XIOSim: power-performance modeling of mobile x86 cores

Article · July 2012

DOI: 10.1145/2333660.2333722

CITATIONS
15

READS
94

3 authors, including:

Gu-Yeon Wei
Harvard University
234 PUBLICATIONS **7,690** CITATIONS
SEE PROFILE

David C Brooks
Brigham and Women's Hospital
283 PUBLICATIONS **12,893** CITATIONS
SEE PROFILE

XIOSim: Power-Performance Modeling of Mobile x86 Cores

Svilen Kanev Gu-Yeon Wei David Brooks
Harvard University
33 Oxford St., Cambridge, MA
{skanev, guyeon, dbrooks}@eecs.harvard.edu

ABSTRACT

Simulation is one of the main vehicles of computer architecture research. In this paper, we present XIOSim – a highly detailed microarchitectural simulator targeted at mobile x86 microprocessors. The simulator execution model that we propose is a blend between traditional user-level simulation and full-system simulation. Our current implementation features detailed power and performance core models which allow microarchitectural exploration. Using a novel validation methodology, we show that XIOSim's performance models manage to stay well within 10% of real hardware for the whole SPEC CPU2006 suite. Furthermore, we validate power models against measured data to show a deviation of less than 5% in terms of average power consumption.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling techniques

General Terms

Performance, Measurement, Experimentation

Keywords

simulation, x86, power-performance, in-order

1. INTRODUCTION

Architecture simulation has been used extensively both in the early stages of a processor design and in architecture research. During the design process, performance estimates from simulations of various levels of detail are used to motivate architectural decisions. Simulators are also heavily used research tools, because they provide a relatively inexpensive means of exploring new ideas.

After power has established itself as a significant metric in microprocessor design, there has been a tendency to produce simpler cores because of their better performance per watt. Simple, in-order cores have been the norm for mobile applications, but recent industrial designs [10, 14, 15] explore the opportunity of using multiple such cores for parallel workloads found in datacenters, multimedia or networking. The simulator we developed, XIOSim, is intended as an integrated model of such simple in-order cores. Unlike other microarchitectural simulators used in the research community [2, 3], XIOSim targets the IA-32 instruction set due to its widespread adoption. To the best of our knowledge, it is the only publicly available (`http://xiosim.org`) detailed microarchitectural model of in-order IA-32 cores.

Genealogically, XIOSim was derived from the Zesto x86 user-level simulator [8], which in turn originates from SimpleScalar [3]. XIOSim extends the Zesto framework by providing a different, binary instrumentation-based functional execution model, a detailed in-order core model, and by integrating very closely with the McPAT power models [7] for producing power consumption traces.

In relation to other architectural simulators, we believe XIOSim has a niche in the design space. As opposed to other frameworks that support the IA-32 instruction set [2, 11, 16], it is not geared towards full-system simulation, but rather focuses on user-level, CPU-intensive workloads in isolation. Furthermore, its core models, albeit slower, are very detailed and strive for accuracy. While we do not go as far as to claim cycle-accuracy, such a level of detail allows us to validate the simulator against a real machine with Intel Atom cores across a statistically significant sample of the 29 SPEC CPU2006 benchmarks. This allows architectural exploration that is grounded on current industrial designs.

Finally, XIOSim's execution model can support executing workloads that are out of reach for traditional user-level simulators. By using the host hardware as a functional model, it can continue correct simulation in the presence of instructions or system calls that are not explicitly supported by the performance model. Thus, in implementing XIOSim there is no need for heroics such as modelling every instruction in the complex IA-32 instruction set, or explicitly capturing every POSIX system call.

In order to make sure that XIOSim is reasonably accurate, we have developed a validation methodology that enables comparing performance and power characteristics of short benchmark fragments. This allows us to compare power and performance of the exact same instruction sequences both running on real hardware and being simulated with a significant slowdown. Doing such a comparison eliminates the need for sampled fragments to be representative of full program execution and allows for a fair validation procedure.

The contributions of this paper are as follows:

- A simulator execution model that executes instruc-

ISLPED'12, July 30–August 1, 2012, Redondo Beach, CA, USA.

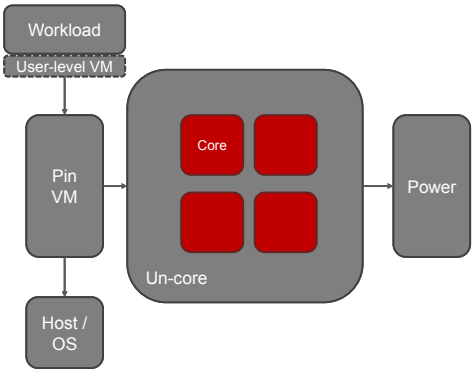


Figure 1: XIOSim execution model.

tions on the host hardware and observes their effects through binary instrumentation.

- A detailed methodology for validating simulators against real hardware based on running full benchmarks, even under a significant speed difference.

- A publicly available very detailed performance model for in-order IA-32 cores, augmented with a power model.

In the beginning of this paper, Section 2 motivates a simulator execution model based on instrumenting instructions running on real hardware. Section 3 describes a mobile IA-32 core model that runs under this execution model. Section 4 describes the extensive validation methodology that we used while developing the power/performance models for XIOSim and finally, Section 5 presents validation results.

2. EXECUTION MODEL

The simulator execution model we propose is a mixture between traditional user-level simulation and full-system simulation. XIOSim is a user simulator in that it does not model instructions past system call boundaries. However, it runs as a pintool, under the Pin virtual machine [9], which translates application instructions to run natively on the simulation host.

Figure 1 sketches out this execution model showing the interactions between functional, performance and power simulation. The user-level workload is run under Pin control, with the Pin VM instrumenting every instruction and capturing the host machine state immediately before it. That instruction context is passed to the performance model, which is expecting it at the fetch stage of the modelled pipeline, effectively achieving execute-at-fetch semantics. At certain cycle intervals, simulated statistics are aggregated and passed to a modified version of McPAT [7] to create a power consumption trace.

The closest analog to this execution model is implemented in PTLSim [16] and MARSS [11]. In these simulators, the performance model is run as an extension to a system-level virtual machine monitor or a full-system emulator. In our approach, keeping the simulator completely in user-level, while limiting the scope of benchmarks whose performance will be accurately modeled, is beneficial from a simplicity and a stability point of view. Full-system simulators need to run a complete OS image, virtual device drivers, etc., all

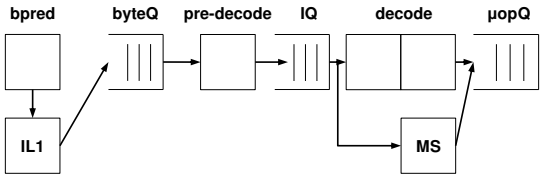


Figure 2: Front-end pipeline model.

of which affect simulation accuracy and reliability. For microarchitectural studies, it is often beneficial to ignore such behavior and only focus on the characteristics of the benchmark under test in isolation.

Furthermore, this model has benefits over traditional user-level simulation. First, all system code, including system call handling, is directly executed on the host, freeing up the simulator from implementing a pass-through layer and keeping up with relatively fast-changing system call interfaces. This allows simulating a broader range of applications than traditional user-level simulation, where system call support is typically added based on the popularity of the particular call. For example, this has allowed us to simulate workloads under the ILDJIT user-level virtual machine [4] that translates CIL bytecode in IA-32 instructions.

Second, by running under an instrumentation engine, the performance model does not require a complete or correct functional model in order to maintain the correct execution of the workload. Since instructions are executed on native hardware, the performance model can only observe their effects without explicit knowledge of instruction semantics. This allows continuing execution even after encountering esoteric unsupported instructions, which is especially important for simulating a complex and constantly evolving instruction set like IA-32.

Finally, this execution model allows for close-to-native speeds of execution of code outside simulation regions of interest. This enables efficient skipping of non-interesting benchmark phases and integration with statistical sampling tools, such as the SimPoint-based PinPoints [12].

The major disadvantage with such an approach comes from speculative execution. Pin cannot instrument instructions on a speculative path since by definition they are not visible above the architecture level of abstraction. In order to deal with that, XIOSim keeps a legacy functional model that is only used on speculative paths. On simulating a mispredicted branch, this functional model is invoked, and the simulated core continues utilizing it until the branch is resolved in the execute portion of the pipeline. At this point, the functional model is switched back to the instrumentation engine. Note that, to ensure correctness, the legacy functional model does not need to be completely correct or accurate, since the mispredicted branch will get resolved regardless of the instructions on the speculative path.

3. PERFORMANCE MODEL

In this section, we describe the in-order core performance model used in XIOSim. Overall, we model an in-order, multi-issue x86 core, a proxy for Atom-style microarchitectures [5]. Its pipeline is typically deeper than the ones found in mobile cores, partially due to the complexities involved in decoding a variable-length ISA, as well as due to the desire for efficiently handling CISC instructions with memory

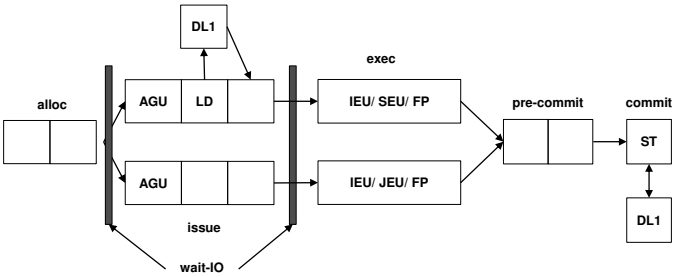
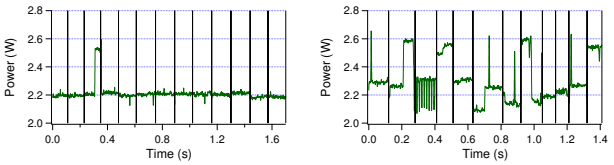


Figure 3: Back-end pipeline model, configured to model an Atom-like architecture.



(a) *453.povray* (b) *447.dealII*

Figure 4: Power traces with qualitatively different behavior, but similar average power.

operands without out-of-order execution.

The front-end model, as seen in Figure 2, is typical for an IA-32 architecture. The pipeline starts with branch prediction, which directs accesses to the instruction cache. It continues with explicit pre-decode stages, whose aim is to predict instruction boundaries in the raw byte stream that was fetched. Pre-decoded instructions then propagate to the asymmetrical decoders, which break the possibly complex instructions into RISC-like μ-ops. The decoders can process multiple instructions in parallel, in order to keep the rest of the multiple-issue pipeline occupied. Complex instructions are sent to the micro-sequencer for decoding, which incurs an additional penalty of several cycles.

The decoders make heavy use of μ-op fusion – combining multiple μ-ops from a single instruction, such that they progress together through the pipeline, not incurring additional latency and occupying less space in queue structures. Out-of-order x86 microarchitectures have traditionally used load-op fusion, which combines a load with its dependant subsequent operation. In a deep in-order pipeline not incurring latency costs is even more important and fusion can be applied more aggressively – combining loads, their dependant operations and a store from a complex, memory-reference instruction. This results in being able to execute a large fraction of instructions in a CISC-like fashion.

Decoded and possibly fused μ-ops then enter the allocation stages of the pipeline (see Figure 3), where they access the register file, and get assigned to an execution port based on functional unit availability and port loading. Because decoding latencies can vary and different execution ports can stall independently of each other, there is an explicit in-order synchronization point at the end of the allocation pipeline. Its purpose is to enforce program order of μ-ops entering the execution ports, so that data dependencies can be observed without any of the complexities of an out-of-order pipeline.

Once in an execution port, μ-ops go through issue stages, mostly dedicated to memory accesses. These include address generation, load dispatch and return from the data cache. Having dedicated pipeline stages for processing loads is in unison with μ-op fusion, effectively achieving a zero-cycle load-to-use penalty for load-op or load-op-store fusions.

There is another in-order synchronization point at the end of the issue stages. Since all execution latencies after it are deterministic, this point ensures that μ-ops are properly serialized for in-order commit. Once executed and serialized, μ-ops enter a pre-commit pipeline stage which models exception handling. Then they follow to the commit buffer, where stores are sent to the data cache and individual μ-ops are composed back to macro instructions in order to commit atomically, observing instruction boundaries.

4. VALIDATION METHODOLOGY

We will show that the performance model described earlier can be configured to match a real design and is therefore useful as a performance exploration tool, as well as a power model driver. We start by presenting the extensive validation methodology that we used for developing XIOSim.

4.1 Fine-grained comparison

The major difficulty in validating a detailed model against real hardware arises from the discrepancy in speed. While real hardware can often commit instructions at rates of 10s billions instructions per second (BIPS), a detailed performance model typically achieves 10-100s kilo-instructions per second (KIPS). Simulating full real-world benchmarks is therefore prohibitively slow and a sampling approach has to be used. This presents two possible reference points for the sampled benchmark run: (i) either data gathered from the full benchmark execution, or (ii) collected data from a similarly sampled run on a real machine.

The first approach is conveniently simple, but has several issues. First, when doing sampling, the short execution slices are selected with a particular metric in mind. For example, SimPoints [13] targets end performance and the selected slices can be used to estimate the full program's execution time. However, this does not guarantee that SimPoint-derived slices are representative of the full program's branch behavior or power consumption, since these metrics are not always strongly correlated with performance.

Furthermore, even when used with the appropriate metric, the predictions from sampling are within some confidence interval of the full-length values. Patil et al. [12] demonstrate that this error can be as high as 10% for predicting CPI with SimPoints. This makes it impossible to separate simulator errors from sampling errors in a validation scenario.

For these reasons, we choose the second validation approach. Since sampling is inevitable when dealing with the slowdown of detailed simulation, we do sample with end performance in mind for our simulated runs. For the reference runs on a real machine, we use a very similar sampling approach, making sure we gather data only during the same instruction intervals that are simulated. The difficulty in this case lies in gathering data for a short period (measured in millions of instructions) without either modifying the workload source code, or perturbing the experiment with the measurement code.

Without source modification, one can use binary instrumentation to trigger the appropriate measurements once execution reaches the simulated region. However, the instrumentation must be light enough in order not to introduce

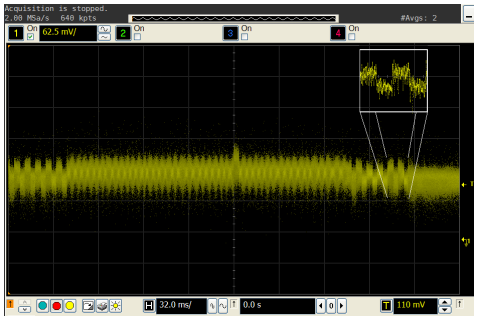


Figure 5: Sample power trace corresponding to a single slice between power markers. Inset shows an expanded view of the pattern used for marking the start and end of slices.

errors in measurements on the host machine. For example, in the Pin virtual machine even inserting a simple basic block counting instrumentation routine can have overheads in execution time on average exceeding 100% [9].

In order not to pay this penalty, we use Pin to insert lightweight trampolines that effectively replace routines of interest with versions that have thin instrumentation code inserted. The instrumentation code tracks function call counts and determines when it is appropriate to start or stop data collection. The routines of interest are defined as the closest ones to the beginning and the end of simulation slices, as identified by SimPoints, and the target call counts for them are gathered during an initial profiling run.

The stub code that we insert is a hand-optimized assembly sequence tracking the number of times the routine has been called and toggling data capture when appropriate. The measured average instruction overhead of this scheme over the SPEC CPU2006 suite is 0.7%. For performance validation, the stub code triggers performance counter collection on the host machine through the Linux perfmon2 interface. Its behavior in power validation mode is described in the following section.

4.2 Targeting power

Measurement setup. Our particular power capture system taps in the processor 12V power supply line and measures current consumption through a low-impedance high-precision sense resistor. At a fixed nominal supply voltage this provides enough data to capture processor power consumption. The traces are collected using a high-end Aglient DSA-91304A oscilloscope and a DSA-3100 differential probe. Trace collection is triggered remotely over the network. The high-end collection system allows us to gather power traces for a short amount of time with enough resolution to capture intra-slice power behavior. The sampling rate used in the following experiments is set to 160 kHz.

The benefit of being able to capture single execution slices can be seen in Figure 4, which shows power traces for samples of the SPEC benchmarks *453.povray* and *447.dealII*. If we average power over all execution slices, the difference between the two benchmarks is only 4%. However, in order to perform any meaningful power validation, we need to be able to capture and compare individual slice behavior for benchmarks like *447.dealII*.

Aligning code and measurements. Since executing a single sampled slice on a real machine can take time on the order of 10ms, which is comparable to the network delay to trigger power capture, explicit measures have to be taken to synchronize the power trace collection with the slice execution. We use the low-overhead instrumentation described in Section 4.1 to insert power markers at the beginning and end of a sampled slice. The markers are repeating sequences of a hand-crafted power virus instance (derived from CPUBurn and modified for an in-order pipeline) and suspending the core by executing a `usleep()` system call. By sandwiching the slice execution with long enough markers, we can mitigate the various variable delays and latencies in the measurement system and be also sure that the correct execution slice has been captured. A sample trace captured with this methodology is shown in Figure 5, with two iterations of a power marker in the inset.

In a post-processing step, the power markers are programmatically removed from the collected trace, so that only the slice region-of-interest is accounted for. The marker removal algorithm operates by searching for the specific frequency footprint of the power markers.

5. VALIDATION RESULTS

We applied the validation methodology described in the previous section to the XIOSim implementation. Measurements were performed on a real system with a dual-core Intel Atom 330 processor, with only one core active. Hyperthreading was also disabled during all experiments.

Since XIOSim focuses on detailed modeling of the core microarchitecture, the benchmark suite we use is the CPU-centric SPEC CPU2006 [1]. The sampled slices are collected using PinPoints [12], with a slice length of 100M instructions. Over the whole benchmark suite, this results in ≈500 slices, totalling ≈50B simulated instructions.

5.1 Performance

The in-order performance model is reasonably accurate. This is demonstrated in Figure 6. It shows measured versus simulated data for overall IPC, as well as for the major microarchitectural events. The dashed unit lines in each subplot represent the ideal case, and each circle marker is measured from a single 100M instruction execution slice. In an ideal, cycle-accurate model, all markers should be clustered along the unit lines.

Figure 6a shows that the performance model tracks end-to-end performance with satisfactory accuracy – the geometric mean of the IPC error over all slices is 11.02%. Furthermore, if we take into account the relative importance of each slice as computed by SimPoints, the mean IPC error decreases to 7.67%. Looking at the distribution of markers in Figure 6a, the simulator is not systematically underestimating or overestimating end performance.

The relative accuracy in predicting individual microarchitectural events is lower, as seen by Figures 6b-6f. For example, the average error in instruction cache miss rates (Figure 6e) is as high as 74%. The explanation of this discrepancy comes from the absolute values of these events – they are often small numbers. In that case, the error between a simulated miss rate of 0.1% and a measured rate of 0.2% is 50%, but the end impact on performance is insignificant because both rates are sufficiently small. This is especially true for miss rates in the instruction cache and the TLB, which are, on average 0.3% and 0.7%.

While most of the event errors also appear unbiased to-

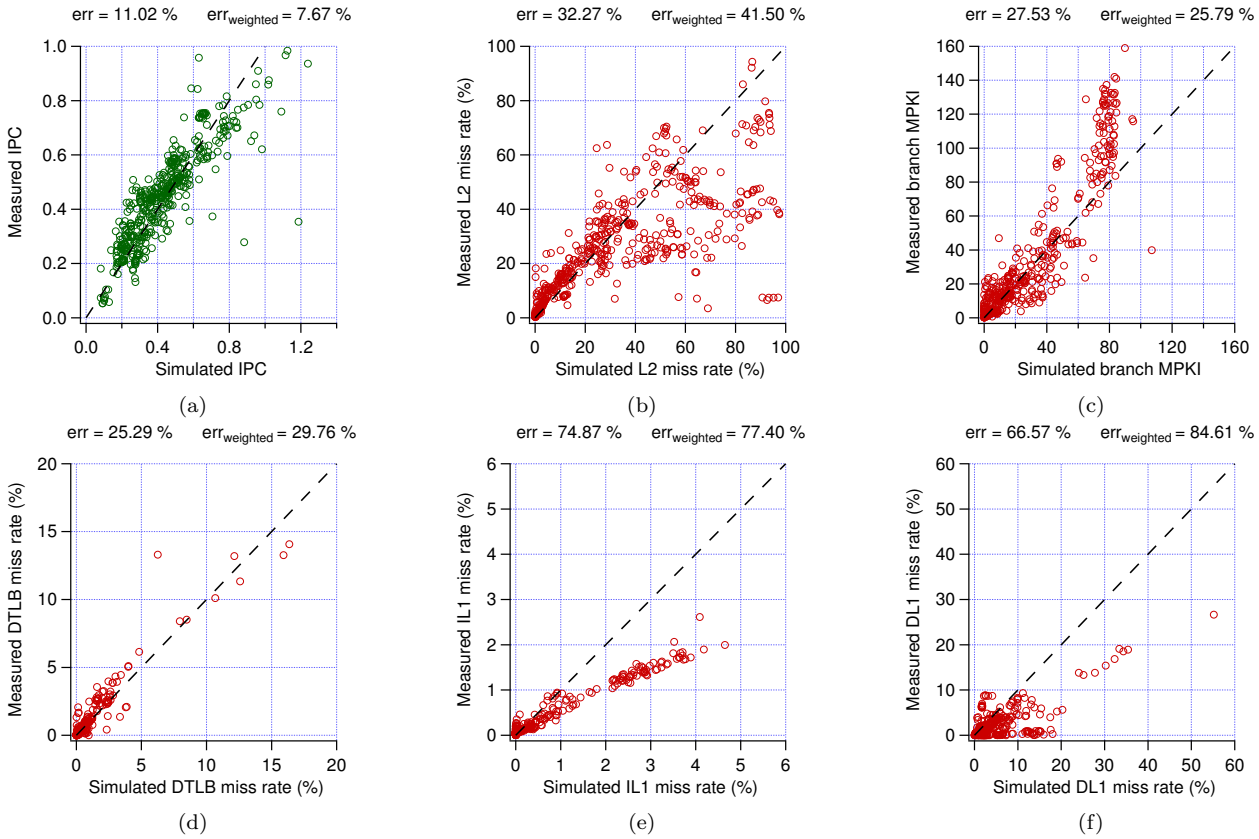


Figure 6: Comparison of microarchitectural statistics between performance counter measurements and simulation for execution slices of the SPEC CPU2006 suite.

ward under-counting or over-counting, there are two distinct clusters of slices in Figure 6c and Figure 6e, where the simulator underestimates branch mispredictions and overestimates instruction cache misses. Most of those slices can be traced to the benchmark *445.gobmk*, which has been shown to have high branch misprediction rates [6]. On a real machine, the higher number of branch misses can have a prefetcher-like effect explaining the lower miss rates in the instruction cache.

5.2 Power

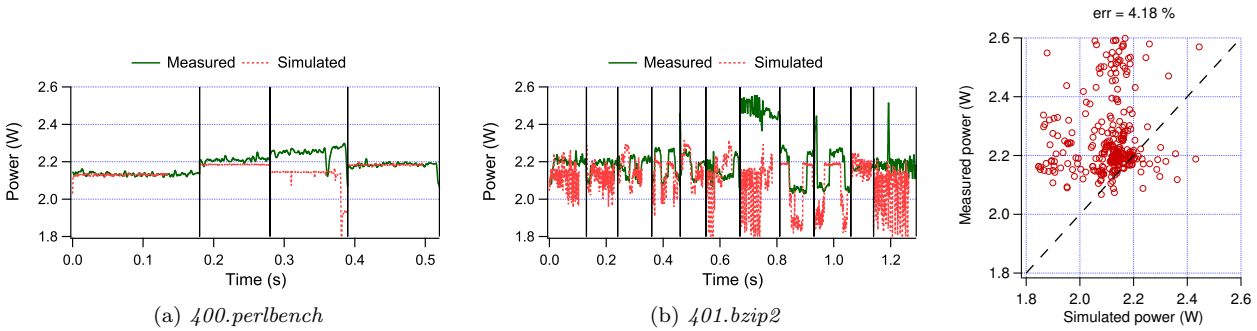
It is reasonable to expect that the power model is less accurate than the performance model. We first show that it can qualitatively track benchmark behavior on a real machine. We then go on to validate the quantitative predictions for power consumption, starting with leakage estimates, and continuing with the sum of static and dynamic power.

Workload tracking. The methodology described in Section 4.2 allows us to demonstrate that the power model can adequately track workload-related activity. Figure 7 shows that by comparing measured versus simulated power consumption for the benchmarks *400.perlbench* and *401.bzip2*. Vertical lines denote execution slice boundaries. Even though the two benchmarks show qualitatively very different power profiles, XIOSim's power model tracks the shape of both relatively well. As with any model, the synchronization is not perfect – for example, notice how the first simulated slice of *400.perlbench* finishes significantly before its corresponding measured slice. In this particular instance, the misalignment comes from a large IPC error in the performance model, which predicts execution time wrongly.

Leakage. In order to quantify the model accuracy, we start by estimating the leakage of our real machine with a simple experiment. While our test processor is executing an instance of the power virus, we scale the frequency from the maximum supported 1600 MHz to the minimum 200 MHz at 200 MHz increments, measuring average power consumption at each step. Since we are only scaling frequency and not voltage, power consumption scales linearly, and we can estimate the static power as the portion independent of frequency in a linear fit.

Performing this experiment, we estimated that the Atom 330 consumes 1.79 W static power. Our power model leakage estimate is 1.4 W. However, it does not include any overheads associated with simultaneous multi-threading (SMT). Even though we keep SMT disabled through our experiments, it still incurs a leakage cost because it requires additional queues and buffers, or additional access ports to existing structures. Intel data [5] suggests that the power cost of adding SMT to an Atom core is 15%. Adding this overhead to our leakage estimates results in 1.61 W in static power, which is within 10.1% of the measured value. It is logical that the model underestimates static power – a real design includes components that are not explicitly modelled, such as PLLs, bus drivers and memory buses, which could consume a non-ignorable amount of energy.

Total power. After incorporating the leakage correction



(a) *400.perlbench* (b) *401.bzip2* (c) Aggregate for CPU2006

Figure 7: Comparing measured and simulated power consumption for execution slices of SPEC CPU2006.

for SMT, we can look at total processor power across benchmarks. We collapse the dynamic behavior of each execution slice to a single average power value and compare measured and predicted per-slice average power in a manner similar to Section 5.1. Figure 7c shows the results of this validation procedure. The geometric mean power error is only 4.18% (after accounting for SMT). However, as seen in Figure 7c, a large fraction of execution slices cluster around an average power consumption of 2.2 W. We attribute this low variability to the fact that the core under test is in-order and relatively simple.

From Figure 7c, we can clearly notice that the simulation systematically underestimates power consumption. As discussed in the previous paragraph, this is expected behavior, since we do not model all components on a real system. There is a distinctive set of slices with measured power between 2.4 and 2.6 W which correlate very poorly with simulation results. One such example is the 7-th slice in Figure 7b. We have not been able to identify the reason for the discrepancy, but possible candidates are the operating system timer triggering on the real system, or the voltage regulator triggering a feedback loop.

6. CONCLUSION

With the growth of the mobile sector, simple power-efficient cores are becoming increasingly more relevant, even with instruction sets as IA-32 that have been traditionally reserved for high-performance, high-power domains. We have presented XIOSim – a simulation tool that is able to model the power and performance of such cores. In order to validate its correctness, we have developed a rigorous methodology and an experimental setup that allows comparing the same execution slices on a real machine and in simulation.

This paper explicitly focuses on single-core modelling and validation, leaving the multi-core case for future work. In the near future, we are planning to extend our simulation methodology and infrastructure to be able to handle heterogeneous multi-core configurations, keeping true to a rigorous validation approach.

Acknowledgements

This work was partially supported under SRC contracts with task ID 1814 and 1973, as well as partially supported by National Science Foundation grants CCF-0903437, CSR-0720566 and CCF-0702344. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and do not necessarily reflect the views of the SRC or NSF.

7. REFERENCES

[1] SPEC CPU2006, http://www.spec.org/cpu2006/.
[2] N. Binkert et al. The GEM5 simulator. *ACM SIGARCH Computer Architecture News*, 2011.
[3] D. Burger and T. Austin. The SimpleScalar tool set, v. 2.0. *ACM SIGARCH Computer Architecture News*, 1997.
[4] S. Campanoni et al. A highly flexible, parallel virtual machine: design and experience of ILDJIT. *Softw., Pract. Exper.*
[5] G. Gerosa et al. A Sub-1W to 2W Low-Power IA Processor for Mobile Internet Devices and Ultra-Mobile PCs in 45nm Hi- κ Metal Gate CMOS. In *2008 IEEE International Solid-State Circuits Conference.*
[6] A. Kejariwal et al. Comparative architectural characterization of SPEC CPU2000 and CPU2006 benchmarks on the intel Core2 Duo processor. *2008 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation.*
[7] S. Li et al. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *42nd International Symposium on Microarchitecture (MICRO-42)*, 2010.
[8] G. H. Loh and S. Subramaniam. Zesto: A Cycle-level Simulator for Highly Detailed Microarchitecture Exploration. *2009 IEEE International Symposium on Performance Analysis of Systems and Software.*
[9] C.-K. Luk et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI '05: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation.*
[10] H. McGahn. Niagara 2 Opens the Floodgates. *Microprocessor Report*, 2006.
[11] A. Patel et al. MARSS: A full system simulator for multicore x86 CPUs. In *Design Automation Conference (DAC)*, 2011.
[12] H. Patil et al. Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation. In *37th International Symposium on Microarchitecture (MICRO-37)*, 2004.
[13] T. Sherwood et al. Automatically characterizing large scale program behavior. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.
[14] S. Vangal et al. An 80-tile Sub-100-W Teraflops processor in 65nm CMOS. *IEEE Journal of Solid-State Circuits*, 2008.
[15] D. Wentzlaff et al. On-chip Interconnection Architecture of the Tile Processor. *IEEE Micro*, 2007.
[16] M. Yourst. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *International Symposium on Performance Analysis of Systems & Software*, 2007.