

Lazy Error Detection for Microprocessor Functional Units

Mahmut Yilmaz¹, Albert Meixner², Sule Ozev¹, and Daniel J. Sorin¹

¹Dept. of Electrical and Computer Engineering

Duke University

{my, sule, sorin}@ee.duke.edu

²Dept. of Computer Science

Duke University

albert@cs.duke.edu

Abstract

We propose and evaluate the use of lazy error detection for a superscalar, out-of-order microprocessor's functional units. The key insight is that error detection is off the critical path, because an instruction's results are speculative for at least a cycle after being computed. The time between computing and committing the results can be used to lazily detect errors, and laziness allows us to use cheaper error detection logic. We show that lazy error detection is feasible, we develop a low-cost mechanism for detecting errors in adders that exploits laziness, and we show that an existing error detection scheme for multipliers can exploit laziness.

1. Introduction

Our goal is to develop and evaluate error detection mechanisms for a microprocessor's functional units (FUs), such as adders and multipliers. We want our error detectors to have three features: negligible impact on microprocessor performance, low hardware cost, and low power consumption. It is obviously sufficient to replicate the FUs, but this simple solution is very costly. Other error detection mechanisms have been developed previously, as we discuss in Section 2, but we will explain why they either degrade performance or use more hardware and power than our approach. Our error detection mechanisms can be used in conjunction with error detection mechanisms for other portions of the microprocessor.

The key to achieving our goal is exploiting laziness. We observe that detecting an error in an FU is generally not on the critical path in a microprocessor's execution. In a dynamically scheduled ("out-of-order") microprocessor, such as the Pentium4 [3] or POWER5 [7], an instruction produces a result that can be consumed immediately by subsequent instructions in program order. However, the instruction's results are still speculative for one or more cycles, while the microprocessor waits for older instructions to commit and checks if the instruction is on a correctly speculated path (i.e., it does not depend on an incorrectly predicted branch). In this time window, the microprocessor can lazily detect errors in the instruction's computation. If an error is detected, the speculative state created by this instruction and its dependent instructions is squashed. If no error is detected, the speculative state is committed. In general, performance is not affected because the critical path—providing the results of each instruction to its dependent instructions—is not affected. The only potential source of slowdown is that we might extend the amount of time in which an instruction is speculative and thereby increase contention for resources that hold speculative state. We make the following three contributions in this paper:

- We quantify the ability of a modern microprocessor to tolerate lazy error detection in its FUs. We show that we can tolerate several cycles of laziness with little impact on performance. This result enables us to trade latency for hardware and power savings.
- We develop a new lazy error detection mechanism that applies to all types of adders, and we show that it uses less hardware and power than comparable checkers.

- We evaluate the feasibility of using modulo arithmetic to lazily detect errors in multipliers. The error detection mechanism is not novel, but the sensitivity analysis is new and demonstrates that lazy checking is not constrained just to adders.

In Section 2, we present the related research in error detection. In Section 3, we describe the experimental methodology we use to evaluate the performance and power consumption of our designs. In Section 4, we quantify the opportunity to use lazy error detection. In Section 5, we develop and evaluate our lazy adder checker. In Section 6, we evaluate lazy error detection for multipliers. In Section 7, we conclude.

2. Related Work in Error Detection

There is a long history of work in detecting errors in functional units, including checkers for specific functional units as well as more comprehensive approaches.

2.1. Functional Unit Checkers

Sellers et al. [15] present a survey of error detection techniques for functional units of all kinds, including different types of adders and multipliers. These techniques include simple replication of the functional units as well as time redundancy techniques. Physical replication uses at least 100% extra hardware (including the comparator), and time redundancy techniques consume a considerable amount of extra energy. Pure time redundancy techniques cannot detect errors due to permanent faults, although clever approaches like re-execution with shifted operands (RESO) [12, 13] can. Some time redundancy approaches add latency, but this latency can often be partially hidden by exploiting idle resources. All time redundancy approaches reduce throughput, because a unit must be used multiple times per operation. Sellers et al. also survey structure-specific checkers like parity predictors, arithmetic error codes, and dual-rail logic designs. Most of these techniques strive for near-instantaneous error detection, because they consider detection to be on the critical path. Achieving such fast detection often leads to more expensive hardware and/or more power consumption than if we can exploit lazy detection.

Adder Checkers. The most closely related self-checking adder is in the seminal paper by Nicolaidis [10]. Nicolaidis presents self-checking versions of several types of adders, including carry lookahead. His approaches use double-rail and parity codes and thus slow down the adder. We show in our experimental results that our approach has no impact on the adder’s performance. The key is that our adder checker is distinct from the primary adder and thus does not increase the primary adder’s wire lengths, fanouts, or capacitance. There are also numerous self-checking adder schemes that only pertain to specific types of adders. For example, there are self-checking techniques for lookahead adders [8], carry select adders [17, 18, 20], and there are self-checking adders that use signed-digit representations [4]. Our approach to designing self-checking adders applies to any type of adder (e.g., ripple carry, carry lookahead, etc.). Townsend et al. [19] develop a self-checking and self-correcting adder that combines TMR and time redundancy. For a 32-bit adder, this scheme has 36% area and 60% delay overhead. Power overhead is not reported. It is somewhat difficult to compare our work to this scheme because we rely on the microprocessor’s built-in recovery mechanism to provide error correction.

Multiplier Checkers. We do not claim to innovate in the area of self-checking multipliers. We instead rely upon modulo arithmetic checking for detecting errors, using a checker that is similar to that presented recently by Yilmaz et al. [22].

2.2. Comprehensive Error Detection

One approach for comprehensive error detection is redundant multithreading [14, 9, 21]. The redundant threads exploit time redundancy to detect errors across the microprocessor, including in its functional units. These schemes suffer a performance opportunity cost: non-redundant

Table 1. Parameters of target system

Feature	Value
pipeline stages	20
width: fetch/issue/commit/check	3/6/3/3
branch predictor	2-level GShare, 4K entries
instruction fetch queue	64 entries
reservation stations	32
reorder buffer (ROB) and load-store queue (LSQ)	128-entry ROB, 48-entry LSQ
integer functional units	3 ALUs (1-cycle), 1 mult/div (14-cycle mult, 60-cycle div)
floating point functional units	2 FPUs (1-cycle), 1 mult/div (1-cycle mult, 16-cycle div)
L1 I-Cache and L1 D-Cache	16KB, 8-way, 64B blocks, 2-cycle
L2 cache (unified)	1MB, 8-way, 128B blocks, 7-cycle

work that could have been done with those extra thread contexts. The redundant threads can also interfere with the primary thread and thus degrade its performance; among the published redundant multithreading schemes, we observe as much 30% performance degradation [9]. The hardware cost of redundant multithreading depends on whether one considers the hardware for the extra thread contexts to have been included for purposes of error detection. The expected power overhead of redundant multithreading is substantial because each instruction is not only executed twice, but also traverses the whole pipeline from the beginning to the end.

Another general approach is DIVA [2], which detects errors by checking an aggressive out-of-order core with simple in-order checker cores that sit right before the aggressive core’s commit stage. Because DIVA and our scheme have different features, we try to balance the comparison by only considering the DIVA checker’s adder; we do not consider the other aspects of DIVA, since they are orthogonal. Hardware overheads for the DIVA checker adder and checker multiplier are both over 100%, because they replicate the FU and also need some comparison logic. For the adder, including the comparator circuit, the expected area overhead is over 115%. The expected power overhead is similar. Thus, our scheme has significantly lower error detection costs than DIVA’s error detection (or error detection with simple FU replication).

3. Experimental Methodology

We have two goals for our experimental evaluations. First, we want to show that lazy error checkers do not degrade microprocessor performance. Second, we want to evaluate the hardware and power costs of lazy checkers and compare them to prior work.

Microprocessor Performance. We used a modified version of SimpleScalar [1]. Table 1 shows the details of our configuration, which was chosen to be similar to that of the Intel Pentium4 [5, 3]. One important point is that our configuration models a 14-cycle multiplier (similar to the multiplier in the Pentium 4), rather than the 4-cycle multiplier we develop in this paper. The 4-cycle pipeline depth for our transistor-level multiplier was chosen to match the clock period of our transistor-level adder design, which takes 1 clock cycle to complete. We believe the 14-cycle multiplier latency is more realistic for microprocessor-level simulation, but a 14-cycle multiplier would have had a clock frequency higher than what we could achieve for the adder, without an industrial design team and CAD infrastructure. We will discuss, when necessary, how the 14-cycle multiplier affects our experimental results.

For benchmarks, we use the complete SPEC2000 benchmark suite with the reference input set. To reduce simulation time, we used SimPoint analysis [16] to sample from the execution of

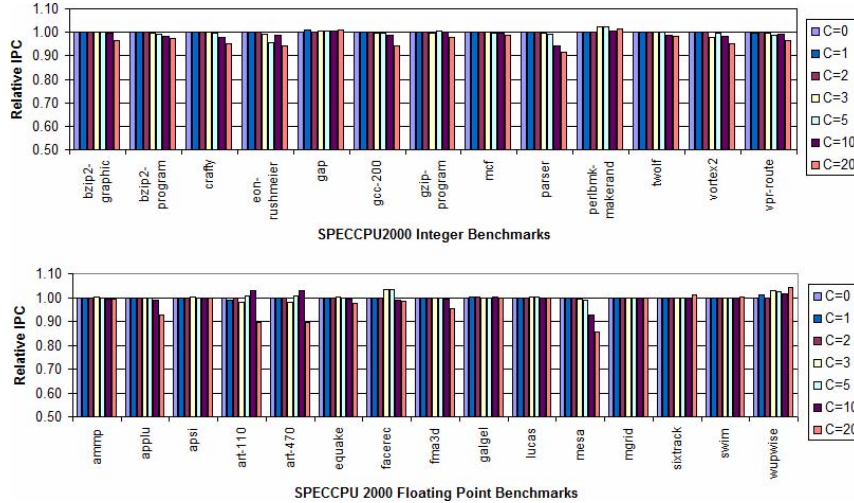


Figure 1. Impact of adding C cycles between instruction completion and commit

each benchmark. We warm up the simulator with 100-million instructions of detailed simulation prior to the sample, and then we simulate the 100-million instruction sample.

Hardware and Power Costs. We use HSpice to evaluate the delay and power consumption of our circuits, and we assume a $0.13\mu\text{m}$ CMOS process. Because we cannot possibly simulate *every* possible input transition for the adder and multiplier and because HSpice simulations take so long, our power results are based on 3000 HSpice simulations with different random input transitions. The power results are averaged across all of the time for all of the simulations for each FU, and they include both dynamic and static (leakage) power (although dynamic power dwarfs static power at $0.13\mu\text{m}$). We approximate the hardware cost of a circuit in terms of the number of transistors in the circuit.

4. Opportunity to Use Lazy Checkers

We leverage the opportunity to lazily perform error detection in the time between when an instruction completes (produces its result) and when it commits. Instruction completion is often on the critical path, since an instruction’s result is often the input to one or more subsequent instructions. Instruction commit, however, is only critical if the reorder buffer (ROB) or load-store queue (LSQ) fills up. To explore the potential to exploit this latency tolerance, we performed the following experiment on the microprocessor described in Section 3). We added C cycles between when each instruction completed and when it could commit. For each instruction, some or all of these C cycles could be hidden, because the instruction might not have been able to commit before then anyway (due to older instructions still waiting to commit). The results of this experiment, shown in Figure 1, reveal a surprisingly high tolerance to even large values of C , which indicates that there is plenty of time in which we can perform error detection. For $C=3$, the worst slowdown is less than 1.5%, and for $C=5$, the worst slowdown is less than 4%. Note that increasing C can, in some cases, slightly *improve* performance. This counter-intuitive phenomenon is due to slightly reduced misprediction rates in these scenarios (data not shown due to space constraints).¹

1. Sometimes, when a load gets delayed (due to increased C), it just so happens to then get disambiguated correctly (where it would have mispredicted without the delay). This is a lucky and fairly rare event.

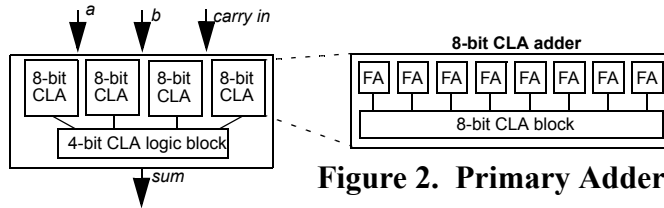


Figure 2. Primary Adder

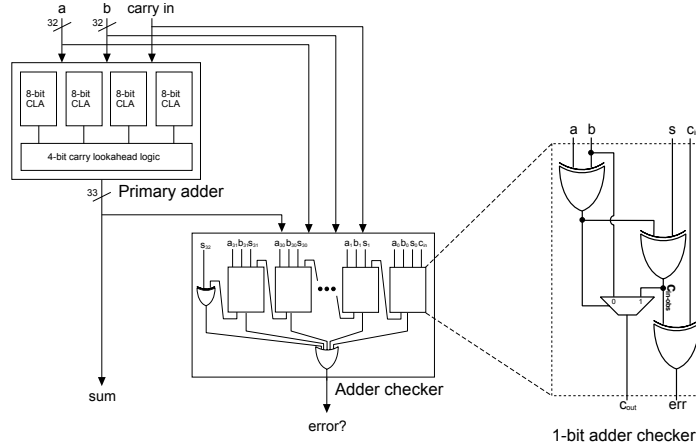


Figure 3. Adder Checker

This latency tolerance enables us to use lazy checkers that are slow and can thus use less hardware and power than the high-performance FUs they are checking. The key is maintaining checker throughput—the checkers can have long latencies, but they must check computations at the same bandwidth as they are being produced by the primary FUs. Otherwise, simple queuing theory shows that the checkers will become a performance bottleneck. Thus, we must have a dedicated checker for each FU, and our lazy checkers must either be 1-cycle or fully pipelined.

5. Lazy Adder Checker

Before we describe the adder checker, we first present the details of the high-performance primary adder that it is checking. We have implemented both the primary adder and the adder checker in HSpice using 0.13 μ m process parameters.

5.1. Primary High-Performance Adder

The primary adder is a 32-bit, two-level carry look-ahead (CLA) adder, as illustrated in Figure 2. We have selected the primary adder to be a CLA adder because it is fast and it is widely used in commodity microprocessors. The lower level of the structure is comprised of 8-bit CLA adders, which each include eight full adders (FA) and an 8-bit CLA logic block. At the higher level, four of these 8-bit adders are connected by a 4-bit CLA logic block.

5.2. Adder Checker and Operation

Our adder checker, shown in Figure 3, detects all single errors. It takes the two addends (A and B), carry-in, and sum from the primary adder and determines whether the primary adder's sum is correct. The checker is modular, in that it is composed of 32 1-bit adder checker *cells*. Each cell produces an error signal, and all of these error signals are ORed together to determine if there is an error in the 32-bit addition. For each cell, C_{in} and C_{out} are the carries computed by the checker, except for bit zero where C_{in_0} is the carry-in to the primary adder.

The i^{th} checker cell first determines the carry-in bit used to compute the i^{th} sum bit during the primary addition by comparing the sum of bits A_i and B_i to the corresponding bit sum_i . This *observed carry-in* ($Cin-obs_i$) bit, which is computed as $A_i \text{ xor } B_i \text{ xor } sum_i$, is used to compute the carry-out bit $Cout_i$. It is also compared to cell i 's carry-in bit Cin_i , which is the carry-out bit of the $(i-1)^{\text{th}}$ checker cell, $Cout_{i-1}$, to detect errors. If the sum computed by the primary adder is error-free, $Cin-obs_i$ and Cin_i will be equal. If at least one bit in the sum is incorrect and the least significant sum bit with an error is j , then checker cell j will detect an error. For all less significant bits $0..j-1$, the observed carry-in bits ($Cin-obs_0..Cin-obs_{j-1}$) are correct (i.e., are equal to the carry-in bits used during a correct computation of the sum) and therefore the carry-out bits $Cout_0..Cout_{j-1}$ are correct (assuming a correct checker). Thus, Cin_j , which equals $Cout_{j-1}$, will also be correct. Because bit j of the sum is incorrect, the j^{th} observed carry-in bit, $Cin-obs_j$, is incorrect and will not equal Cin_j . This mismatch causes an error to be signalled. The carry-out bit of cell j ($Cout_j$) and all more significant cells are no longer guaranteed to be correct, because they are computed from incorrect $Cin-obs$ bits. This, however, does not matter, because an error has already been signalled. Note that our error detection mechanism detects *all* errors in the primary adder, not just errors in its carry computation circuitry. If full adder i in the primary adder computes an incorrect sum, then checker cell i will see an incorrect $Cin-obs_i$.

$Cout_i$ is computed from $Cin-obs_i$ using a circuit that is commonly used for computing carries in full adders. If A_i and B_i are equal ($A_i \text{ xor } B_i = 0$), there will be a carry-out iff $A_i = 1$ (or $B_i = 1$, but because they are equal it does not matter). If A_i and B_i are not equal ($A_i \text{ xor } B_i = 1$), there will be a carry-out iff there is a carry-in. The mux is used to distinguish these two cases. Because Cin_i is not used to compute $Cout_i$, there is no dependence chain between cells, although the middle portion of Figure 3 might imply this.

Because there is no dependence chain between cells, the adder checker's computation of the *Error* signal requires only one additional cycle beyond that required for the primary adder. It actually takes less than a full cycle, but the clock period is the minimum granularity. The adder checker's delay (0.65ns) is significantly less than that of the primary adder (1.08ns), which is a 1-cycle unit. Thus, we do not have to pipeline the adder checker. Most of the 0.65ns is consumed by ORing together the outputs of each checker cell. The cells themselves have short critical paths, and there is no dependence between cells.

5.3. Experimental Evaluation

To study the impact of lazy error detection for adders, we added C cycles between when an instruction that used an integer adder (add, sub, load, store, branch) completed and when it can commit. Our particular adder checker uses one extra cycle between complete and commit (i.e., $C=1$), but we explored other values of C to determine the sensitivity. Across the benchmarks, 3-40% of all dynamic instructions used an integer adder. In Figure 4, we show the error-free performance impact of lazy adder checkers, and we observe that, for $C=1$, there is no noticeable performance impact. There is also an opportunity to exploit at least two more cycles without performance loss, but we have not found an adder checker that can make use of that opportunity.

Table 2. Hardware and Power Overheads for Adder

module	size (#transistors)	percentage of primary adder	average power	percentage of primary adder
primary adder	3488	100%	2.38mW	100%
adder checker	1108	32%	0.97mW (error-free)	41%
			1.04mW (worst-case errors)	44%

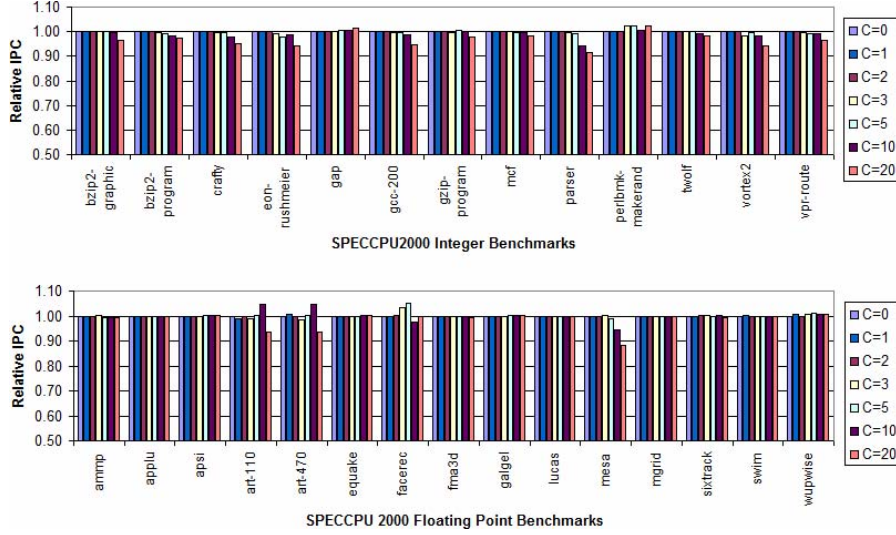


Figure 4. Error-Free Performance Impact of Lazy Adder Checkers

As can be seen in Table 2, the adder checker is inexpensive. In total, it uses 1108 transistors, which is 32% the size of the primary adder and far less than replication. Of these transistors, 960 are used to compute the 32 1-bit checks, and 148 comprise the logical OR-tree to detect if any of the bit checks detect an error. Power overhead is somewhat greater than hardware overhead.

6. Lazy Multiplier Checker

As another case study, we evaluate the feasibility of using lazy error detection for multipliers. Because Yilmaz et al. [22] already presented a study of a self-checking multiplier, we present this section for completeness rather than as a brand new contribution. We want to confirm that lazy error detection does not only apply to adders. One contribution of this section is that we present an analysis of how sensitive performance is to C . Another contribution is that we are studying a Booth encoded primary multiplier instead of the recursive design of Yilmaz et al.

6.1. Primary High-Performance Multiplier

The primary multiplier is a pipelined 32-bit modified Booth encoded integer multiplier. We have selected the primary multiplier to be a Booth encoded multiplier because, due to its fast operation, it is widely used in commodity microprocessors [6, 11]. This multiplier is divided into 4 pipeline stages, and it thus calculates a multiplication result in 4 clock cycles.

6.2. Multiplier Checker

We refer the interested reader to Yilmaz et al. [22] for details on modulo checking of a multiplier in a microprocessor. Due to the different primary multiplier designs, the error coverage for our checker is somewhat different than Yilmaz’s coverage. For our Booth encoded multiplier and a modulus of 3, the probability of not detecting an error is 2.2%. For a modulus of 7, this probability is 1.4%.

6.3. Experimental Evaluation

In Figure 5, we show the error-free performance impact of lazy multiplier checkers for various values of C . Our multiplier checker has a value of $C=1$ for the 4-cycle primary multiplier, but it is likely to be 2 or 3 cycles for a 14-cycle primary multiplier (in a system with a faster clock), since computing $(A*B)\%3$ and comparing it to $[(A\%3)*(B\%3)]\%3$ will take more than a

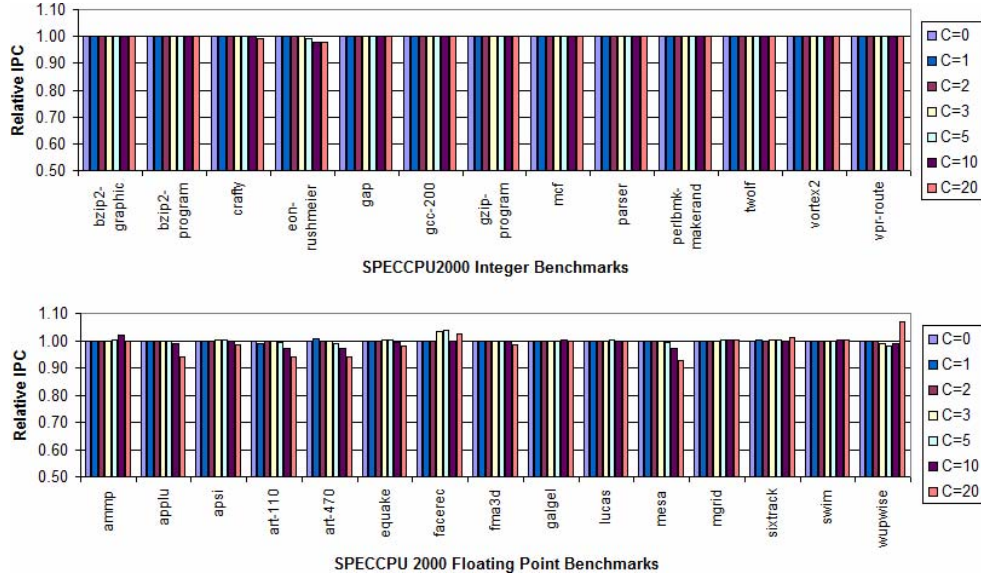


Figure 5. Error-Free Performance Impact of Lazy Multiplier Checker

Table 3. Hardware and Power Overheads for Multiplier

module	size (# of transistors)	percentage of primary mult.	average power	percentage of primary mult.
primary multiplier	32633	100%	42.1mW	100%
modulo checker	3868	11.8%	5.1mW	12%

single cycle. We observe that, even for $C=3$, there is negligible performance impact. In fact, for most benchmarks, we could tolerate even lazier checkers with little impact, but some floating point benchmarks see non-trivial performance loss once C reaches 10.

As can be seen in Table 3, the modulo checker is only one-tenth the size of the primary 4-cycle multiplier. This shows that the checker itself is low-cost in terms of hardware. The power overhead numbers correspond very closely to the area overhead numbers.

If we had implemented a 14-cycle multiplier at the transistor-level, as in the Pentium 4-like processor that we simulate at the microprocessor level, it would have changed the overhead results slightly in our favor. A deeper pipeline for the primary multiplier requires more buffering of partial products, which typically have higher bit-width than operands. Implementing the modulo checker with a deeper pipeline (currently, the modulo operation finishes in one cycle) would also require pipeline buffers. However, the partial modulo results have lower bit-width than the operands. As a result, a deeper primary multiplier pipeline will make the primary multiplier larger with respect to the multiplier checker. Thus, the area and power overheads of the multiplier checker would be lower than in Table 3.

7. Conclusions

We have developed low-cost mechanisms for detecting errors in functional units. Lazy error checkers exploit the opportunity to perform error detection during the time between when an instruction completes and when it commits. We believe that the low costs—in terms of hardware, power, and performance—make our approach viable for this important problem. In the future, we would like to extend this approach to other parts of the microprocessor.

Acknowledgments

This material is based upon work supported by the National Science Foundation under grants CCR-0309164, the National Aeronautics and Space Administration under grant NNG04GQ06G, an equipment donation from Intel Corporation, and a Duke Warren Faculty Scholarship.

References

- [1] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer*, 35(2):59–67, Feb. 2002.
- [2] T. M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proc. of the 32nd Annual IEEE/ACM Int'l Symposium on Microarchitecture*, pages 196–207, Nov. 1999.
- [3] D. Boggs et al. The Microarchitecture of the Intel Pentium 4 Processor on 90nm Technology. *Intel Technology Journal*, 8(1), Feb. 2004.
- [4] G. C. Cardarilli et al. Error Detection in Signed Digit Arithmetic Circuit with Parity Checker. In *Proc. of the 18th IEEE Int'l Symposium on Defect and Fault Tolerance in VLSI Systems*, 2003.
- [5] 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.
- [6] R. Kaivola and N. Narasimhan. Formal Verification of the Pentium4 Floating-Point Multiplier. In *Proc. of Design, Automation and Test in Europe Conference*, pages 20–27, Mar. 2002.
- [7] R. Kalla, B. Sinharoy, and J. M. Tendler. IBM POWER5 Chip: A Dual-Core Multithreaded Processor. *IEEE Micro*, 24(2):40–47, Mar/Apr 2004.
- [8] G. G. Langdon and C. K. Tang. Concurrent Error Detection for Group Look-ahead Binary Adders. *IBM Journal of Research and Development*, 14(5):563–573, Sept. 1970.
- [9] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed Design and Implementation of Redundant Multithreading Alternatives. In *Proc. 29th Int'l Symp. on Computer Architecture*, pages 99–110, May 2002.
- [10] M. Nicolaidis. Efficient Implementations of Self-Checking Adders and ALUs. In *Proc. of the 23rd Int'l Symposium on Fault-Tolerant Computing Systems*, pages 586–595, June 1993.
- [11] S. Oberman. Floating Point Division and Square Root Algorithms and Implementation in the AMD-K7TM Microprocessor. In *Proc. of the 14th IEEE Symposium on Computer Arithmetic*, pages 106–115, Apr. 1999.
- [12] J. H. Patel and L. Y. Fung. Concurrent Error Detection in ALUs by Recomputing with Shifted Operands. *IEEE Transactions on Computers*, C-31(7):589–595, July 1982.
- [13] J. H. Patel and L. Y. Fung. Concurrent Error Detection in Multiply and Divide Arrays. *IEEE Transactions on Computers*, C-32(4), Apr. 1983.
- [14] E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *Proc. of the 29th Int'l Symposium on Fault-Tolerant Computing Systems*, pages 84–91, June 1999.
- [15] F. F. Sellers et al. *Error Detecting Logic for Digital Computers*. McGraw Hill Book Company, 1968.
- [16] T. Sherwood et al. Automatically Characterizing Large Scale Program Behavior. In *Proc. of the Tenth Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.
- [17] F. W. Shih. High Performance Self-Checking Adder for VLSI Processor. In *Proc. of IEEE 1991 Custom Integrated Circuits Conference*, pages 15.7.1–15.7.3, 1991.
- [18] E. S. Sogomonyan, D. Marienfeld, V. Ocheretnij, and M. Gossel. A New Self-Checking Sum-Bit Duplicated Carry-Select Adder. In *Proc. of the Design, Automation, and Test in Europe Conference*, 2004.
- [19] W. J. Townsend, J. A. Abraham, and E. E. Swartzlander, Jr. Quadruple Time Redundancy Adders. In *Proc. of the 18th IEEE Int'l Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 250–256, Nov. 2003.
- [20] D. P. Vadusevan and P. K. Lala. A Technique for Modular Design of Self-Checking Carry-Select Adder. In *Proc. of the 20th IEEE Int'l Symposium on Defect and Fault Tolerance in VLSI Systems*, 2005.
- [21] T. N. Vijaykumar, I. Pomeranz, and K. K. Chung. Transient Fault Recovery Using Simultaneous Multithreading. In *Proc. of the 29th Annual Int'l Symp. on Computer Architecture*, pages 87–98, May 2002.
- [22] M. Yilmaz, D. R. Hower, S. Ozev, and D. J. Sorin. Self-Detecting and Self-Diagnosing 32-bit Microprocessor Multiplier. In *Int'l Test Conference*, Oct. 2006.