 core-to-bus frequency ratio; 8 banks, 1 KB row buffer; Open-row first policy; 4-CAS, 4-RP, 4-RCD and 30-RAS cycles;

B. Real Machine and Simulation Parameters

Table III shows the parameters used to model the Intel Core 2 Duo (Conroe microarchitecture - model E6300) [18]). Table IV presents the parameters from Intel Xeon (Sandy Bridge microarchitecture - model Xeon E5-2650) [19] architectures inside SiNUCA. The tables show parameters of the execution cores, branch predictors, cache memories and pre-fetchers, as well as the memory controllers.

C. Micro-Benchmark Results

Figure 2 presents the comparison in terms of Instructions per Cycle (IPC) for the micro-benchmarks running on the real Core 2 Duo (real) and with SiNUCA (sim). For Core 2 Duo, the geometric mean of the absolute IPC difference for all the micro-benchmarks is 10%. The average error was 9% for the control category, 8% for the dependency, 1% for execution, 26% for memory load dependent , 5% for memory load independent and 22% for memory store independent.

Figure 3 shows results for the Sandy Bridge (real) machine and SiNUCA (sim). For Sandy Bridge, the geometric mean of the absolute IPC difference for all the micro-benchmarks is 6%. The average error was 1% for the control category, 1% for the dependency, 2% for execution, 12% for memory load dependent, 6% for memory load independent and 26% for memory store independent.

The results of each micro-benchmark category are discussed below:

Control benchmarks: Evaluating the control benchmark *Small_bbl*, we conclude that the maximum number of parallel predicted branches

TABLE IV
PARAMETERS TO MODEL THE SANDY BRIDGE PROCESSOR.

Processor Cores: 8 cores @ 2.0 GHz, 32 nm; 4-wide Out-of-Order; 16 B fetch block size; 1 branch per fetch; 8 parallel in-flight branches; 16 stages (3-fetch, 3-decode, 3-rename, 2-dispatch, 3-commit stages); 18-entry fetch buffer, 28-entry decode buffer; 168-entry ROB; MOB entries: 64-read, 36-write; 1-load and 1-store units (1-1 cycle); 3-alu, 1-mul. and 1-div. int. units (1-3-32 cycle); 1-alu, 1-mul. and 1-div. fp. units (3-5-10 cycle);
Branch Predictor: 4 K-entry 4-way set-assoc., LRU policy BTB; Two-Level GAs 2-bits; 16 K-entry PBHT; 256 lines and 2048 sets SPHT;
L1 Data + Inst. Cache: 32 KB, 8-way, 64 B line size; LRU policy; 2-cycle; MSHR: 8-request, 10-write-back, 1-prefetch; Stride prefetch: 1-degree, 16-strides table;
L2 Cache: Private 256 KB, 8-way, 64 B line size; LRU policy; MSHR: 4-request, 6-write-back, 2-prefetch; 4-cycle; Stream prefetch: 2-degree, 32-dist., 256-streams;
L3 Cache: Shared 16 MB (8-banks), 2 MB per bank; Bi-directional ring; 16-way, 64 B line size; LRU policy; 6-cycle; Inclusive; MOESI coherence protocol; MSHR: 8-request, 12-write-back;
Memory Controller: Off-chip DRAM controller, 4-channels, 8 burst length; DDR3 1333 MHz; 3 core-to-bus frequency ratio; 8 banks, 1 KB row buffer; Open-row first policy; 9-CAS, 9-RP, 9-RCD and 24-RAS cycles;

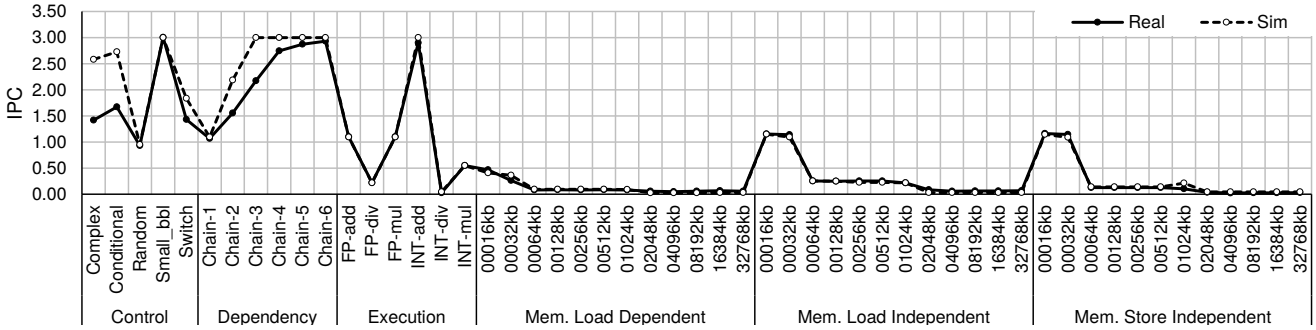

Fig. 2. Performance results for the Core 2 Duo.

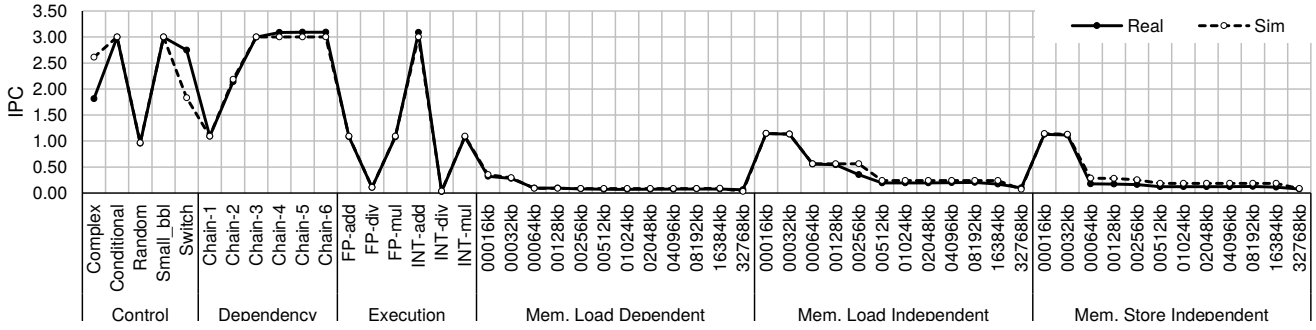

Fig. 3. Performance results for the Sandy Bridge.

in execution inside the pipeline is equal to 8 for both machines. With the *Random* benchmark, we calculate the misprediction penalty as 20 cycles for Core 2 Duo and 14 cycles for Sandy Bridge. The first source of differences regarding the two-level branch predictor implemented inside SiNUCA is that this well known mechanism is not guaranteed to be exactly the same as the one implemented in a real machine. Since this mechanism's prediction is sensitive to its implementation, small implementation differences can lead to different predictions and also performance.

Dependency benchmarks: Observing the behavior of dependency benchmarks, we can notice that for *Chain-1* and *Chain-6* the simulation obtains very close results to the real machines. Inside the simulator, all the dependencies are solved one cycle after the result is available. For *Chain-1*, we can observe that the latency for data forwarding is modeled correctly. However, the results from *Chain-2* to *Chain-5* show that the real machines have a limited maximum number of data forwardings per cycle. For *Chain-6*, the instruction parallelism is high enough to hide this bottleneck. We observe that Core 2 Duo has a higher differences due to its limited number of forwarding paths.

Execution benchmarks: Although we obtained very accurate results for the integer and floating point applications (less than 4% of difference), we notice that *INT-div* has a very low IPC, which is caused by the high latency of the functional unit. Due to this low IPC, the performance difference is high (35% on average) compared to the simulation for this particular benchmark. Additionally, the latencies for instructions that use the same functional unit can vary in a real machine. However, in SiNUCA, the execution latencies are defined by the functional unit used, and not by the instruction itself. Such modification in the simulator would be impractical since there are hundreds of instructions in the x86-64 ISA.

Memory benchmarks: Although the average relative error for memory load dependent and memory store independent categories appear to be high, their absolute error is low due to the low IPC. For instance, the IPC in the Core 2 Duo for the memory store independent with 32 MB on the real machine is 0.030 while in the simulator it is 0.046, resulting in an relative error of 55%. Such a low absolute error tends not to affect the performance of real applications with a mixed types of operations.

The memory pre-fetching algorithms implemented in SiNUCA are well known techniques. However, the information about the real processor pre-fetcher is missing some details and it is not guaranteed that the simulated pre-fetchers perform exactly the same way as the real ones. This difference also happens because small changes in the pre-fetcher parameters, such as number of strides, can have a high impact on the Misses per Kilo Instructions (MPKI). The memory controller scheduling policy can also cause performance differences, depending on the real hardware implementation. For our experiments, the common open-row first policy was used.

D. SPEC-CPU2006 Results

In order to evaluate the simulator results with real benchmarks, this section presents the IPC results of executing the applications from the SPEC-CPU2006 suite [20] on both evaluated architectures. The results for the real machines consider the full application execution, while the simulation results were obtained executing a representative slice of 200 million instructions of each application, selected by PinPoints [21].

Figure 4 presents the performance results for the Core 2 Duo (real) and with SiNUCA (sim). The average IPC error for this benchmark suite was 19%, where the integer applications had an average error of 24% and the floating point applications an error or 16%. Figure 5 presents the performance results for the Sandy Bridge machine. The

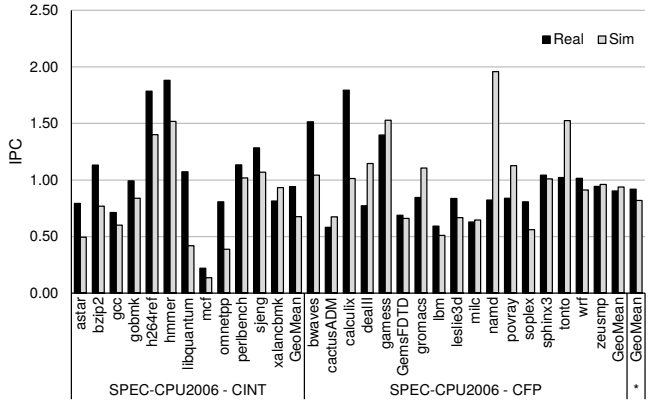

Fig. 4. Performance results for the Core 2 Duo.

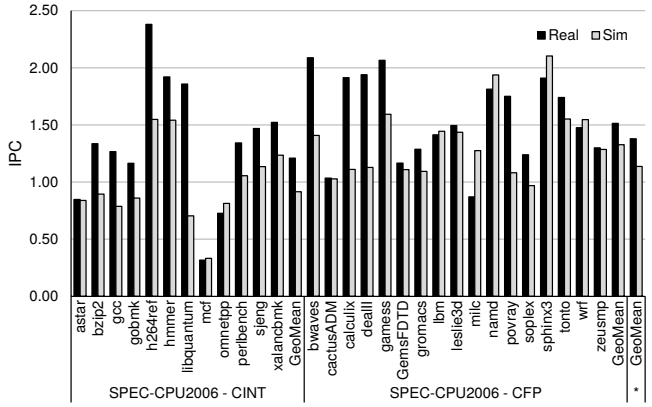

Fig. 5. Performance results for the Sandy Bridge.

average IPC error for this benchmark suite was 12%, where the integer applications had an average error of 17% and the floating point applications an error or 10%. Compared to the related work presented in Section II, SiNUCA presents a moderately higher accuracy.

For the SPEC-CPU2006 benchmarks, the main source of difference was the branch predictor, with an average error of 19% on both architectures. In our experiments, we could observe that applications behave differently when switching between the PAs and GAs branch predictors, in such way that a hybrid predictor is being considered to reduce the gap between the real and simulator difference. This, together with the fact that we simulate only a slice of the application, explains the higher differences compared to the micro-benchmarks. It also indicates that the interaction between components has to be analyzed in more detail, suggesting the extension of the micro-benchmarks to study this interaction.

VI. Conclusions

We presented SiNUCA, a performance validated micro-architecture simulator. We also introduced a set of micro-benchmarks that stresses separate architectural components. SiNUCA supports simulation of emerging techniques and is easy to extend. Our validation showed an average difference on performance of 10% with a set of micro-benchmarks when simulating a Core 2 Duo machine. For a Sandy Bridge system, the average performance difference was 6%. Evaluation results with the SPEC-CPU2006 benchmarks showed higher accuracy compared to previously proposed simulators. SiNUCA and the microbenchmarks are licensed under the GPL and publicly available online at `https://github.com/mazalves/sinuca` and at `https://github.com/mazalves/microbenchmarks`. As future work, we will compare the energy consumption between the real machine and SiNUCA coupled to an energy modeling tool such as McPAT. We also intend to perform a detailed validation with parallel applications.

Acknowledgments

The authors gratefully acknowledge the support of CNPq and CAPES.

References

[1] R. Desikan, D. Burger, and S. Keckler, "Measuring experimental error in microprocessor simulation," in *Proc. IEEE/ACM Int. Symp. on Computer Architecture*, 2001.

[2] J. Doweck, "White paper inside intel® core® microarchitecture and smart memory access," *Intel Corporation*, 2006.

[3] C. Kim, D. Burger, and S. W. Keckler, "Non-uniform cache architetures for wire-delay dominated on-chip caches," *IEEE Computer*, 2003.

[4] T. Bjerregaard and S. Mahadevan, "A survey of research and practices of network-on-chip," *ACM Computing Surveys (CSUR)*, vol. 38, 2006.

[5] T. Austin, E. Larson, and D. Ernst, "Simplescalar: An infrastructure for computer system modeling," *IEEE Micro*, vol. 35, 2002.

[6] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *IEEE Micro*, vol. 35, Feb 2002.

[7] V. Weaver and S. McKee, "Are cycle accurate simulations a waste of time," in *Workshop on Duplicating, Deconstructing, and Debunking*, 2008.

[8] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos, "SESC simulator," January 2005.

[9] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX Annual Technical Conference, FREENIX Track*, 2005.

[10] N. Binkert, B. Beckmann, G. Black *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, 2011.

[11] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, and S. Reinhardt, "The m5 simulator: Modeling networked systems," *IEEE Micro*, 2006.

[12] M. Marty, B. Beckmann, L. Yen, A. Alameldeen, and K. M. Min Xu, "General execution-driven multiprocessor simulator," in *Proc. ISCA Tutorial*, 2005.

[13] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli, "Accuracy evaluation of gem5 simulator system," in *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on*, 2012.

[14] M. Yourst, "Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator," in *Proc. Int. Symp. on Performance Analysis of Systems & Software*, 2007.

[15] R. Ubal, J. Sahuquillo, S. Petit, and P. López, "Multi2sim: A simulation framework to evaluate multicore-multithread processors," in *International Symposium on Computer Architecture and High Performance computing*, 2007.

[16] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega, "Cotson: infrastructure for full system simulation," *ACM SIGOPS Operating Systems Review*, vol. 43, 2009.

[17] A. Patel, F. Afram, S. Chen, and K. Ghose, "Marssx86: A full system simulator for x86 cpus," in *Design Automation Conference*, 2011.

[18] T. Bojan, M. Arreola, E. Shlomo, and T. Shachar, "Functional coverage measurements and results in post-silicon validation of core 2 duo family," in *Int. High Level Design Validation and Test Workshop*, 2007.

[19] M. Yuffe, E. Knoll, M. Mehalel, J. Shor, and T. Kurts, "A fully integrated multi-cpu, gpu and memory controller 32nm processor," in *Int. Solid-State Circuits Conference Digest of Technical Papers*, 2011.

[20] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, 2006.

[21] H. Patil, R. S. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of large intel® itanium® programs with dynamic instrumentation," in *Proc. IEEE/ACM Int. Symp. on Microarchitecture*, 2004.