Argus: Low-Cost, Comprehensive Error Detection in Simple Cores

Argus, a novel approach for detecting errors in simple processor cores, dynamically verifies the correctness of the four tasks performed by a von Neumann core: control flow, data flow, computation, and memory access. Argus detects transient and permanent errors, with far lower impact on performance and chip area than previous techniques.

Albert Meixner
Michael E. Bauer
Daniel J. Sorin
Duke University

Technological trends are leading to more hardware errors due to both transient and permanent physical phenomena.[1] The first and most important step in dealing with these errors is detecting them. Once an error is detected, the system can avoid silent data corruption and potentially recover to a pre-error state and resume execution. In this work, we focus on relatively simple cores, rather than speculative, out-of-order cores. Simple cores are becoming more attractive because of their low power consumption, particularly for multicore chips, embedded applications, and applications for which throughput is more important than latency. For example, the UltraSPARC T1 (Niagara) chip contains eight simple cores, the Cray MTA consists of simple multithreaded cores, and the Silicon Packet Processor in Cisco's CRS-1 router has 188 Tensilica Xtensa single-issue, in-order cores. For many applications of simple cores, reliability is important but its hardware and power costs must be minimal.

Core error detection can be achieved by simply replicating each core—dual modular redundancy (DMR)—but this option is extremely expensive. Even if providing the required number of transistors is technologically feasible, DMR incurs a large opportunity cost and approximately doubles core power consumption. Other detection schemes exist—such as DIVA and redundant multithreading (see the "Related Work" sidebar)—but all of them are either incomplete or expensive, in terms of area or performance, for simple cores. Our goal is to provide a low-cost, low-power mechanism for comprehensively detecting transient and permanent errors in a simple microprocessor core.

Argus overview

The key insight that Argus exploits is that von Neumann processor cores perform only four basic activities: choosing the sequence of instructions to execute (control flow), performing the computation specified by each instruction, passing the result of each instruction to its data-dependent instructions (data flow), and interacting with memory. We have proven that by checking that these activities are performed correctly, Argus can detect all possible core errors, except errors in the parts of the core that handle I/O, exceptions, and interrupts.[2] We consider Argus to be a high-level error detection scheme, because these four activities are not specific to any particular microarchitecture, design, or implementa-

Published by the IEEE Computer Society

tion. This set of activities is similar to the set of activities that DIVA checks,[3] but Argus' approach to checking them is fundamentally different.

Control-flow checking

A control-flow checker verifies that the runtime execution path is valid with respect to the static control-flow graph (CFG) of the program binary.[4] If the static and dynamic CFGs conflict, an error has been detected. Unlike many control-flow checkers, Argus considers liveness to also be a part of control-flow correctness.

Data-flow checking

A data-flow checker ensures that the static data-flow graph (DFG) of the program binary matches the DFG reconstructed at runtime, and that the values traversing the DFG are not corrupted.[5]

Computation checking

A computation checker detects errors in functional units. Some checkers require a fully replicated functional unit, but many utilize knowledge about the initial result to simplify the redundant computation. Sellers, Hsiao, and Bearnson's book provides an excellent survey of existing checkers for adders, multipliers, dividers, bit-wise logic units, and so on.[6]

Memory checking

A minimal memory checker must be able to detect data corruption in the memory system as well as errors that cause the wrong data word to be accessed. In more complex memory systems that support multiple outstanding requests and potentially multiple cores, faults can also manifest themselves as incorrect orderings of memory accesses. We don't consider this type of error, because there are only a few unlikely scenarios for ordering errors in simple cores. An example of a complex memory checker that could be used with Argus is available elsewhere.[7]

Argus-1 implementation

Argus-1 is an implementation of Argus that illustrates the engineering trade-offs between checker costs and error coverage. Although perfect checkers can be designed, their costs are not always worth their additional error coverage, as compared to near-perfect checkers. We have proven that Argus-1 detects the same errors as an ideal Argus implementation, except for false negatives due to finite-size checksums and memory-ordering errors.[2]

To obtain realistic information about the costs and complexity associated with implementing Argus-1, we have built Argus-1 in Verilog and incorporated it into the OpenRISC 1200 (OR1200) processor core.[8]

Baseline OpenRISC processor

The OR1200 processor core is a 32-bit scalar (one-wide), in-order RISC core with a four-stage pipeline and 32 general-purpose registers. This core represents the low end of the simple cores that are expected to be used, perhaps in conjunction with a small number of superscalar cores, in multicore chips. Figure 1 shows how Argus-1 is integrated into the OR1200 core.

Control-flow and data-flow checkers

Argus-1's control-flow and data-flow checkers are based on our earlier work on dynamic data-flow verification (DDFV).[5] DDFV detects errors in the core's data flow by comparing the static DFG specified in the program to the dynamic DFG within the processor during execution. To avoid problems with data-dependent branches, which dynamically alter the DFG, DDFV performs checks at the granularity of basic blocks for which the correct DFG is known at compile time. Both DFGs are represented using constant-size signatures. The DFG signatures are computed as a summary of state history signatures (SHSs) attached to each architectural location within the system. Whenever a location is assigned a new value, its history is updated to reflect the operation that generated the value and the history of the input operands to that operation.

The static DFG signatures are computed at compile time and embedded in the program such that they can be read by the processor and compared to the dynamic signatures computed at runtime. We minimized the number of embedded Signature instructions (NOPs) by storing signature bits in unused instruction bits; actual

Related Work

There is a long history of research in error detection.

Redundant cores

Replicating a core provides a conceptually simple mechanism for detecting errors. This approach is, in terms of hardware and power, prohibitively expensive for commodity hardware.

DIVA

DIVA (Dynamic Implementation Verification Architecture) and Argus check similar invariants, but DIVA's approach is quite different.[1,2] DIVA uses *N* simple checker cores to detect errors in a *N*-wide superscalar processor. DIVA is an excellent, low-cost design option for protecting large cores with simple instruction decoding logic. For example, a DIVA checker is only 6 percent of an Alpha 21264 core.[1] However, for simple, small cores, there is little opportunity to make the checker cores smaller than the cores they are checking, so using DIVA becomes almost indistinguishable from using redundant cores.

Redundant multithreading

There are many varieties of redundant multithreading (RMT) schemes,[3] but they all share the goal of using otherwise idle thread contexts to provide redundancy in SMT cores. RMT has three significant costs: the performance loss due to thread contention (estimated at 30 percent[4]), the opportunity cost of using threads for redundant computation instead of useful work, and the energy consumed by the redundant threads. RMT also has the implicit cost of requiring an SMT core and cannot detect permanent errors in non-replicated units.

BulletProof

The BulletProof pipeline uses built-in self-test (BIST) to detect and diagnose (isolate) 89 percent of permanent faults,[5] but it cannot detect transient errors. BulletProof increases the area of a four-wide very long instruction word (VLIW) core, excluding caches, by 9.6 percent. This overhead is likely to be greater for a single-wide core, because BIST tables and other checker hardware singletons cannot be amortized over multiple instances of the units they check.

Software redundancy

Software replication of instructions can serve to detect transient hardware errors,[6,7] albeit at the cost of a 50 percent slowdown for an out-of-order processor and high energy consumption.[7] The performance loss for a simple in-order core would be roughly 100 percent, owing to the lack of idle slots in which to execute redundant instructions.

Error detecting and correcting codes

Error codes are excellent at detecting errors in storage and messages. Certain codes can also check computations. However, error codes are not applicable to general logic.

References

1. C. Weaver and T. Austin, ''A Fault Tolerant Approach to Microprocessor Design,'' *Proc. Int'l Conf. Dependable Systems and Networks* (DSN 01), IEEE CS Press, 2001, pp. 411-420.
2. T.M. Austin, ''DIVA: A Dynamic Approach to Microprocessor Verification,'' *J. Instruction-Level Parallelism,* vol. 2, May 2000; http://www.jilp.org/vol2/v2paper7.pdf.
3. E. Rotenberg, ''AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors,'' *Proc. 29th Int'l Symp. Fault-Tolerant Computing Systems* (FTCS 99), IEEE CS Press, 1999, pp. 84-91.
4. S.S. Mukherjee, M. Kontz, and S.K. Reinhardt, ''Detailed Design and Implementation of Redundant Multithreading Alternatives,'' *Proc. 29th Ann. Int'l Symp. Computer Architecture* (ISCA 02), IEEE CS Press, 2002, pp. 99-110.
5. S. Shyam et al., ''Ultra Low-Cost Defect Protection for Microprocessor Pipelines,'' *Proc. 12th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 06), ACM Press, 2006, pp. 73-82.
6. N. Oh, P.P. Shirvani, and E.J. McCluskey, ''Error Detection by Duplicated Instructions in Super-Scalar Processors,'' *IEEE Trans. Reliability,* vol. 51, no. 1, Mar. 2002, pp. 63-74.
7. G.A. Reis et al., ''SWIFT: Software Implemented Fault Tolerance,'' *Proc. Int'l Symp. Code Generation and Optimization* (CGO 05), IEEE CS Press, 2005, pp. 243-254.

Signature instructions are embedded only in basic blocks with insufficient unused bits.

Unified control and data-flow checking. Argus-1 uses a basic block's data-flow signature as both a representation of the block's internal control and data flow and a unique, address-independent block identifier necessary for interblock control-flow checking. We refer to this single signature as the *data-flow and control signature* (DCS).

What enables Argus-1 to use the DCS for both data-flow and interblock control-flow checking is the way DCSs are embedded and used. Unlike DDFV, which embeds a single signature into each block,[5] Argus-1 embeds into each basic block the DCS of each of its legal successor blocks. At runtime, the control-flow checker decides, on the basis of information received from the computation checker, which of the legal successors will be executed next, and then

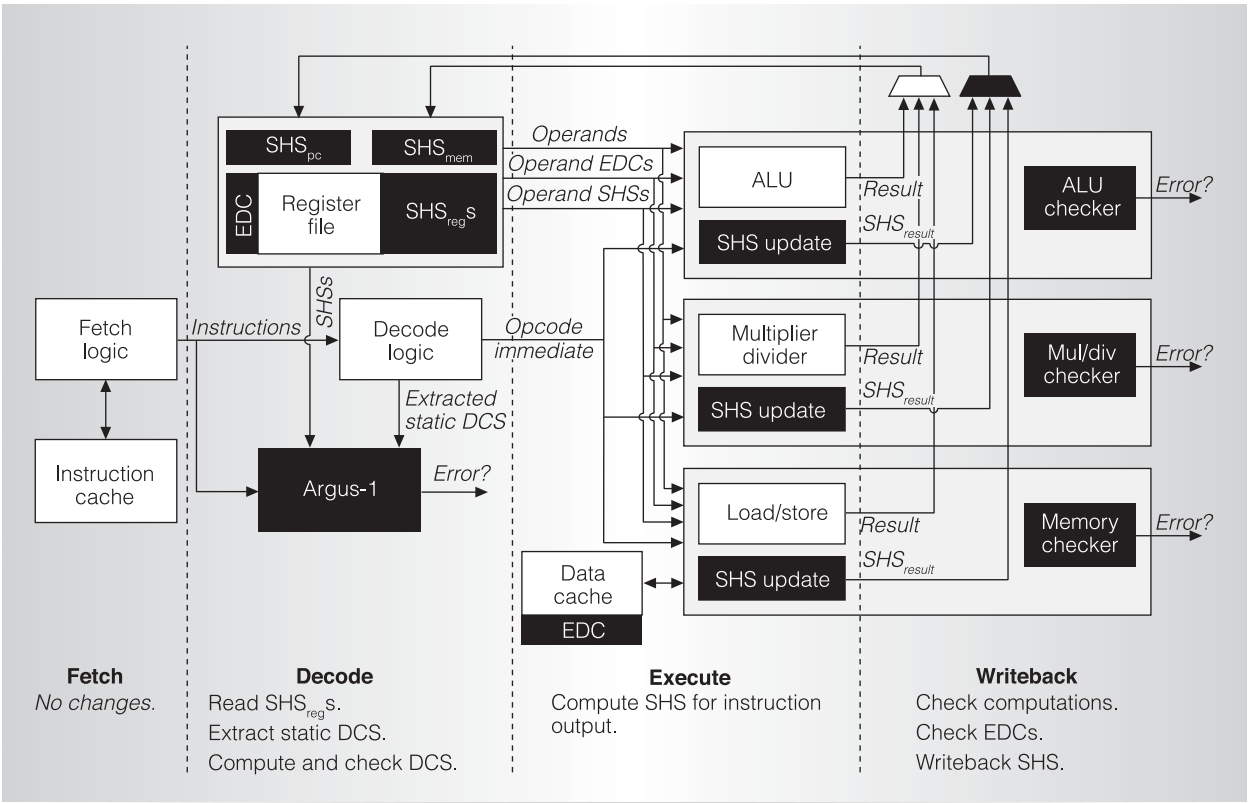

Figure 1. Summary of Argus-1 implementation.

passes the corresponding DCS to the data-flow checker. If, because of an error, the wrong successor or any other illegal block is executed next, the DCS computed by the data-flow checker will not match the DCS anticipated by the control-flow checker and, barring signature aliasing, an error will be detected.

Data value correctness. Beyond checking the shape of the data-flow graph, the data-flow checker must also ensure that data values are transmitted correctly. To detect errors in data values, Argus-1 adds a parity bit to each register and each part of the data path that carries an operand or instruction result.

Checking liveness. There are many techniques for checking liveness, including simple watchdog units.[9] Our watchdog has a six-bit counter. At every clock cycle, the counter is reset if the pipeline is not stalled, and is incremented when the pipeline is stalled. When the counter saturates, the watchdog indicates an error. To bound the time between control-flow checks, Argus-1 also requires a fixed limit on the size of basic blocks.

Computation checker

The computation checker consists of several functional-unit subcheckers. Figure 2 shows how Argus-1 is integrated into a functional unit. For each functional unit, we have a subchecker and an SHS compu-

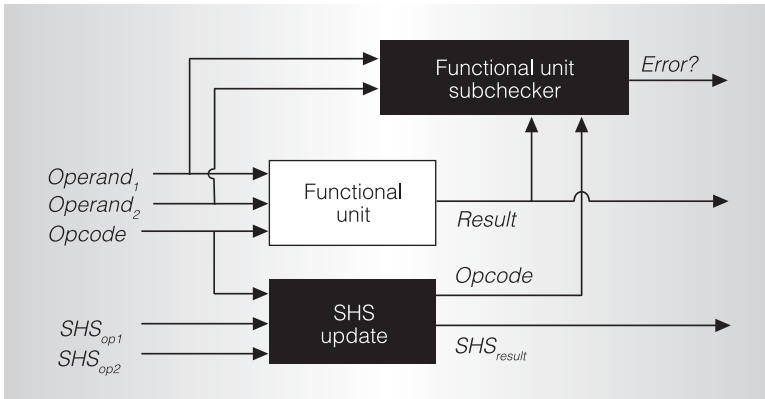

Figure 2. Functional unit with error detection.

Table 1. Error injection results.

Error type	Unmasked, undetected* (%)	Unmasked, detected (%)	Masked, undetected (%)	Masked, detected (%)
Transient	0.76	37.4	38.2	23.7
Permanent	0.46	37.6	38.2	23.7

* Silent corruptions

tation unit. The subcheckers are similar to previously developed mechanisms for checking computation (for example, using modulo arithmetic to check multiplication).[6]

Memory checker

The memory hierarchy consists of the core's load-store unit, caches, and main memory. Argus-1 detects address computation errors using an adder checker like the one used for checking the ALU's adder. Argus-1 also detects errors in data realignment in byte and half-word (16-bit) loads and stores using the ALU's subchecker. To detect data corruption, Argus-1 adds parity to each word in the data cache and memory—assuming an error detecting code (EDC) is not already present. No parity is needed for the instruction cache, because errors in instructions will cause incorrect control flow or data flow. Argus-1 also protects against errors that cause a load or store to access the wrong word, despite providing a correct address, by embedding the physical address into the data in the caches and memory.[2]

Argus-1 cannot detect memory errors in two concrete, yet unlikely, scenarios: an access that misses in a cache even though it should have hit, and a store issued to the cache that does not perform the actual write. We could modify Argus-1 to detect such errors by adding redundant tag comparisons and verifying reads after every store. Because of power consumption and performance considerations, we decided against these options.

Experimental evaluation

The goals of our evaluation are to confirm Argus-1's error detection coverage and determine its area and performance overheads.

Error detection coverage

An Argus implementation with perfect checkers can detect all possible single-error scenarios (and many multiple-error scenarios) in the nonexceptional part of the core. However, because of physical constraints, the Argus-1 implementation has sacrificed some small amount of error coverage. Argus-1 cannot detect errors in exception and interrupt logic, errors due to aliasing (in data checksums, DCS, and the multiply checker), and some memory errors described earlier.

We performed error injection experiments to empirically test Argus-1's error detection coverage. We performed the experiments while the core was running a stress-test microbenchmark that involves a broad range of registers and instruction types. We injected single transient and permanent bit-inversion errors in all portions of the microprocessor core, including the features added for Argus-1. From among the roughly 40,000 total gates, we randomly sampled 5,000 gate outputs on which to inject bit flips. We did not inject errors in the caches, but we did inject errors in the core's interface to memory.

For each error injection experiment, we classified its result along two axes. First, was the error detected? Second, was the error masked? The errors that we want most to avoid are unmasked, undetected errors, which represent silent data corruptions. In Table 1, we show the results of these experiments. One result not listed in the table is that Argus-1 never reported *false positives* (errors that did not actually occur).

Unmasked errors. Most importantly, we observe that silent data corruptions are extremely rare, compared to detected errors. Of the unmasked transient and permanent errors, Argus-1 detected 98.0 percent and 98.8 percent, respectively.

We examined which parts of Argus-1 were responsible for detecting each error. The computation checkers were responsible for 45 percent of detected errors. The next largest contributor to error coverage was parity (on operands, registers, and load values), which caught 36 percent of detected errors. The DCS comparison caught 16

percent, and the watchdog caught 3 percent. These results confirm that a composition of all checkers is necessary to achieve good coverage.

Masked errors. A large fraction of injected errors were masked, which is not surprising. All errors in Argus-1 hardware are masked, because they have no impact on the OR1200 core's execution. Of the masked errors, Argus-1 detected 38.3 percent.

Area overhead

We used our CAD tools, Synopsys Design Compiler and Cadence Silicon Ensemble, to floor-plan and lay out the core, both with and without Argus-1. We did not include the debugging hardware or the translation look-aside buffer (TLB). Our CAD tools use the publicly available VTVT 0.25-µm standard cell library.[10] We present our results in Table 2. We first observe that the unmodified OR1200 core requires an area of 6.59 mm^2 (2.565 mm × 2.565 mm), and the core with Argus-1 uses 16.6 percent more area. Most of Argus-1's area is consumed by history fields and signature computation logic used for data-flow and control-flow checking. The various computation checkers are the second major area contributor. The remainder of Argus-1's area is for control logic and the watchdog timer.

To determine total chip area overhead (not only core area overhead), we used Cacti 3.0 to calculate the area of the caches.[11] Both the instruction and data caches are 8 Kbytes. Argus-1's data cache adds area for parity, but, as we've mentioned, Argus-1 does not need to add parity to the instruction cache. If we compare Argus-1 to an unmodified OR1200 chip (with no error detection on the core or caches), Argus-1 consumes only 10 to 11 percent more area.

Table 2. Area overhead for adding Argus-1 error detection.

	Area (mm^2)		
Component	OR1200 alone	OR1200 with Argus-1	Area overhead (%)
Core	6.58	7.67	16.6
Instruction cache: 1-way	2.14	2.14	0
Instruction cache: 2-way	2.42	2.42	0
Data cache: 1-way	2.14	2.24	4.9
Data cache: 2-way	2.42	2.54	5.1
Total: 1-way	10.86	12.05	10.9
Total: 2-way	11.42	12.63	10.6

Performance overhead

Argus-1's error detection hardware does not cause any pipeline stalls or delay instruction retirement, because Argus-1 is designed to invoke backward error recovery once an error is detected. Our CAD tools also showed no increase in any critical paths due to Argus-1 logic, and thus we do not expect changes in clock cycle time. Hence, Argus-1's only potential impact on core performance comes from having Signature instructions embedded in the instruction stream when insufficient unused bits are available to store the DCSs. Signature instructions consume instruction cache space as well as fetch and instruction decoding bandwidth. We use the OR1200 simulator and the MediaBench benchmark suite[12] to analyze how this impacts performance. We assume a memory configuration typical for an embedded system; the data and instruction cache are each 8 Kbytes and two-way set-associative; hits take one cycle;

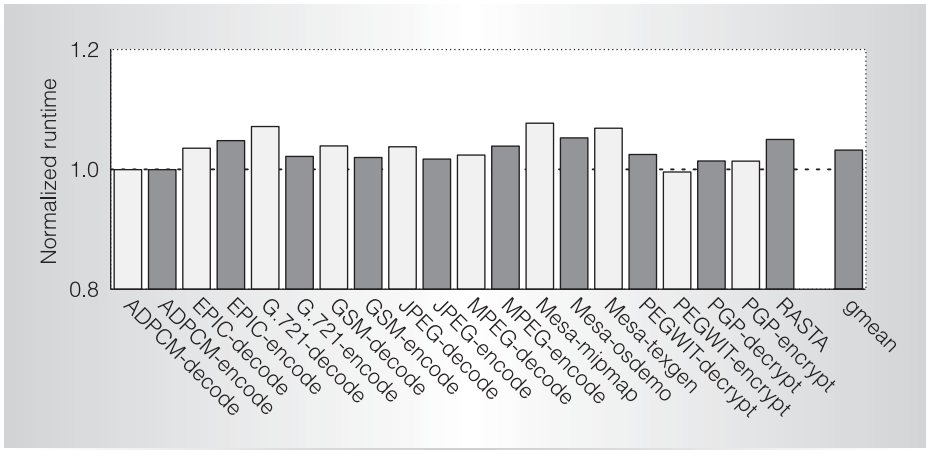

Figure 3. Argus-1 runtime overhead (two-way cache).

and misses take 20 cycles. In Figure 3, we plot Argus-1's performance impact. The observed slowdown varies greatly between benchmarks, but it never exceeds 10 percent, and it averages only 3.2 percent across all benchmarks.

We expect simple cores to remain popular for many embedded applications as well as for multicore chips, and detecting errors in such cores is important for ensuring dependability. Argus provides a viable, efficient, low-cost solution to this problem. The key to Argus' efficiency is to check invariants instead of components. In addition to improving efficiency, invariant checking also makes it is easier to formally reason about Argus' error coverage capability. The next step in this project is to increase Argus' error detection coverage to include the exceptional portion of the core and to integrate Argus with a previously developed mechanism for detecting errors in the memory system.[7] Another avenue of research we are pursuing is to provide mechanisms for diagnosing and recovering from errors detected by Argus. MICRO

Acknowledgments

This material is based on work supported by the National Science Foundation under grant CCR-0444516, the National Aeronautics and Space Administration under grant NNG04GQ06G, a gift from Toyota InfoTechnology Center, and an equipment donation from Intel. We thank Fred Bower, Derek Hower, Alvy Lebeck, Anita Lungu, and Bogdan Romanescu for their feedback on this work. We thank Bogdan Romanescu and Heather Sarik for help with the experiments.

References

1. *International Technology Roadmap for Semiconductors*, 2003; http://www.itrs.net.
2. A. Meixner, M.E. Bauer, and D.J. Sorin, ''Argus: Low-Cost, Comprehensive Error Detection in Simple Cores,'' *Proc. 40th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (MICRO 07), IEEE CS Press, 2007, pp. 210-222.
3. T.M. Austin, ''DIVA: A Dynamic Approach to Microprocessor Verification,'' *J. Instruction-Level Parallelism*, vol. 2, May 2000; http://www.jilp.org/vol2/v2paper7.pdf.
4. X. Delord and G. Saucier, ''Formalizing Signature Analysis for Control Flow Checking of Pipelined RISC Microprocessors,'' *Proc. Int'l Test Conf.* (ITC 91), IEEE Press, 1991, pp. 936-945.
5. A. Meixner and D.J. Sorin, ''Error Detection Using Dynamic Dataflow Verification,'' *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT 07), IEEE CS Press, Sept. 2007, pp. 104-118.
6. F.F. Sellers, M.-Y. Hsiao, and L.W. Bearnson. *Error Detecting Logic for Digital Computers*, McGraw Hill Book Company, 1968.
7. A. Meixner and D.J. Sorin, ''Dynamic Verification of Memory Consistency in Cache-Coherent Multithreaded Computer Architectures,'' *Proc. Int'l Conf. Dependable Systems and Networks,* (DSN 06), IEEE CS Press, 2006, pp. 73-82.
8. D. Lampret, OpenRISC 1200 IP Core Specification, rev. 0.7, Sept. 2001, http://www.opencores.org.
9. A. Mahmood and E. McCluskey, ''Watchdog Processors: Error Coverage and Overhead,'' *Proc. 15th Int'l Symp. Fault-Tolerant Computing Systems* (FTCS 85), IEEE Press, 1985, pp. 214-219.
10. J.B. Sulistyo, J. Perry, and D.S. Ha, ''Developing Standard Cells for TSMC 0.25 μm Technology under MOSIS DEEP Rules,'' tech. report VISC-2003-01, Dept. of Electrical and Computer Engineering, Virginia Polytechnic Institute and State Univ., 2003.
11. S.J. Wilton and N.P. Jouppi, ''An Enhanced Access and Cycle Time Model for On-Chip Caches,'' research report 93/5, DEC Western Research Laboratory, July 1994; http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-93-5.pdf.
12. C. Lee, M. Potkonjak, and W.H. Mangione-Smith, ''MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems,'' *Proc. 30th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (MICRO 97), IEEE CS Press, 1997, pp. 330-335.

Albert Meixner is a PhD student in the Department of Computer Science at Duke

University. His research interests focus on architectures and compilers for fault-tolerant computing. He has an MS in applied computer science from Paris Lodron Universität, Salzburg, Austria.

Michael E. Bauer is a BSE student in the Department of Electrical and Computer Engineering at Duke University. His research interests focus on computer architecture and applied mathematics.

Daniel J. Sorin is an assistant professor of electrical and computer engineering and of computer science at Duke University. His research interests focus primarily on dependable computer architecture. He has a PhD in electrical and computer engineering from the University of Wisconsin–Madison.

Direct questions and comments about this article to Daniel J. Sorin, PO Box 90291, Durham, NC 27708; sorin@ee.duke.edu.

For more information on this or any other computing topic, please visit our Digital Library at http://computer.org/csdl.