Achieving Fault Detection and Performance on CMPs

Gordon B. Bell, Member, IEEE, and Mikko H. Lipasti, Member, IEEE

Abstract—**Technology scaling in integrated circuits has consistently provided dramatic performance improvements in modern microprocessors. However, increasing device counts and decreasing on-chip voltage levels have made transient errors a first-order design constraint that can no longer be ignored. Several proposals have provided fault detection and tolerance through redundantly executing a program on an additional hardware thread or core. While such techniques can provide high fault coverage, they at best provide equivalent performance to the original execution and at worst incur a slowdown due to error checking, contention for shared resources, and synchronization overheads. This work achieves a similar goal of detecting transient errors by redundantly executing a program on additional processor cores; however, by sacrificing a small degree of fault coverage, it speeds up (rather than slows down) program execution compared to the unprotected baseline case. This scheme exploits the fact that a small number of instructions are detrimental to overall performance, and selectively skipping them enables some cores to advance far ahead of others and obtain prefetching and large instruction window benefits. We highlight the incremental hardware required and show that reducing fault coverage from 100% to an average of 96%/81% can result in a speedup of 15%/150% for a collection of integer/floating point benchmarks.**

I. Introduction

Prior work has shown that redundant execution on chip multiprocessors is an attractive approach for improving a processor's capabilities for fault detection and fault tolerance, with only a minor impact on performance. However, additional processor cores can also be used to improve the performance of a single thread of execution, by using the resources of the additional cores to buffer slices of miss-dependent instructions and allowing a lead processor to continue running ahead, prefetching cache misses and resolving branch mispredictions. This paper describes a design that integrates these two benefits, providing substantial performance improvements for applications that experience frequent cache misses, while providing nearly the same level of fault coverage as fully redundant execution. In effect, redundant execution is used to achieve two goals that prior work has pursued: improved performance and improved fault tolerance. In order to achieve these goals, a baseline CMP must be augmented with several structures that are used to maintain correct sequential execution semantics while also detecting faults that may occur. This paper provides a brief description of these structures and their operation, and reports the measured fault coverage and performance achieved by the proposed approach.

II. Prior Work on SRT/CRT

Simultaneous and redundantly threaded (AR-SMT or SRT) processors execute an identical copy of a program on an unused hardware thread or processor core [14],[13]. Errors can be detected by buffering retired stores in a store comparator queue where they are compared to identical stores executed on a second thread. If a mismatch is detected in either a store's data or address, an error is signaled to the processor so that it can respond appropriately. Combining different instances of the same dynamic store presents the appearance to the memory system that a single thread is running and maintains uniprocessor program semantics. Chip-level redundant threading (CRT) [11] extends this concept to run multiple instances of the same thread on different cores in a chip multiprocessor (CMP) rather than on separate thread contexts in a simultaneous multithreaded processor (SMT). Depending on the degree of hardware replication in an SMT, CRT is able to provide better performance than SRT by removing contention for resources shared among threads (such as the physical register file, issue window, ROB, load and store queues, fetch unit, execution units, etc.). However CRT's performance is still limited to that of the original unreplicated thread, and in the worst case it incurs a slowdown due to the additional checking overhead and synchronization between cores.

III. Profligate Execution

SRT and CRT check all memory system updates and can therefore provide 100% transient error coverage within the sphere of replication. However, while all instructions may be equally likely to produce an error, they do not contribute equally to overall performance. In particular, L2 data cache misses are inordinately expen-

sive, as even aggressive dynamically scheduled microarchitectures are not able to sufficiently cope with increasing relative memory latency. Therefore when an L2 cache miss occurs in a CRT processor, both cores stall and fail to perform any useful work while waiting for the memory access to return the data.

This work builds on the CRT philosophy of detecting faults through redundant program execution on unused CMP processor cores, but addresses CRT's overhead of increased execution time. It proposes to force only a single core in a CMP to stall and wait for L2 cache miss to complete, and allows the second processor to advance ahead by dropping the miss and its forward slice of dependent instructions. Those instructions executed by a single core cannot be checked for correctness; however, because they are not inhibiting forward progress, the processor that dropped them can continue and prefetch data into the shared L2. It operates as follows:

When an L2 cache miss occurs one core stalls and becomes responsible for waiting for the miss to complete. The second core immediately marks the miss as completed and sets an INV bit associated with its result registers. Any instructions that consume the result of the miss inherit the INV bit are also immediately marked completed and set the INV bit associated with their result register. This mechanism of identifying the forward slice of cache misses has been used extensively in a variety of other microarchitectural optimizations [5],[12],[17],[7],[20],[2]. Eventually a second L2 cache miss is uncovered and again a core is assigned to wait for it. This partitions cache misses between the cores and has the dual benefit of reducing the total number of stalls experienced by each, as well as allowing one to prefetch data for the other.

A. Register Communication

Aside from error checking by the store comparator, CRT requires no communication between cores since both cores execute all instructions. This is not true in profligate execution, as register values that depend upon L2 cache misses are not available in all cores. Although executing most instructions everywhere drastically reduces communication requirements, there are several situations that require register transfers.

•*Slice Joins:* If an instruction's source operands depend on different slices executed by different cores, operands need to be communicated such that at least one core has all source registers and can execute the instruction.

•*Control Flow:* Branches with a target or direction that depends on a miss need to be resolved before committing. Therefore any miss-dependent registers that directly feed a branch must be communicated to all other processors.

•*Precise Exceptions:* Because no processor executes every instruction, architected state is distributed across multiple cores. When an exception occurs, we must ensure that precise state can be reconstructed and execution can be restarted at the excepting instruction. This requires buffering and transferring miss-dependent register values between cores prior to branching to operating system exception exception handling routines.

Reorder buffers (ROBs) have long been used by dynamically-scheduled microarchitectures as a mechanism for buffering and communicating values, and we adopt a similar approach here. As processors commit miss-dependent instructions they insert their results into a Global Reorder Buffer (GROB), which other cores can retrieve values from. The GROB is implemented as a RAM, and each core maintains a tail pointer that corresponds to the insertion point for its oldest slice instruction. Each core also maintains a Global Register Alias Table (GRAT) that indicates the GROB index of the most recent miss-dependent producer of every architected register value. When an instruction needs to retrieve a value due to one of the above conditions, it waits until it becomes the oldest in its respective core, then indexes the GRAT to determine the GROB entry that will contain the source register value. Finally, it uses that index to read the missing operand from the GROB and executes the instruction. When all cores have advanced beyond a slice instruction the output value is no longer required and the GROB entry is deallocated. This results in efficient and infrequent communication of values between cores.

GROB accesses are efficient because each core privately maintains a set of indexes into it. This avoids costly associative searches or centralized lookup tables. However it requires that each core has a consistent view of how slice instructions map to GROB entries. Specifically, cores need to agree on which instructions are miss-dependent so that they can increment their GROB tails at exactly the same points in the dynamic instruction stream. Here we detail one such method to accomplish this.

The first processor to execute an L2 load miss allocates a miss status handling register (MSHR), marks the load destination register INV, and discards that instruction. As long as the miss is still outstanding when the second core executes the same load, that core will also correctly identify the load as an L2 miss. However if the second processor reaches the same load much later, the miss could have already completed and updated the L2 cache. In this case the second core would identify the load as a hit, and will not increment its GROB tail when it commits the load. This will cause each core's GROB indexes to shift by one entry, and will result in incorrect

TABLE I. MACHINE CONFIGURATION

Parameter	Value
Out-of-order execution	4-wide fetch/issue/commit, 10-cycle pipeline, 64 ROB, 32 LSQ, 64 rename registers
Functional Units	4 integer ALU, 2 FP ALU, 2 integer MULT/DIV, 2 FP MULT/DIV, 2 memory ports
Branch Prediction	Combined bimodal (16k entry) / gshare (16k entry) with selector (16k entry), 16 RAS, 1k entry 4-way BTB
Memory System (latency)	32KB 2-way 32B line IL1 (2), 8KB 4-way 16B line DL1 (2), 512 KB 4-way 64B line unified L2 (15), main memory (500), hardware prefetcher

communication of register values between cores.

To deal with this problem we must maintain the invariant that all cores identify the same instructions as falling within the miss slice regardless of whether they miss the L2 cache or not. When an L2 miss completes, the L2 MSHR is marked with the GROB index of the youngest miss in program order to that line and de-allocating the MSHR and updating the L2 is delayed until all cores have reached that instruction. Conceptually, any loads older than the marked index should be identified as a miss, and any loads younger than that index should be identified as a hit. Delaying updating the L2 cache forces subsequent L2 accesses to that address to access the MSHR and perform this age comparison. L1 MSHRs need to be similarly marked so that subsequent L1 hits can know when GROB tail increments are required. Note that this scheme does not prevent intervening references to the block from being satisfied as hits; they are, but the GROB tail pointer is incremented in order to maintain a consistent view of slice instructions across the whole machine.

B. Memory Dependences

Even though individual cores execute instructions out of program order, memory instructions must still appear to complete in order and adhere to uniprocessor program semantics. Specifically, loads must receive the value written by the most recent store to their address, and stores must update the memory hierarchy in program order. These requirements are fulfilled in a traditional uniprocessor via a FIFO store queue. Stores exit the store queue and update the L1 at retirement, and loads search the store queue to identify older in-flight stores to the same address from which they need to receive data. Profligate execution adopts a similar approach to enforce in-order updates of the shared L2 and correct disambiguation of memory addresses between cores. All store addresses must be available at all cores, and are communicated through the GROB if necessary. Most store values are available locally and can be forwarded either through the local store queue or local data cache. Whenever values are not available, the local block is invalidated (write-invalidate policy), forcing subsequent dependent loads to fetch their values from the second-level cache or the global store queue, if necessary. More details regarding the implementation and complexity of the memory interface are in [4].

IV. RESULTS

A. Methodology

This section presents data indicating the increase in performance and corresponding decrease in error coverage measured in a detailed cycle-counting simulator that models profligate execution, including all associated hardware structures detailed in [4].

SPEC2000 integer and floating point benchmarks were compiled for the Alpha instruction set with peak optimization by the DEC OSF optimizing compiler, and executed with the reference input sets. Benchmarks were fast forwarded 1 billion instructions before timing simulation was collected on the following 10 billion instructions using the SMARTS sampling methodology [19]. All no-ops are removed from execution and consume no processor resources. Table 1 presents the machine model configuration used to collect data.

B. Speedup

Figure 1 shows the average number of instructions committed per cycle for three machine configurations. The first bar corresponds to the baseline case of a single core. The second bar shows the IPC when independent copies of the program are run on two cores in a profligate execution system. Executing a program on two cores and partitioning the L2 cache misses between them results in an average speedup of 15% and 150% for the integer and floating point benchmarks, respectively. Applications with low baseline IPCs due to high L2 miss rates, such as several of the floating point benchmarks, can achieve considerable speedup by enabling an additional core to overlap subsequent misses.

The final bar shows performance if four cores execute the program instead of two. Partitioning cache misses among more cores can achieve additional speedup because it allows each core to discard more misses and stall less frequently. While two cores are generally sufficient to expose additional memory-level parallelism, some benchmarks with a large number of L2 misses can benefit from additional cores. For example, increasing the number of cores from two to four results in an average speedup from 150% to 211% in the floating point benchmarks.

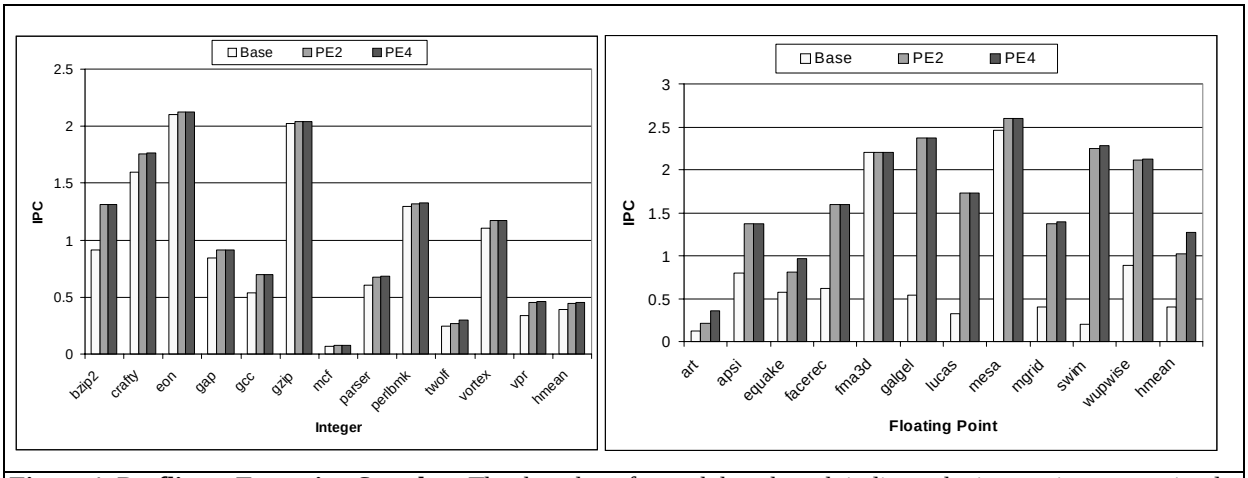

Figure 1. Profligate Execution Speedup. The three bars for each benchmark indicate the instructions committed per cycle for the base configuration and two and four core profligate execution.

C. Error Coverage

The previous section showed that significant performance gains are possible by preventing all cores from stalling on all L2 cache misses. However, forcing a subset of a program's instructions to execute on a single core sacrifices some degree of redundancy and soft error coverage. If a miss slice instruction handled by a single core executes incorrectly, its dependant stores will not be compared to those of the remaining core and any errors may go undetected. This section quantifies the reduction in error coverage.

Figure 2 graphs the fraction of all dynamic instructions executed by both cores. Single-bit errors in this subset of instructions will be detected in the store comparator queue similar to SRT/CRT. The remaining instructions are not executed redundantly because they depend on L2 misses and are therefore vulnerable to transient faults.

Despite their high cost, L2 cache misses occur relatively infrequently, and most instructions do not depend on them and execute redundantly. On average, 96% of all dynamic instruction in the integer benchmarks are executed by both cores and are therefore protected from soft errors. If *mcf* is not considered, over 99% of all instructions are covered. The higher L2 miss rates exhibited by the floating point benchmarks result in fewer globally-executed instructions. While this can enable dramatic speedups, it also decreases effective error coverage. On average, 81% of all instructions in the FP benchmarks are redundantly executed. One particular benchmark, *galgel*, has an unusually high fraction of miss-dependant instructions; if it is eliminated, the FP average jumps to nearly 90%. The fact that *galgel* has relatively few L2 cache misses indicates that at least some misses have long chains of dependent instructions. Periodic synchronization of miss-dependant registers between cores can force redundant execution, however this increased error coverage may come at the expense of slower execution time. We point out that this data corresponds to a design that partitions all L2 cache misses, and therefore indicates upper and lower bounds on performance and error coverage, respectively. However, a continuum of design options exist that balance these variables against each another. While we leave a quantitative evaluation of such designs to future work, we believe this flexibility increases the attractiveness of this technique.

D. Additional Cores

SRT/CRT is able to achieve 100% error detection coverage by executing an additional copy of a program on one other core. Profligate execution, on the other hand, provides less than 100% coverage but can decrease execution time. Figure 1 showed that additional available cores in a CMP can be used to further reduce execution time. However, we point out here that they could alternately be used to increase error coverage. For example, if three cores are available to run a single program thread, always assigning two of them to wait for every L2 cache miss will obtain all the speedup of two cores, as well as provide 100% error detection coverage. Furthermore, such a design not only provides error detection, but can provide forward error recovery as well. Because the majority of instructions are miss-independent and execute on all three cores, triple modular redundancy can correct the error to the value computed by the majority of cores. This offers a significant improvement over SRT's ability to only detect errors, as the presence of an error is not known until it propagates to a store. At that point it may have been delayed so long that the program cannot gracefully recover.

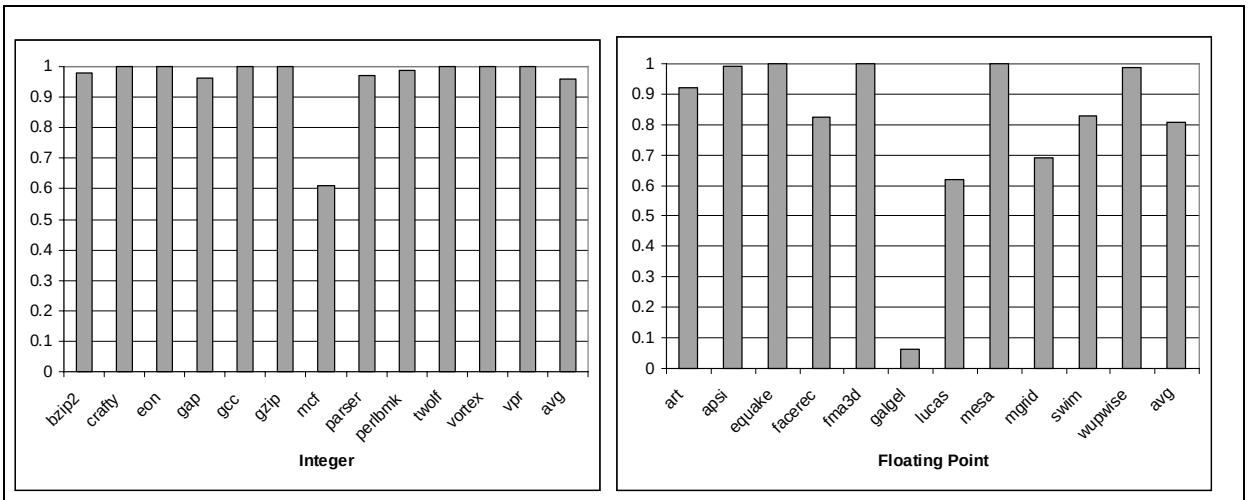

Figure 2. Profligate Execution Error Coverage. Fraction of all instructions executed by multiple cores.

V. Related Work

Several papers have proposed using additional cores in CMPs to provide transient fault coverage [14][13],[11],[8],[15],[1], often at the expense of increased execution time. Others have focused on using additional CMP cores to increase single-thread performance [16],[9],[10],[18],[20],[7]. However by spreading program computation across more transistors, these techniques effectively increase an application's vulnerability to transient errors. The notion of adding microarchitectural support for selectively skipping miss-dependant instructions was first proposed by Dundas [5], and later elaborated on by several others [12],[3],[17],[6]. Such designs tend to require significant changes to established baseline microarchitecture, and thereby complicate validation effort in addition to consuming additional chip real estate that could otherwise be used for additional CMP cores.

This work combines many of these ideas to simultaneously provide both fault tolerance and decreased execution time by utilizing mostly unmodified processor cores. To our knowledge, it is the first proposal to provide a solution for both of these barriers to high performance processor design.

VI. Conclusion

This work extends prior research of executing identical programs on multiple cores to detect transient errors. It makes the observation that not all instructions have the same impact on system performance and selectively discarding them can realize significant performance gains while giving up disproportionately small degrees of error coverage. Two system design trends accentuate this point: larger L2 (and L3) caches will decrease the fraction of miss-dependent non-redundant instructions. At the same time, increasing relative memory latency will make these misses more expensive in terms of performance, and skipping them more attractive. While several proposals have attacked the problems of soft errors and increasing relative memory latency in isolation (and often at the expense of each other), this is the first to simultaneously achieve the benefits of both.

Acknowledgments

This research was supported in part by the National Science Foundation under grants CCR-0133437 and CCF-0429854, as well as grants and equipment donations from IBM and Intel.

References

[1] Todd Austin. Diva: A reliable substrate for deep-submicron processor design. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-32)*, Los Alamitos, November 30–December 2 1999. IEEE Computer Society.

[2] Ronald D. Barnes, Erik M. Nystrom, John W. Sias, Sanjay J. Patel, Nacho Navarro, and Wen mei W. Hwu. Beating in-order stalls with "flea-flicker" two-pass pipelining. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2003.

[3] Ronald D. Barnes, Shane Ryoo, and Wen mei W. Hwu. "flea-flicker" multipass pipelining: An alternative to the high-power out-of-order offense. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 319–330, Washington, DC, USA, 2005. IEEE Computer Society.

[4] Gordon B. Bell and Mikko H. Lipasti. Profligate execution. Technical report, 2006. Available from http://www.ece.wisc.edu/~pharm.

[5] James David Dundas. *Improving processor performance by dynamically pre-processing the instruction stream*. PhD thesis, 1998. Chairman-Trevor Mudge.

[6] Amit Gandhi, Haitham Akkary, Ravi Rajwar, Srikanth T. Srinivasan, and Konrad Lai. Scalable load and store processing in latency tolerant processors. *isca*, 00:446–457, 2005.

[7] Ilya Ganusov and Martin Burtscher. Future execution: A hardware prefetching technique for chip multiprocessors. In *PACT '05: Proceedings of the 14th International Con-*

ference on Parallel Architectures and Compilation Techniques, pages 350–360, Washington, DC, USA, 2005. IEEE Computer Society.

[8] Mohamed Gomaa, Chad Scarbrough, T. N. Vijaykumar, and Irith Pomeranz. Transient-fault recovery for chip multiprocessors. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 98–109, New York, NY, USA, 2003. ACM Press.

[9] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 58–69, New York, NY, USA, 1998. ACM Press.

[10] Venkata Krishnan and Josep Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Trans. Comput.*, 48(9):866–880, 1999.

[11] Shubhendu S. Mukherjee, Michael Kontz, and Steven K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 99–110, Washington, DC, USA, 2002. IEEE Computer Society.

[12] O Mutlu, J Stark, C Wilkerson, and YN Patt. Runahead execution: an alternative to very large instruction windows for out-of-order processors. In *Proceedings of HPCA-9*, January 2003.

[13] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient fault detection via simultaneous multithreading. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 25–36, New York, NY, USA, 2000. ACM Press.

[14] Eric Rotenberg. Ar-smt: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the 29th Fault-Tolerant Computing Symposium*, June 1999.

[15] Jared C. Smolens, Brian T. Gold, Babak Falsafi, and James C. Hoe. Reunion: Complexity-effective multicore redundancy. In *MICRO 39: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, Washington, DC, USA, 2006. IEEE Computer Society.

[16] Gurindar S. Sohi, Scott E. Breach, and T.N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[17] Srikanth T. Srinivasan, Ravi Rajwar, Haitham Akkary, Amit Gandhi, and Mike Upton. Continual flow pipelines. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 107–119, New York, NY, USA, 2004. ACM Press.

[18] J. Steffan and T Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, page 2, Washington, DC, USA, 1998. IEEE Computer Society.

[19] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. Smarts: accelerating microarchitecture simulation via rigorous statistical sampling. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 84–97, New York, NY, USA, 2003. ACM Press.

[20] Huiyang Zhou. Dual-core execution: Building a highly scalable single-thread instruction window. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 231–242, Washington, DC, USA, 2005. IEEE Computer Society.

Gordon Bell is a Ph.D. candidate in the department of Electrical and Computer Engineering at the University of Wisconsin. He has a B.S. in computer engineering from the University of Notre Dame and an M.S. in electrical and computer engineering from the University of Wisconsin. His research interests include several aspects of high-performance microarchitecture and simulation methodology, with an emphasis on developing new techniques to mitigate increasing relative memory latency. He is part of the performance modelling and analysis group for next-generation systems at IBM, and is a member of the ACM and IEEE.

Mikko Lipasti is an associate professor of Electrical and Computer Engineering at the University of Wisconsin. He earned his Bachelor of Science degree in computer engineering from Valparaiso University in 1991, and his M.S. and Ph.D. degrees in Electrical and Computer Engineering from Carnegie Mellon University in 1992 and 1997. Prior to joining the University of Wisconsin, he spent several years with IBM, helping develop software and hardware for PowerPC servers, and he currently consults actively with Intel Corporation. Eta Kappa Nu named him the 2002 Outstanding Young Electrical Engineer, and he received the Gerald Holdridge Excellence in Teaching Award in 2006. He holds several patents, has co-authored a popular graduate-level textbook on computer architecture, and has published over 50 peer-reviewed technical papers in journals, conferences, and workshops. His primary research interests include complexity- and power-efficient processors, memory hierarchies, and cache coherence protocols that don't compromise delivered performance.