

Circuit-Aware Architectural Simulation

Seokwoo Lee, Shidhartha Das, Valeria Bertacco, Todd Austin
David Blaauw, and Trevor Mudge
Advanced Computer Architecture Lab
The University of Michigan
1301 Beal Ave, Ann Arbor, MI 48109
razor@eecs.umich.edu

ABSTRACT

Architectural simulation has achieved a prominent role in the system design cycle by providing designers the ability to quickly examine a wide variety of design choices. However, the recent trend in system design toward architectures that react to circuit-level phenomena has outstripped the capabilities of traditional cycle-based architectural simulators. In this paper, we present an architectural simulator design that incorporates a circuit modeling capability, permitting architectural-level simulations that react to circuit characteristics (such as latency, energy, or current draw) on a cycle-by-cycle basis. While these additional capabilities slow simulation speed, we show that the careful application of circuit simulation optimizations and simulation sampling techniques permit high levels of detail with sufficient speed to examine entire workloads.

Categories and Subject Descriptors

B.8.2 [Performance Analysis and Design Aids]: Architectural Simulation; B.5.2 [Register-Transfer-Level Implementation]: Design Aids—*Simulation*

General Terms

Computer system simulation

Keywords

Architectural simulation, High-performance simulation, Circuit simulation

1. INTRODUCTION

To accelerate the hardware design cycle, architects often employ architectural simulators of the hardware they are designing. They implement these models in traditional programming languages or hardware description languages, and then execute programs on them to validate the performance and correctness of a proposed hardware design. Although software models run slower than their hardware counterparts, architects can construct hardware models in minutes

or hours rather than in the months needed to manufacture real hardware. The faster build time speeds up the hardware design cycle, giving architects the ability to evaluate more designs before committing to a single solution for fabrication.

In the traditional approach to architectural simulation, a software model of the architecture is constructed by first identifying the major components of the system and determining their operation latency as a function of the expected clock cycle of the machine. For example, ALUs and register files typically have a latency of one cycle, while the latency of the caches are dependent on the size of the cache. With the latency of components defined, an architectural model of the complete machine is constructed by quantifying the number of each component, their datapath connections within the microarchitecture, and the hazards (or stalls) that can be experienced by instructions using the various components in the design. Once the model is defined, architectural simulation becomes the process of determining the total number of cycle it takes to execute a program, based on instructions executed, availability of resources, and the hazards experienced during execution.

There is a recent trend in computer architecture design toward systems that can adapt to circuit-level phenomena. In these highly adaptable systems, it is possible for the architecture to influence circuit operation and vice versa. Examples of these type of systems include di/dt [6] and thermal throttling [14] and Razor clocking [7]. Throttling techniques monitor current and temperature characteristics of the underlying circuit implementation. If current demands get too high (which induces noise on the supply lines) or if temperatures rise too high, an architectural-level system controller will be invoked to throttle down instruction fetch bandwidth. With fewer instruction entering the microarchitecture, current demands and temperature are quickly reduced. Razor clocking is a technique to reduce circuit energy levels below the point required by worst case computation paths [7]. In the event a computation fails due to extraordinary energy requirements, an error recovery mechanism restores correct state. With prudent energy tuning, the approach can greatly reduce circuit energy demands with little impact on computational speed.

These novel circuit-aware architectural optimizations share the requirement that the architectural simulator must accurately gauge detailed circuit phenomena to correctly simulate the operation of the machine under study. For example, the throttling techniques must count the total number of devices switching during each cycle of operation, and simulation of Razor clocking requires detail timing information of pipeline stages on a per-cycle basis. The approach that

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Design Automation Conference 2004

Copyright 2004 ACM 0-12345-67-8/90/01 ...\$5.00.

has been taken to analyze much of this work has been to utilize extremely simplistic analytical circuit models of microarchitectural components. The primary advantage of the analytical circuit models is flexibility and speed. They also have minimal performance impact with typically less than a 100% slowdown. However, researchers have begun to question the accuracy of simple analytical circuit models [8].

In this paper, we present an architectural simulation modeling infrastructure that incorporates circuit simulation capabilities. The approach is quite accurate because we analyze detailed circuit-level phenomena including individual gate delay and energy characteristics. Performance, while considerably slower than architectural simulation, is maintained using an effective combination of circuit and architecture level simulation optimizations. The optimization we implement include *i*) early circuit simulation termination based on architectural constraints, *ii*) circuit timing memoization, and *iii*) fine-grained instruction sampling. Using our optimized circuit-aware architectural simulator, we are able to examine the performance of a large program in detail in under 5 hours of simulation.

The remainder of this paper is organized as follows. Section 2 details related work in architectural simulation, circuit simulation, and simulator performance optimization. Section 3 details our circuit simulation methodology and its integration into an architectural simulation model. Section 4 describes the optimizations we implemented to further improve the performance the simulator. Section 5 demonstrates the use of our simulator with a case study of Razor clocking. Finally, Section 6 summarizes the paper and suggests future directions.

2. BACKGROUND AND RELATED WORK

A number of popular architectural simulation infrastructures exist that are widely used in academia and industry. One of the most notable examples is the SimpleScalar tool set [2], a collection of simulation models capable of running programs compiled for the PISA, Alpha and ARM instruction sets. At the core of the simulator infrastructure is an emulation mechanism to execute programs of interest. They also include event management routines, resource tracking mechanisms, and statistical analysis packages. The resulting models are at a high enough abstraction level that they execute fairly efficiently. The most detailed SimpleScalar models execute programs at a rate of about 100,000 instruction per second (IPS), permitting architects to examine seconds of real-time execute (billions of instructions) in a few hours of simulation.

The approach that has been taken to incorporate circuit characteristics (such as power demands) into microarchitectural simulators has been to utilize analytical circuit models of components and drive them with the events in the architectural simulator. In the Wattch power analyzer [3], circuit level power behavior was characterized with analytical models, including those developed for the CACTI on-chip cache model [17]. The microarchitectural simulator records the switching characteristics of the components such as caches, register files, branch target buffers, and translation look-aside buffers. The dynamic power consumption can then be calculated directly.

A similar approach was adopted by SimplePower [10]. It incorporated register transfer level (RTL) power models based on look-up tables (LUT) into a microarchitectural

simulator. Finally, the Cai-Lim power-performance simulator introduced empirical power density models derived from Intel internal design data for major microarchitectural functional blocks [4]. In all of these tools, microprocessor power is estimated by multiplying the frequency of the accesses to the microarchitectural functional blocks with their lumped capacitance. This in turn is derived from the circuit model of the block.

The primary advantage of these analytical circuit models is flexibility and speed. The models are sufficiently high level that they can be readily reconfigured to different component configurations (*e.g.*, cache size, branch predictor configuration) and fabrication technologies. They also have minimal simulation performance impact with less than a 100% slowdown for all of the models discussed. Recently, however, the accuracy of analytical models in architectural models has come into question. In a paper by Ghiasi [8] it was noted that even at a 90% confidence level, two analytical-based power models (Wattch and Cai-Lim) failed to agree on the benefits of a variety of power-based optimizations. The results of the two models were uncorrelated, suggesting that at least one of the tools was based on incomplete or inaccurate models. The authors concluded that the accuracy of circuit-level models must be improved to detect anything but the grossest power savings. In our previous Razor work (by Ernst *et al.* [7]), we further refined this approach to architectural modeling to allow direct measurement of module latency, based on input vectors from a live architectural simulation. Our model was built by hand, thus limiting our experimental studies to examining the effects of Razor clocking on a single Kogge-Stone adder circuit [7].

Ideally, for circuit-aware architectural designs, we would like to leverage an analysis framework with the accuracy of circuit-level simulation and the flexibility and speed of architectural-level simulation. This is not possible with state-of-the-art circuit simulation tools alone. When running the Razor clocking experiment (detailed in Section 5.2) on Synopsys VCS, a compile-based Verilog simulator which we configured to use SDF back-annotation timing information, simulations ran at rates of about 50 instructions per second on a Sun Blade 1000 workstation. Typical architectural simulations examine up to 1 billion instructions, which would require at least six months of simulation! In addition, the framework was not sufficiently flexible to accomplish the necessary analysis, *e.g.*, the tool could not accommodate voltage changes during simulation as all logic and wires were characterized as voltage-derived delays bound in the simulation code. Hence, this particular tool is not sufficiently flexible to examine dynamic voltage scaling (DVS), an optimization of intense interest in the architecture literature.

In the domain of circuit-level simulation, SPICE has been the industry standard for the past 25 years. During this time, much research [1, 11] has been devoted to improve the simulation performance to handle the increasing complexity of circuit designs without sacrificing simulation accuracy. In this context, SPECS2 [15] marked a change of pace by proposing one of the first table-based approaches to timing simulation, which is also the technique of choice for our simulator and for many of the current tools in this arena. Today, state-of-the-art commercial simulators can simulate designs of millions of transistors at a speed a thousand times faster than SPICE with an accuracy trade-off of just a few percent. However, architectural designs require an analysis

based on the simulation of billions of instructions, which is still far beyond the reach of these simulators. To this end, we developed our simulator to exploit much more aggressive approximations of circuit behavior in order to achieve the performance required.

3. SIMULATION METHODOLOGY

3.1 Architectural Simulation

Figure 1 illustrates the software architecture of our circuit-aware architectural simulator. The simulator model is based on the SimpleScalar modeling infrastructure [2]. The SimpleScalar tool set is capable of modeling a variety of platforms ranging from simple unpipelined processors to detailed dynamically scheduled microarchitectures with multiple-level memory hierarchies. The toolset supports several instruction sets, including PISA, Alpha, and ARM.

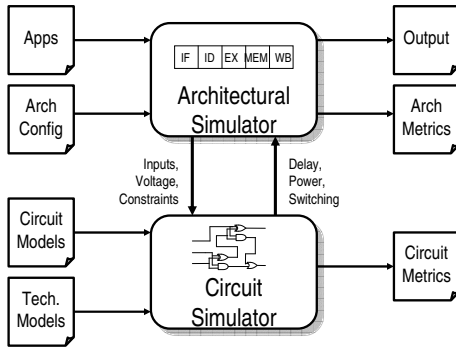


Figure 1: Simulator Software Architecture

The architectural simulator takes two primary inputs: a configuration file that defines the microarchitecture being modeled, and a program to execute. The microarchitecture configuration defines the stages of the pipeline, plus any special units that reside in those stages, such as branch predictors, caches, functional units, and bus interfaces. The microarchitecture configuration used in the experiments in this paper are detailed in Section 5.2. All programs analyzed are compiled for the Alpha instruction set.

The architectural simulator produces two primary outputs. If the program executes any I/O operations (e.g., file accesses or console writes), the I/O operations are executed by the simulator on behalf of the simulated program. In addition, SimpleScalar provides an extensive instrumentation capability, such that operations exercised during simulation can be monitored to produce runtime metrics, such as instructions per cycle (IPC), average memory latency (MLAT), or branch predictor accuracy. The metrics output at the end of simulation are used to evaluate the quality of the microarchitecture configuration, with respect to the program that was executed on it.

3.2 Circuit Simulation

To support circuit-awareness in the architectural simulator, we embedded a circuit simulator (implemented in C++) within our SimpleScalar models. The embedded circuit simulator references a combinational logic description of each

relevant component of the architecture under evaluation, and interfaces with the architectural simulator on a stage-by-stage basis. At initialization, the circuit description of the various components is loaded from a structural Verilog netlist. Concurrently, the interconnected wire capacitance is loaded from files provided by global routing and placement tools. In addition, a technology model is loaded that details the switching characteristics of the standard cell blocks used in the physical implementation.

During each simulation cycle, each logic block is fed a new input vector from the architectural simulator. The vectors correspond to the set of values latched at each pipeline stage input. With this information, the circuit simulator can compute the relevant measures for the analysis under study: delay of the computation, total energy dissipated, and additional switching characteristics such as total current draw. Depending on the purpose of the simulation, these measures are returned to the architectural simulator to direct the high level progress of the simulation and/or returned as output for evaluation. The circuit simulator has enough accuracy to operate as a standalone circuit analysis tool, capable of transient fault injection experiments, and of investigating process variation.

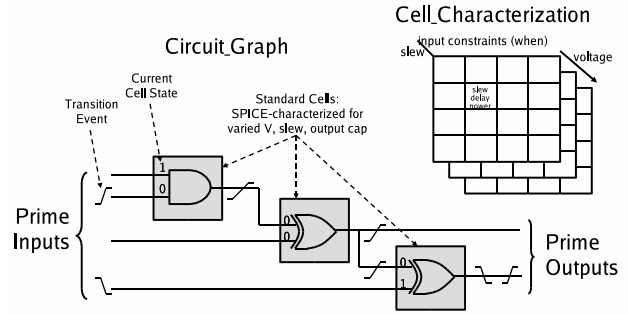


Figure 2: Circuit Simulation Methodology

The implementation of the circuit simulator is based on an event-driven scheme. Each activity in the logic circuit is represented as a transition which is composed of final logic value, arrival time and slew (slope of the voltage transition). Figure 2 provides a schematic description of the circuit model used by the simulator. At each simulation cycle, the first set of transitions are generated by comparing the new set of input values with those of the previous cycle. Within each set, input transitions all have the same arrival time and a fixed slew. When there is a transition on a cell input line, the function of the cell is evaluated and a new transition may or may not be generated at the cell output, depending on the logic function of the cell. Newly generated transitions are added to an event queue sorted by ascending arrival time. We found experimentally that at each simulation cycle only a small fraction, between 2 and 10%, of the circuit's cells need to be evaluated, making our simulation approach comparable to state-of-the-art commercial logic simulators. In order to achieve the performance required by the architectural simulator, we represent voltages as a 0 or 1 logic values. The sorted event queue allows us to easily cancel very close proximity glitches, by eliminating a pair of opposite transitions which happen within an interval shorter than the full transition time. The pseudo-code in

Figure 3 outlines the simulation algorithm just described.

```

Circuit::Simulate( vector Input ) {
    power = 0;
    for each (changedBit in Input) {
        Tr = GenInputTransition(changedBit);
        EventQueue.InsertSorted(Tr);
    }
    while (!EventQueue.empty()) {
        Tr = EventQueue.RemoveEarliest();
        for each (cell, Tr.net ∈ cell.Inputs) {
            power += cell.Evaluate(Tr);
        }
    }
}

```

Figure 3: Circuit simulation algorithm

At configuration time, we compute a characterization table for each cell. These tables, illustrated in Figure 2, support the computation of the output transitions during simulation. Each characterization table is parameterized based on the supply voltage (V_{dd}) of the system, input transition slew and input transition constraints (0 to 1, or 1 to 0), and they provide the slew and delay of the transition on the output net and the power consumption of the cell for the event. To generate a characterization table, we need to first compute the output capacitance of the cell, which we obtain by combining the two capacitive components due to the wire lengths, provided by the place and route software and the cell's fanout. Fanout capacitances are obtained by adding the capacitances due to each of the cell's fanout elements, which we find in the SPICE models of the library cells. Once we have computed the cell's output capacitance we can derive the cell's own characterization table from the technology model file which provides delay and power data based on capacitance, voltage and type of transition. The table is then used during simulation to compute the relevant measures of each transition as outlined in Figure 4. To evaluate the accuracy of our approach, we compared our circuit simulator against a set of SPICE simulations with a number of test circuit topologies at varied voltages and input slew rates and found timing errors rates to be consistently less than 11%, with most less than 3%.

```

Cell::Evaluate ( transition Tr ) {
    newOut = logic_function(Tr);
    if (newOut != output) {
        Tr_out = new();
        Tr_out.value = newOut;
        Tr_out.slew = Slew(vdd, Tr.slew, newOut);
        Tr_out.delay = Tr.delay +
            CellDelay(vdd, Tr.slew, newOut);
        EventQueue.InsertSorted(Tr_out);
    };
    return Power(vdd, Tr.slew, newOut);
}

```

Figure 4: Cell evaluation during simulation

4. PERFORMANCE OPTIMIZATIONS

4.1 Constraint-Based Circuit Pruning

Often architectural analyses require running simulations that collect only one relevant measure from the circuit sim-

ulator, and such measure is only relevant if it is above a specified threshold. Both throttling techniques and Razor clocking fall into this category. For instance, in architectures with Razor clocking, circuit simulation needs to evaluate if the propagation delays are above or below the clock cycle time, while the actual value of the propagation delay is not relevant. In general, domain-specific static analyses can be used to compute worst-case values of the constraint measure and the simulation can be pruned to eliminate the portion of the circuit netlist where the constraint cannot be violated.

In the specific case of Razor designs, all those paths in the logic network that are guaranteed to have propagation delays below the specified clock threshold can be removed since these paths can not affect the architectural simulation. Using this approach, we can achieve higher performance, without affecting the accuracy of results.

To compute which cells can be removed from the logic network, we compute the worst-case delay from the inputs to each output. The propagation delay through a cell is found on a per-voltage basis from the characterization tables. Once we know the worst-case delay at each output, we can remove from the netlist the outputs that are guaranteed to have stabilized by the specified cycle time, and all the cells in the cone of logic that only contributes to those outputs. Every time that the voltage is changed in the circuit simulation, we recalculate the constraint-based pruning before continuing simulation. This optimization has proven to be very effective, particularly in those circuit blocks that are highly control-driven.

Another optimization derived from the one above, involves maintaining a maximum satisfying budget for the parameter at each internal node of the netlist. Whenever the simulation reach that node, if the parameter under evaluation has a lower value than the satisfying limit allows, we are again guaranteed that the simulation will reach the output nets without violating the threshold. In this case we can proceed from those nodes to the outputs switching to a pure logic simulation, without computing the other simulation measures.

Pruning is in general quite effective, but performance does depend on the tightness of the constraint (*i.e.*, voltage level for a Razor design). In our case study at 1.8V nominal voltage at 200 MHz design, pruning eliminated 64% of the circuit. At a more highly constrained voltage of 1.4V, 24% of the circuit is eliminated. Even at 1.4V, simulation speed is nearly doubled with pruning.

4.2 Circuit Timing Memoization

Locality is a key principle in the design of modern microarchitectures. The principle states that program instructions and data recently used are more likely than random values to be used in the future. Consequently, devices such as caches and value predictors [9] can accurately anticipate the requirements of a program based on past activities.

We can leverage value locality to improve the performance of circuit timing simulation. We construct a hash table that records (a.k.a. memoizes) the following mapping for each circuit-level module:

$$(vector_{state}, vector_{in}, V_{dd}) \rightarrow (delay, energy)$$

Where $vector_{state}$ represents the current state of the circuit, $vector_{in}$ is the current input vector, and V_{dd} is the current

operating voltage. The hash table returns the circuit evaluation latency and the circuit evaluation energy. We index the hash table with a combination of $vector_{state}$ and $vector_{in}$ because $vector_{state}$ encode the current state of the circuit and $vector_{in}$ indicates the input transitions. Combined with the current operating voltage, V_{dd} , the inputs to the hash table fully encodes the factors that determine delay and energy.

Whenever the hash table does not include the requested entry, full-scale circuit simulation is performed to compute the delay and energy of the circuit computation. The result is then inserted into the hash table with the expectation that later portions of the program will generate similar vectors. In our implementation, the size of the hash table is limited to 256 MB. In addition, we found better performance when we dynamically re-order the hash bucket chains, by bringing the most recently referenced element to the head of the chain. The latter optimization further exploits fine-grained program value locality.

For our baseline hash table implementation, we achieved a hit rate of typically less than 50%, which still rendered a sizeable speedup. Investigation into the access stream quickly revealed that hashing the entire input vector to pipeline stage logic is overly restrictive. For example, load instructions that pass through the execute (EX) stage of the pipeline include two input register operands in their input vectors, yet, the second operand is ignored during execution of the load (instead the instruction offset field is used). By including the second operand in the input vector, multiple hash table entries are required to memoize the same load address computation. To alleviate this problem, a per-opcode input vector filtering mechanism was developed. Each instruction opcode indicates with a mask which inputs do not influence stage logic evaluation. These inputs are masked off before attempting to memoize the circuit simulation. The optimization resulted in a much higher hash table hit rate of 70-85% on average. Simulation speedups due to memoization were quite noticeable, with most experiments experiencing 3-5x improvements.

4.3 SimPoint Analysis

Typical architectural simulations in the literature analyze dynamic program lengths of 1 billion or more instructions. Even after deploying all of the previous optimizations, we will only reach simulation speeds of the order of a 1000 instructions per second, which would require more than a week of simulation time to complete a single program run.

Fortunately, we can draw on a recent result in computer simulation sampling to relax the performance demands for the circuit-aware architectural simulator. SimPoint analysis was recently proposed as a technique to dramatically reduce the number of instructions simulated to characterize a program's performance on a complex microarchitecture [13]. SimPoint uses basic block distribution analysis along with several techniques from clustering analysis to concisely summarize the behavior of an arbitrary section of execution in a program. This information summarizes whole program behavior and greatly reduces simulation time by using only representative samples.

In our work, we use 10 million instruction length samples (called Early Multiple SimPoints) [12]. The SimPoints indicate a collection of sample starting points to simulate within the program, the length of the samples, and the weight to use when combining simulation statistics (*e.g.*, IPC). With this

technique, even our slowest simulation was able to analyze a complete program in just over 5 hours (at 554 instructions/second). Error analysis of these SimPoints indicate an error of less than 10% (typically less than 3%) for a wide variety of benchmarks [5].

5. EXAMPLE CASE STUDY

To evaluate the quality our circuit-aware architectural simulator, we modeled the Razor clocking technique proposed by Ernst *et. al.* [7]. In Razor designs, the latency of an instruction (in cycles) may vary based on the latency of the circuit evaluation within a pipeline stage. In this section, we present a high level overview of the Razor clocking technology, and demonstrate its evaluation using our circuit-aware architectural simulator.

5.1 Razor Timing Speculation

The key observation underlying the design of Razor is that the worst-case conditions that drive traditional design are improbable conditions. Thus, by building error detection and correction mechanisms into the Razor design, it becomes possible to tune voltage to typical energy requirements, rather than worst case. The resulting design has significantly lower energy requirements, even in the presence of added energy processing demands due to occasional error recoveries. The Razor design utilizes an in-situ timing error detection and correction mechanism implemented within the Razor flip-flop. Razor flip-flops double-sample pipeline stage values, once with an aggressive fast clock and again with a delayed clock that guarantees a reliable second sample. A metastability-tolerant error detection circuit is employed to check the validity of all values latched on the fast Razor clock. In the event of a timing error, a modified pipeline flush mechanism restores the correct stage value into the pipeline, flushes earlier instructions, and restarts the next instruction after the erroneous computation. For additional background on Razor timing verification and related dynamic verification work in general, see references [7, 16].

5.2 Experimental Framework

To model Razor clocking, we implemented an architectural model of a baseline 64-bit Alpha processor model. The processor architecture is a simple in-order pipeline consisting of instruction fetch, instruction decode, execute, and memory/writeback with 8 Kbytes of I-cache and D-cache. In addition, the entire processor was described in Verilog and synthesized using Synopsys Design Analyzer (version 2003.03-2). Global routing capacitances were estimated by performing global place and route using Cadence Silicon Ensemble (version 5.4.126) and Mentor Graphics Xcalibre (version 9.1.5.6). The processor was mapped to a 0.18um TSMC process, and it was validated to operate at 200 MHz. After careful performance analysis, it was found that only the instruction decode and execute stages were critical at the worst-case voltage and frequency settings; hence, only these stages are incorporated into the circuit-aware architectural simulations.

5.3 Simulation Case Study

Table 1 shows that the baseline performance of our circuit-aware simulator is comparable to the compile-based Verilog VCS simulator, which simulated the Razor design at about

50 instructions per second. However, after applying all optimizations, it reaches a speed of 887 instructions per second, more than 8 times faster than its barebone counterpart.

Optimization options	instructions/sec
None	102
Pruning	347
Pruning and Memoization	887

Table 1: Benefits of Circuit Simulation Optimizations (when simulating GCC)

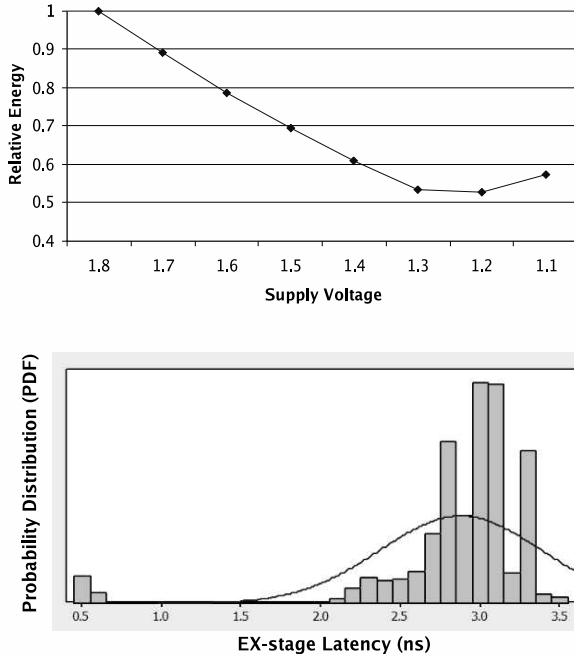


Figure 5: Case Study of a Razor Design

Figure 5 demonstrates the performance of Razor clocking as measured by our circuit-aware architectural simulator. The top graph shows the relative energy of the pipeline with decreasing voltage. As voltage decreases, simulated pipeline energy decreases, even in the presence of expensive timing error recoveries. Because our circuit-aware architectural simulator can accurately gauge the per-cycle stage evaluation latency, it is possible to assess the voltage (around 1.2V) at which the cost of Razor timing error recovery outweighs the benefits of further decreasing voltage. In addition, the bottom graph demonstrates the measurement capability of our circuit simulator. The figure illustrates the latency, through the EX stage logic, as a probability distribution function for all input vectors produced during a simulation of the GNU GCC compiler. Given that the worst-case latency through this stage is over 4ns, it is clear that typical case latencies are much less, allowing Razor to lower voltage with only small increases in circuit timing error rates.

6. CONCLUSIONS

In this paper we have shown that it is possible to combine circuit simulation with an architectural simulator and still achieve significant simulation throughput rates. By identifying those events that repeat or are not critical we can still capture delay information that is voltage or data dependent in the simulation. In the past, this sort of analysis required

a clumsy coupling of architectural simulation and selective SPICE simulation. Our framework provides not only an automated solution to the specific problem of voltage and data dependent delays, but it can be extended in a natural way to other run-time dependencies, such as process variation and noise coupling.

Acknowledgements

This work is supported by grants from ARM Ltd., the National Science Foundation, and the Gigascale Systems Research Center.

7. REFERENCES

- [1] E. Acuna, J. Dervenis, A. Pagonis, and R. Saleh. iSPLICE3: a new simulator for mixed analog/digital circuits. In *IEEE Custom Integrated Circuits Conference*, pages 13.1/1–13.1/4, May 1989.
- [2] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. In *IEEE Computer*, Feb. 2002.
- [3] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proc. 27th Int. Symp. on Computer Architecture (ISCA27)*, May 2000.
- [4] G. Cai and C. H. Lim. Architectural level power/performance optimization and dynamic power estimation. In *Cool Chips Tutorial in conjunction with the 32nd Int. Symp. on Microarchitecture (MICRO-32)*, Nov. 1999.
- [5] B. Calder. Simpoint website. In <http://www.cse.ucsd.edu/calder/simpoint/>, 2003.
- [6] W.-K. Chen. The VLSI handbook. In *CRC Press publisher*, 2000.
- [7] D. Ernst, N. S. Kim, S. Das, S. Pant, T. Pham, R. Rao, C. Ziesler, D. Blaauw, T. Austin, T. Mudge, and K. Flautner. Razor: A low-power pipeline based on circuit-level timing speculation. In *36th Annual International Symposium on Microarchitecture (MICRO-36)*, Dec. 2003.
- [8] S. Ghiasi and D. Grunwald. A comparison of two architectural power models. In *Workshop on Power Aware Computing Systems (PACS-2000)*, Dec. 2000.
- [9] M. H. Lipasti and J. P. Shen. Exploiting value locality to exceed the dataflow limit. In *29th International Symposium on Microarchitecture (MICRO-29)*, Dec. 1996.
- [10] N. Vijaykrishnan et al. Energy-driven integrated hardware-software optimizations using SimplePower. In *Proc. 27th Int. Symp. on Computer Architecture (ISCA27)*, May 2000.
- [11] C. Ratzlaff, N. Gopal, and L. Pillage. RICE: Rapid interconnect circuit evaluator. In *DAC, Proceedings of Design Automation Conference*, pages 555–560, June 1991.
- [12] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2001.
- [13] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.
- [14] K. Skadron, M. Stan, and T. Abdelzaher. Control-theoretic techniques and thermal-RC modeling for accurate and localized dynamic thermal management. In *8th International Symposium on High-Performance Computer Architecture (HPCA-8)*, Feb. 2002.
- [15] C. Visweswariah and R. Rohrer. SPECS2: An integrated circuit timing simulator. In *ICCAD, Proceedings of the International Conference on Computer Aided Design*, pages 94–97, Nov. 1987.
- [16] C. Weaver and T. Austin. A fault tolerant approach to microprocessor design. In *IEEE International Conference on Dependable Systems and Networks (DSN-2001)*, June 2001.
- [17] S. Wilton and N. Jouppi. An enhanced access and cycle time model for on-chip caches. In *Western Research Laboratory Research Report 93/5*, July 1993.