Proc. Asia South Pacific Design Automation Conf. (ASP-DAC), Shanghai, China, vol. 1, Jan. 2005, pp. I/2-I/7.

Opportunities and Challenges for Better Than Worst-Case Design

Todd Austin, Valeria Bertacco, David Blaauw, and Trevor Mudge

Advanced Computer Architecture Lab
The University of Michigan
razor@eecs.umich.edu

ABSTRACT

The progressive trend of fabrication technologies towards the nanometer regime has created a number of new physical design challenges for computer architects. Design complexity, uncertainty in environmental and fabrication conditions, and single-event upsets all conspire to compromise system correctness and reliability. Recently, researchers have begun to advocate a new design strategy, called *Better Than Worst-Case* design, that couples a complex core component with a simple reliable checker mechanism. By delegating the responsibility for correctness and reliability of the design to the checker, it becomes possible to build correct-certified designs that effectively address the challenges of deep submicron design.

In this paper, we present the concepts of Better Than Worst-Case design and highlight two exemplary designs: the DIVA checker and Razor logic. We show how this approach to system implementation relaxes design constraints on core components, which reduces the effects of physical design challenges and creates opportunities to optimize performance and power characteristics. We demonstrate the advantages of relaxed design constraints for the core components by applying *typical-case optimization (TCO)* techniques to an adder circuit. By analyzing the carry-propagation characteristics of real programs, it is possible to design an adder circuit that when incorporated into a Better Than Worst-Case design exhibits significantly reduced latency. Finally, we discuss the challenges and opportunities posed to CAD tools in the context of Better Than Worst-Case design. In particular, additional support is required for analyzing runtime characteristics of designs, and many opportunities are created to incorporate typical-case optimizations into synthesis, testing and verification.

1. INTRODUCTION

The advent of nanometer feature sizes in silicon fabrication has triggered a number of new design challenges for computer architects. These challenges include design complexity, device uncertainty and soft-errors. It should be noted that these new challenges add to the many challenges that architects already face in order to scale systems' performance while meeting power and reliability budgets.

The first challenge of concern is design complexity. As silicon feature sizes decrease, architects have available increasingly large transistor budgets. According to Moore's law, which has been tracked for decades by the semiconductor industry, architects can expect to have available twice the number of transistors every 18 months. In pursuit of increased system performance, they typically employ these transistors in components that contribute to increased instruction level parallelism and reduced operational latency. While many of these transistors are assigned to regular, easy-to-verify components, such as branch predictors and caches, many others find their way into complex devices that increase the burden of verification placed on the design team. For example, the Intel Pentium IV architecture (follow-on of the Pentium Pro) introduced a number of complex components, including a trace cache, instruction replay unit, vector arithmetic, and staggered ALUs [13]. These new devices, made affordable by generous transistor budgets, lead to even more challenging verification efforts. In a recent paper detailing the design and verification of the Pentium IV processor, it was observed that its verification required 250 person-years of effort, a full three-fold increase in human resources compared to the design of the earlier Pentium Pro processor [6].

The second challenge architects face is the design uncertainty that is created by increasing environmental and process variations. Environmental variation is caused by changes in temperature and supply voltage. Process variation results from device dimension and doping concentration variation that occurs during silicon fabrication. Process variation is of particular concern because its effects on devices are amplified as device dimensions shrink [2]. Architects are forced to deal with these variations by designing for worst-case device characteristics (usually, a 3-sigma variation from typical conditions), which leads to overly conservative designs. The effect of this conservative design approach is most evident by examining the extent to which hobbyists can overclock high-end microprocessors. For example, AMD's best-of-class Barton 3200+ microprocessor is specified to run at 2.2 GHz, yet it has been successfully overclocked up to 3.1 GHz [1]. This is accomplished by optimizing device cooling and voltage supply quality and by tuning system performance to the specific process conditions of their individual chip.

The third challenge of growing concern is soft errors that are caused by charged particles (such as alpha particles or neutrons) that strike the bulk silicon portion of a die. The

striking particle creates extra charge that can migrate into the channel of a transistor, and temporarily turn it on or off. The end result is a logic glitch that can potentially corrupt logic computation or state bits. While a variety of studies have been performed to demonstrate the unlikeliness of such events [16], great concern remains in the architecture and circuit communities, fueled by the trends of reduced supply voltage and increased transistor budgets, both of which exacerbate a design's vulnerability to soft errors.

The combined effect of these three design challenges is that architects are forced to work harder and harder just to keep up with system performance, power and reliability design goals. The unsurmountable task of meeting these goals with limited resource budgets and increasing time-to-market pressures has raised these design challenges to crisis proportion. In this paper, we highlight a novel design strategy to address these challenges, called *Better Than Worst-Case* design, that embraces a design style which separates the concerns of correctness and robustness from the ones of performance and power. The approach decouples designs into two primary components: a core design component and a simple checker. The core design component is responsible for performance and power efficient computing, and the checker is responsible for verifying that the core computation is correct. By concentrating the concerns of correctness into the simple checker component, the majority of the design is freed from these overarching concerns. With relaxed correctness constraints in the core component, architects can more effectively address the three highlighted design challenges. We have demonstrated in prior work (highlighted herein) that it is possible to decompose a variety of important processing problems into effective core/checker pairs. The designs we have constructed are faster, cooler and more reliable than traditional worst-case designs.

The remainder of this paper is organized as follows. Section 2 overviews the Better Than Worst-Case design approach and presents two effective designs solutions: DIVA checker and Razor logic. Better Than Worst-Case designs have the unique property that their performance is related to the typical-case operation of the core component. This is in direct contrast to worst-case designs, where system performance is bound by the worst-case performance of any component in the system. In Section 3, we demonstrate how *typical-case optimization (TCO)* can improve the performance of a Better Than Worst-Case design. We show that a typical-case optimized adder is faster and simpler than a high-performance Kogge-Stone adder. The opportunity to exploit typical-case optimization creates many new CAD challenges. In Section 4, we discuss the need for deeper observability of run-time characteristics at the circuit-level and present a circuit-aware architectural simulator that addresses this need. Section 5 suggests additional opportunities for CAD tools in the context of Better Than Worst-Case design, in particular highlighting opportunities brought by typical-case optimizations in synthesis, verification, and testing. Finally, Section 6 draws conclusions.

2. BETTER THAN WORST-CASE DESIGN

Better Than Worst-Case design is a novel design style that has been suggested recently to decouple issues of design correctness from those of design performance. The name *Better Than Worst-Case* design[1] underlines the improvement that this approach represents over traditional worst-case design techniques.

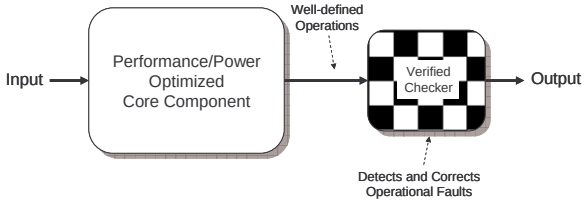

Figure 1: Better Than Worst-Case Design Concept

Traditional worst-case design techniques construct complete systems which must satisfy guarantees of correctness and robust operation. The previously highlighted design challenges conspire to make this an increasingly untenable design technique. Better Than Worst-Case designs take a markedly different approach, as illustrated in Figure 1. In a Better Than Worst-Case design, the core component of the design is coupled with a checker mechanism that validates the semantics of the core operations. The advantage of such designs is that all efforts with respect to correctness and robustness are concentrated on the checker component. The performance and power efficiency concerns of the design are relegated to the core component, and they are addressed independently of any correctness concerns. By removing the correctness concerns from the core component, its design constraints are significantly relaxed, making this approach much more amenable to address physical design challenges.

To find success with a Better Than Worst-Case design style, the checker component must meet three design requirements: i) it must be simple to implement lest the checker increase overall design complexity, ii) it must be capable of validating all core computation at its maximum processing rate lest the checker slow system operation, and iii) it must be correctly implemented lest it introduce processing errors into the system. In the following subsections, we present two Better Than Worst-Case designs that demonstrate how simple checker components can meet these requirements. The DIVA checker is an instruction checker that validates the operations of a complex microarchitecture design. Razor logic is a circuit-timing error checker that validates the timing of circuit-level computation. Using this capability to tolerate timing errors, a Razor design can eliminate power-hungry voltage margins. Additional examples of Better Than Worst-Case designs (including Razor) have been highlighted in a recent issue of IEEE Computer magazine [9].

2.1 DIVA Instruction Checker

At the University of Michigan, we have been exploring ways to ease the verification burden of complex designs. The DIVA (Dynamic Implementation Verification Architecture) project has developed a clever microprocessor design that provides a near complete separation of concerns for performance and correctness [5, 8, 18]. The design, illustrated in Figure 2, employs two processors: a sophisticated core processor that quickly executes the program, and a checker

[1]The term was coined by Bob Colwell, architect of the Intel Pentium Pro and Pentium IV processors.

processor that verifies the same program by re-executing all instructions in the wake of the complex core processor.

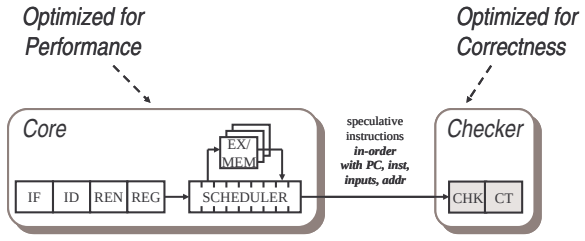

Figure 2: Dynamic Implementation Verification Architecture

The core processor is responsible for pre-executing the program to create the prediction stream. The prediction stream consists of all executed instructions (delivered in program order) with their input values and any memory addresses referenced. In a typical design, the core processor is identical in every way to the traditional complex microprocessor core, up to the retirement stage of the pipeline (where register and memory values are committed to state resources). In this design, the complex core processor is effectively "predicting" values because latent bugs or electrical faults could render its instruction results incorrect.

The checker processor follows the core processor, verifying the activities of the core processor by re-executing all program computation in its wake. The high-quality stream of instruction predictions from the core processor is exploited to simplify the design of the checker processor and speed up its processing. Pre-execution of the program on the complex core processor eliminates all the processing hazards (*e.g.*, branch mispredictions, cache misses, and data dependencies) that slow simple processors and necessitate complex microarchitectures. Thus it is possible to build an in-order checker pipeline without speculation, that can match the retirement bandwidth of the core. In the event of the core producing a bad prediction value (*e.g.*, due to a core design error), the checker processor fixes the errant value, flushes all internal state from the core processor, and then restarts the core at the instruction following the errant one.

We have shown through cycle-accurate simulation and timing analysis of a physical checker design that our approach preserves system performance while keeping low area overheads and power demands [5]. Furthermore, analysis suggest that the checker is a fairly simple state machine that can be formally verified [15] and scaled in performance and reused [19].

The simple DIVA checker addresses the concerns highlighted in the introduction, in that it provides significant resistance to design and operational faults, and provides a convenient mechanism for efficient and inexpensive detection of manufacturing faults. Specifically, if any design errors remain in the core processor, they will be corrected (albeit inefficiently) by the checker processor. The impact of design parameter uncertainty is mitigated since the core processor frequency and voltage can be tuned to typical-case circuit evaluation latency. Moreover, significant resistance to operational faults is also provided through low-cost and high-coverage techniques for detecting and correcting soft-error related faults. The DIVA approach uses the checker processor to detect energetic particle strikes in the core processor; as for the checker processor, we have developed a re-execute-on-error technique that allows the checker to check itself [18].

2.2 Razor Logic

Dynamic voltage scaling (DVS) has emerged as a powerful technique to reduce circuit's energy demands. In a DVS system, the application or the operating system identify periods of low processor utilization that can tolerate reduced frequency. The switch to a reduced frequency, in turns, enables similar reductions in the supply voltage. Since dynamic power scales quadratically with supply voltage, DVS technology can significantly reduce energy consumption with little impact on the perceived system performance. *Razor Logic* is an error-tolerant DVS technology [11, 3]. It incorporates timing error tolerance mechanisms that eliminate the need for the ample voltage margins required by traditional worst-case designs.

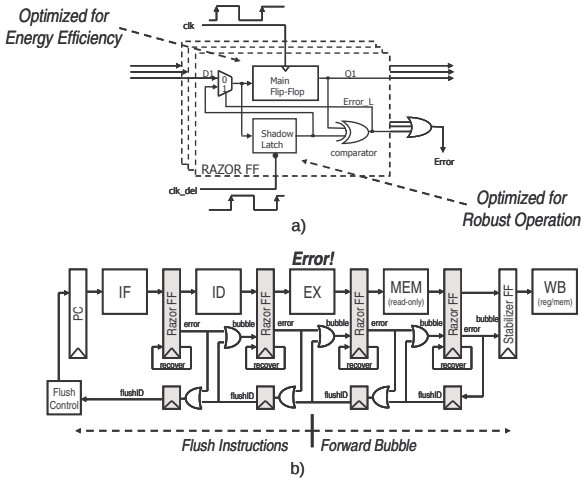

Figure 3: Razor Logic. The figure illustrates (a) the Razor flip-flip used to detect circuit timing errors, and (b) the pipeline recovery mechanism.

Figure 3a illustrates the Razor flip-flop, the mechanism by which Razor detects circuit timing errors. At the circuit level, a shadow latch augments each delay-critical flip-flop. A delayed clock controls the shadow latch, which provides a reliable second-sample of all pipeline circuit computations. In any particular clock cycle, if the combinational logic meets the setup time of the main latch, the main flip-flop and the shadow latch will latch the same data and no error will be detected. In the event that the voltage is too low or the frequency too high for the circuit computation to meet the setup time of the main latch, the main flip-flop data will not latch the same data as the shadow latch. In this case, the shadow latch data is moved into the main flip-flop where it becomes available to the next pipeline stage in the following cycle. To guarantee that the shadow latch will always latch the input data correctly, the allowable operating voltage is constrained at design time such that even under worst-case conditions, the combinational logic delay does not exceed the shadow latch's setup time.

Once a circuit-timing error is detected, a pipeline recovery

mechanism guarantees that timing failures will not corrupt the register and memory state with an incorrect value. Figure 3b illustrates the pipeline recovery mechanism. When a Razor flip-flop generates an error signal, pipeline recovery logic must take two specific actions. First, it generates a bubble signal to nullify the computation in the failing stage. This signal indicates to the next and subsequent stages that the pipeline slot is empty. Second, recovery logic triggers a backward moving flush train which voids all instructions in the pipeline behind the errant instruction. When the flush train reaches the start of the pipeline, the flush control logic restarts the pipeline at the instruction following the failing instruction.

While Razor cannot address the challenges posed by design complexity, it can effectively address design uncertainty and soft errors, while at the same time providing typical-case optimization of pipeline energy demands. In a worst-case methodology, design uncertainty leads to overly conservative design styles. In contrast, a Razor system can adapt energy and frequency characteristics to the specific process variation of an individual silicon die, eliminating the need for design-time remedies. Many soft errors manifest themselves as circuit-level timing glitches, which are addressed by Razor in the same manner as subcritical voltage-induced timing errors. We have implemented a prototype of Razor pipeline in 0.18um technology. Simulation results of the design executing the SPEC2000 benchmarks showed impressive energy savings of up to 64 percent, while the energy overhead for error detection and recovery was below 3 percent [11].

3. TYPICAL-CASE OPTIMIZATION

Better Than Worst-Case designs create opportunities to optimize the characteristics of the core component based on a thorough analysis of operational characteristics. For example, in a DIVA system, it is possible to reduce design time by functionally validating only the most likely operational states of the core component. In a Razor design, the decreased energy requirements of frequently executed circuit paths, mitigates the overall energy requirements of the design. We call this approach to design *typical-case optimization (TCO)*.

In this section, we provide an example of the benefits of TCO by optimizing the typical-case latency of an adder circuit. We identify common carry-propagation paths, based on program run-time characteristics, and construct a modified adder circuit with optimized latency characteristics for frequently-executed carry propagation paths. The resulting adder circuit is simpler and typically faster than a high-performance Kogge-Stone adder.

The first step in developing a TCO design is to understand the relevant run-time characteristics. In the specific situation of optimizing the carry propagation delay of an adder design, we must first gain a detailed understanding of carry-propagation distances for each bit position in an adder circuit, in the context of real program operations. To gather these measurements, we collected program addition vectors, that were generated by add, branch, load, and store instructions invoked during the execution of SPEC 2000 integer benchmarks, and ran them through a circuit-level representation of a 64-bit Kogge-Stone adder [17] (The simulator we used to perform these measurements is presented in Section 4). The adder circuit was instrumented to collect data on i) the bit locations were carry propagations started, ii) the length of carry propagations chains, and iii) the distribution of adder evaluation latency. To evaluate the added benefits of TCO for real program data, we also performed a similar analysis on random input vectors.

Figures 4 and 5 show the carry-propagation results for SPEC 2000 program data and random data, respectively. The surface graphs illustrate the carry-propagation distance for each bit-position of the adder circuit. The X axis indicates the starting bit-position of the carry-propagation, and the Y axis reports the length of the carry-propagation chain. For each carry-propagation, the Z axis gives the probability of a particular carry-propagation initial bit position and length when executing the specified data set.

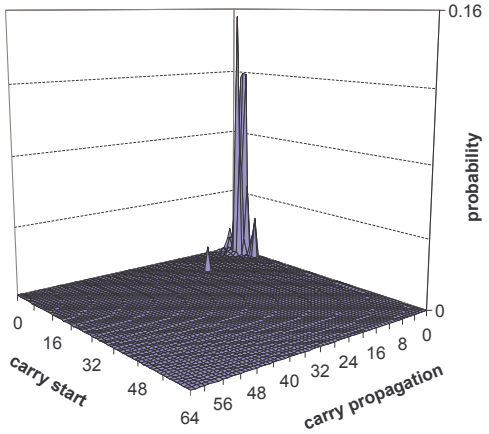

Figure 4: Carry Propagation Distribution for Typical Data

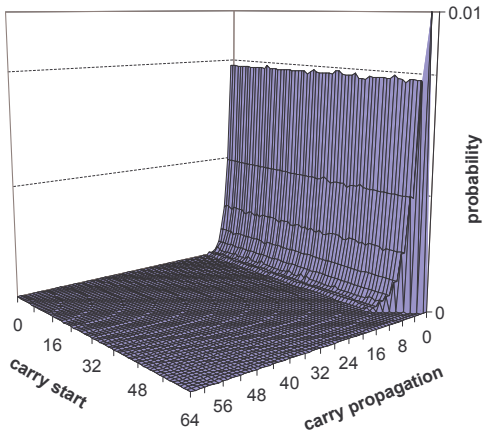

Figure 5: Carry Propagation Distribution for Random Data

As shown in Figure 4, real program data exhibits primarily short carry-propagation distances. In the least significant bits, propagation distances are nearly always less than 6 bits, while the more significant bits rarely generate a carry that propagates for more than 2 bit positions. As expected, the probability of a carry-propagation for purely random input vectors is independent of the initial bit position, and the propagation distance probability decreases geometrically with the distance of the propagation, since each successive

bit is equally likely to terminate the propagation chain.

This carry-propagation analysis suggests that for real program data, most carry propagation occurs in the least significant bits, and are propagated only for a short distance. We can optimize an adder design for these characteristics by creating an efficient carry propagation circuit optimized for frequently executed carry-propagation paths. Our 64-bit TCO adder is illustrated in Figure 6b, below the baseline Kogge-Stone adder of Figure 6a, a popular adder topology optimized for minimal worst-case latency. The TCO adder implements a dedicated carry-lookahead circuit for carry-propagation of up to 6 bits of length and starting from any of the least-significant 9 bit positions of the adder. The remaining bit positions in the TCO adder implement a dedicated 2 bit carry propagation. Any computation requiring an unsupported carry propagation pattern will eventually compute correctly on the TCO adder through the use of a fall-back ripple-carry backbone logic.

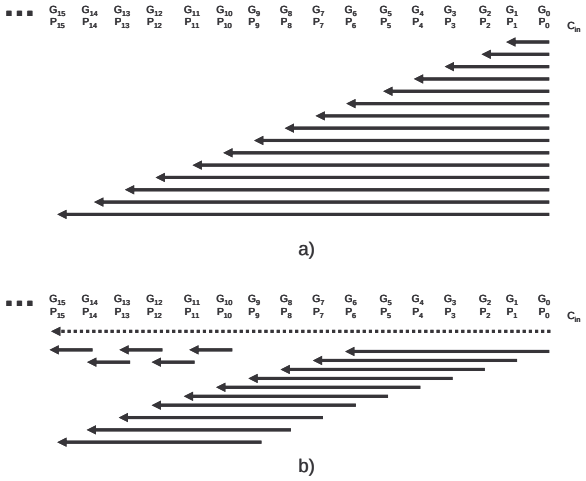

Figure 6: Adder Topologies. The figure illustrates the carry propagation logic for the (a) Kogge-Stone adder and (b) typical-case optimized adder. Solid lines represent a carry-lookahead logic circuit; dashed lines represent a ripple-carry logic circuit.

Table 1 compares the relative performance of the baseline Kogge-Stone adder with the TCO adder. For each adder, the table lists the worst-case latency for any input vector (in gate delays), the average latency for all typical-case vectors, and the average latency over all random input vectors.

Adder	**Latency** (in gate delays)		
Topology	Worst-Case	Typical-Case	Random
Kogge-Stone	8	5.08	7.09
TCO Adder	128	3.03	3.69

Table 1: Relative Performance of Adder Designs

The worst-case latency is indicative of the delay one would expect from the adder if placed into a traditional worst-case style design. The worst-case performance of the Kogge-Stone adder is proportional to $log_2 N$, where N is the number of bits in the adder computation. The worst-case computation of the TCO adder is proportional to N, since some computation will require full evaluation of the ripple carry adder backbone. As shown, the worst-case performance of the Kogge-Stone adder is much more favorable than the TCO adder, making the Kogge-Stone adder more appropriate for a worst-case style design.

The typical-case latency represents the average delay for all the input vectors in the SPEC2000 test set to complete. The typical-case latency of the TCO adder is much less than the worst-case latency of even the highly optimized Kogge-Stone adder circuit. This result is to be expected since only a few evaluations require the use of the backbone ripple-carry logic. Moreover, The TCO adder performs better, on average, even on the random data set, since the optimized paths have enough impact to contrast the rare worst-case scenarios.

As expected, the results of the random-case experiments on the TCO design, while better than worst-case latency, cannot compete with the typical program data experiments. It is clear from the random-case results that understanding the typical-case operations of a component and then targeting the optimization to these operations can have a dramatic effect on the typical-case latency of a core component.

As evidenced by these experiments, typical-case optimization of circuits can render significant improvements in typical-case performance. However to enable successful TOC designs, there is a need for new specialized CAD tools that are enhanced to *expose* and *exploit* run-time operational characteristics. We discuss these challenges in the following two sections.

4. ARCHITECTURAL SIMULATION AND ANALYSIS

The development of Better Than Worst-Case designs poses a whole new set of demands on CAD tools. One core requirement of this approach is the need to gain a deeper appreciation of which situations are typical and which situations are extreme and rare when operating the system to be designed. For instance, for the adder circuit presented above, we need to evaluate the most probable sources of carry chains and the most typical carry propagation depths. Or, in the case of Razor logic, it's important to be able to evaluate how frequently the recovery mechanism intervenes to correct the system's operation. Novel simulation solutions are needed to address these new class of concerns and evaluation demands. Moreover, new simulations tools must enable designers to evaluate the performance and correctness of these new systems, which often bring together circuit-level issues (such as voltage and process variations) with high level solutions.

To address at least some of these simulation requirements, we have developed an architectural simulation modelling 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.

Figure 7 illustrates the software architecture of our circuit-aware architectural simulator. The simulator model is based on the SimpleScalar modeling infrastructure [4]. The Sim-

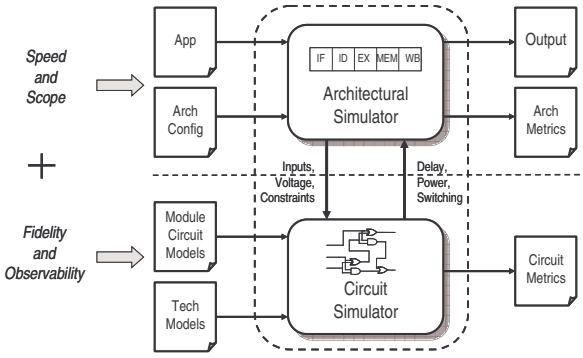

Figure 7: Circuit-Aware Architectural Simulation

pleScalar tool set is capable of modeling a variety of platforms ranging from simple unpipelined processors to detailed dynamically scheduled microarchitectures with multiple levels of memory hierarchies. The architectural simulator takes two primary inputs: a configuration file that defines the microarchitecture being modelled, 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 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.

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 stand alone circuit analysis tool, capable of transient fault injection experiments, and of investigating process variation.

The great challenge of implement circuit-aware architectural simulation is achieving acceptable simulation speeds. To meet this goal we have employed three domain-specific circuit simulation speed optimizations: 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 first optimization is constraint cased circuit pruning. This optimization allows the architectural simulator to specify constraints upon which circuit simulator results are of interest to the architectural simulation (*i.e.*, they would perhaps cause an architectural level control decision to be invoked). For example, a Razor simulation is interested in circuit latency only when the latency is known to be longer than the clock period of the current clock. The circuit simulator uses these constraints to determine when to drop logic transition events that are guaranteed to not violate the constraints.

The second optimization we implemented was circuit timing memoization. We leverage program 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.

Finally, we employed SimPoint Analysis to reduce the number of instructions we needed to process in order to make clear judgments about program performance characteristics [7]. 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. 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.

5. TYPICAL-CASE OPTIMIZED SYNTHESIS, VERIFICATION AND TESTING

Circuit-aware architectural simulation is only a small example of new solutions in computer-aided design software

to respond to the new design challenges described above, and the trends towards designs optimized for typical case scenarios.

In the synthesis domain, the traditional approach has been to characterize library components and modules by their worst-case metric values. For instance, given a specific feature size and operating voltage, the characterizing metrics would report the worst-case propagation delay and power consumption. While these metrics have worked well in the past to design conservative systems that operated correctly under any possible condition, they are already too limiting in today's developments, where high performance demands force design teams to shave off any extra margins, for instance, by overruling the worst-case metrics of the component in isolation, and focusing on its electrical characteristics in the context of the system where it is used. The lack of synthesis software that can fully exploit these extra margins, poses a much higher demand on the engineering team that has to manually iterate multiple times through the synthesis process to achieve timing closure and to satisfy power and performance requirements.

In a Better-Than-Worst-Case scenario synthesis libraries would have to characterize components by cost metrics distributions, instead of single data points. For instance, the delay of a component, for a given set of operating conditions, could be simplified as a set of discrete intervals of delay values vs. the probability of the component stabilizing within that delay. In relation to the traditional approach, the delay value that is met with probability 1 corresponds to the delay value reported by a traditional synthesis library. Synthesis software should support the designer in selecting a desired level of confidence in the cost metrics of the components for different portion of a design. In general, the checker portion of the design should be designed using the most conservative metrics, while the high performance portion could use more aggressive selections. The use of statistical analysis in CAD software has been mostly in the area of analog design [12, 14, 10]; recent work by Agarwal incorporates process variation effects in the statistical analysis of clock skews [2]. These are all initial attempts of evaluating design parameters using statistical means, in a TCO design methodology, statistical techniques must be much more pervasive in all aspects of the design process.

Moreover, component characterization and optimized design of macro-module could allow for extra optimizations if based on "typical" data sets, as in the adder example of Section 3. Enabling designers to explore this additional opportunity requires specialized simulation software that summarizes results in distribution curves appropriate for the synthesis process.

While the synthesis of typical-case systems poses mostly a new set of challenges to CAD software, the burden of functional verification could be alleviated in the new methodology. Today, the challenge of design verification is to guarantee that a system is functionally correct under any possible input stimuli. On one hand, simulation-based software can only provide confidence in design correctness that is limited to the specific set of tests run on the system; on the other hand, formal and semi-formal verification tools struggle in tackling the complexity of current designs, and can typically only focus on small modules and macro-blocks of the system. In a TCO design setting, verification has the opportunity to prioritize its focus: the checker portion of the design demands the highest level of correctness, while the focus for the high performance portion is on typical-case correctness. The benefit is that the simpler, smaller checker portion of the design lends itself more easily to be formally verified, as it is the case for the DIVA architecture of Section 2.1 [15]. On the other hand the high-performance, complex portion, is more suitable to simulation-based verification where simulation tests are mostly focused on the typical, most frequently-used execution scenarios. Architectures where checker and performance portions are not as easily separable, an example of which is the Razor architecture, can still benefit from the conceptual separation between verification-critical and verification-typical portions within the design. For instance, in the Razor design, most verification efforts should focus on the execution paths through the shadow latches.

Testing presents new challenges as wells as new opportunities when faced with TCO designs. Once again, the most critical portion to be tested is the checker part of a design. Because of its simpler architecture, it is easier to obtain a complete and compact set of tests for this portion. Once the checker is verified, the high performance design can often be tested by running the system with the operational checker, and the checker itself can be used to evaluate the quality of the die. An analysis of the testability of the DIVA architecture was presented in [18]. Complex TCO systems, however, present a whole set of new challenges for testing. For instance, it is even more critical that the checker is fully tested than in traditional designs, since in TCO systems the high performance components are expected to be more faulty than traditional designs. Moreover, when the TCO systems target the separation between correctness and performance through complex new devices, such as the high specialized Razor latches, novel, ad-hoc testing techniques need to be developed.

6. CONCLUSIONS

In this paper we have discussed Better Than Worst-Case design methodology, a new approach to designing high performance complex digital systems, that defeats the challenges posed by the increasingly high integration, and small feature size trends of the semiconductor industry. We discussed two design solutions in these domain, the DIVA checker and the Razor logic. We also showed an adder design example that performs typical-case optimization and performs better than traditional worst-case optimized solutions in the context of Better Than Worst-Case designs. While on one hand this novel design methodology is gaining increasing interest from the design community, on the other it requires a re-evaluation of the driving optimization goals in CAD tools, by posing a whole new set of challenges, and sometimes opportunities, in synthesis, verification and testing, some of which have been highlighted.

Acknowledgements

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

7. REFERENCES

[1] Overclockers.com website, overclockers forum. *http://www.overclockers.com*, 2004.

[2] A. Agarwal, V. Zolotov, and D. Blaauw. Statistical clock skew analysis considering intra-die process variations. *IEEE*

Transactions on Computer-Aided Design of Integrated Circuits and Systems, 23(8):1231–1242, Aug. 2004.

[3] T. Austin, D. Blaauw, T. Mudge, and K. Flautner. Making typical silicon matter with razor. *IEEE Computer*, Mar. 2004.

[4] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer*, Feb. 2002.

[5] T. M. Austin. Diva: A reliable substrate for deep submicron microarchitecture design. *32nd International Symposium on Microarchitecture (MICRO-32)*, Dec. 1999.

[6] R. M. Bentley. Validating the pentium 4 microprocessor. *International Conference on Dependable Systems and Networks (DSN-2001)*, July 2001.

[7] B. Calder. Simpoint website. In *http://www.cse.ucsd.edu/ calder/simpoint/*, 2003.

[8] S. Chatterjee, C. Weaver, and T. Austin. Efficient checker processor design. In *33rd International Symposium on Microarchitecture (MICRO-33)*, Dec. 2000.

[9] B. Colwell. We may need a new box. *IEEE Computer*, 2004.

[10] P. Crippa, C. Turchetti, and M. Conti. A statistical methodology for the design of high-performance CMOS current-steering DACs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(4):377–394, Apr. 2002.

[11] 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.

[12] N. Herr and J. Barnes. Statistical circuit simulation modeling of CMOS VLSI. *IEEE Transactions on Circuits and Systems*, 5(1):15–22, Jan. 1986.

[13] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, Feb. 2001.

[14] C. Michael and M. Ismail. *Statistical Modeling for Computer-Aided Design of MOS VLSI Circuits*. Kluwer Academic Publishers, 1993.

[15] M. Mneimneh, F. Aloul, S. Chatterjee, C. Weaver, K. Sakallah, and T. Austin. Scalable hybrid verification of complex microprocessors. In *38th Design Automation Conference (DAC-2001)*, June 2001.

[16] S. S. Mukherjee, C. T. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. Measuring architectural vulnerability factors. *IEEE MICRO*, Dec. 2003.

[17] J. M. Rabaey, A. Chandrakasan, and B. Nikolic. *Digital Integrated Circuits*. Prentice-Hall, 2003.

[18] 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.

[19] C. Weaver, F. Gebara, T. Austin, and R. Brown. Remora: A dynamic self-tuning processor. *UM Technical Report CSE-TR-460-02*, July 2002.