Fault Tolerant Microprocessors for Space Missions

Daniel J. Sorin and Sule Ozev

Department of Electrical and Computer Engineering, Duke University

PO Box 90291, Durham, NC 27708

{sorin, sule}@ee.duke.edu

***Abstract*—This research project is developing microprocessors that can autonomically handle hard (permanent) faults that occur during space missions. Rather than use macro-scale redundancy and incur severe power and hardware overheads, we have developed low-cost solutions in the areas of error detection, fault diagnosis, and reconfiguration around hard faults.**

I. Introduction

NASA relies on microprocessors for its space missions. Microprocessors control life-support equipment, navigation, and on-board science experiments. Thus, microprocessor failure can have catastrophic consequences. NASA has traditionally solved the problem of hard (permanent) hardware faults by using macro-scale redundancy, such as triple modular redundancy (TMR). TMR provides good reliability, but it incurs around 200% overhead in terms of hardware and power consumption. As microprocessors continue to use increasing amounts of power, TMR becomes an unappealing solution for power-constrained environments, such as space missions.

Our goal in this work is to create microprocessors that can tolerate hard faults without adding significant redundancy. The key observation, made also by previous research [8, 10, 11], is that modern microprocessors, particularly simultaneously multithreaded (SMT) microprocessors [12] and multicore processors, already contain significant amounts of redundancy for purposes of enhancing performance. We want to use this redundancy to mask hard faults, at the cost of a graceful degradation in performance for microprocessors with hard faults. To achieve our goal, the microprocessor must be able to do three things while it is running.

- It must detect and correct errors caused by faults (both hard and transient).
- It must diagnose where a hard fault is and deconfigure the faulty component in order to prevent its fault from being exercised.

Our research group has made contributions in both of these areas, and we will discuss each in this paper. In Section II, we discuss low-cost error detection mechanisms that use dynamic verification (online checking of invariants) to provide comprehensive coverage. In Section III, we present our novel diagnosis schemes for microprocessors and multipliers. We also discuss how we provide reconfigurability after diagnosis. We conclude in Section IV.

II. Error Detection and Correction

We have developed two low-cost approaches for error detection and correction. In Section A, we discuss dynamic verification of memory consistency, our approach for detecting all errors in the memory systems of multithreaded and multicore processors. In Section B, we discuss dynamic dataflow verification, which is a low-cost way to detect all microprocessor core errors that manifest themselves as dataflow errors.

A. Dynamic Verification of Memory Consistency

The memory system of a modern computer system is a complicated collection of interacting components. For a multicore processor (e.g., Intel CoreDuo) or a multichip multiprocessor, the memory system includes DRAM memories, SRAM caches, an interconnection network over which the cores can communicate, and cache and memory controllers that implement a coherence protocol for sharing data among the cores. We can add error detection mechanisms to each component (e.g., parity bits on messages that traverse the interconnection network), but it is difficult and costly to compose a large number of component error checkers such that they detect all errors, especially errors that involve interactions between components.

To address this problem, we have developed a scheme called Dynamic Verification of Memory Consistency (DVMC). The key idea behind DVMC is to check invariants rather than components. All memory systems must implement a software-visible interface known as a memory consistency model [adve:tutorial:ieeecomputer:1996]. The consistency model is an invariant that an error-free memory system is guaranteed to enforce, and different architectures can specify different consis-

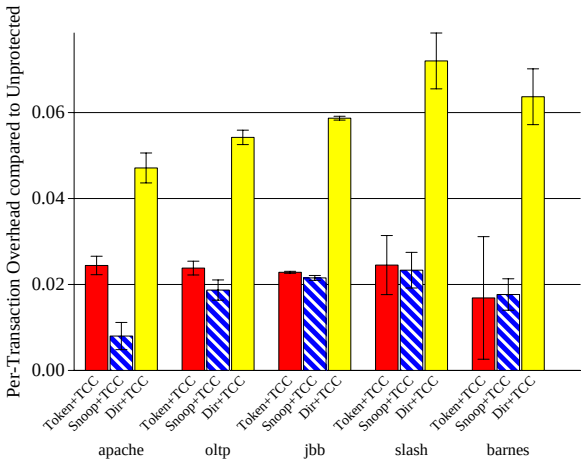

Fig. 1. DVMC Traffic Overhead. For each benchmark, we plot the per-transaction overhead for each of three types of cache coherence protocol (Token Coherence, Snooping, and Directory).

tency models (e.g., the model for Intel IA-32 processors differs from the model for PowerPC processors). Thus, by dynamically verifying (i.e., checking at runtime) that the hardware is implementing its memory consistency model, DVMC can comprehensively detect all possible errors in the memory system. Any error in the memory system must manifest itself as a violation of the consistency model and will thus be detected by DVMC.

We have developed several implementations of DVMC [5, 6, 7]. We now have an implementation [7] that incurs minimal performance degradation and adds only about 1-8% extra traffic on the interconnection network, as shown in Figure 1 for five benchmarks. This overhead is mostly a function of the specific cache coherence protocol. The details of this experiment are explained in our prior paper [7].

When DVMC detects an error, it restores the state of the system to a pre-error checkpoint using the SafetyNet backward error recovery mechanism [9].

B. Dynamic Dataflow Verification

There already exist solutions for detecting errors within a processor core, but they are expensive. Replicating cores or using redundant threads degrades performance significantly and greatly increases power consumption.

Our approach, similar to DVMC, is to check an invariant rather than individual components. For a core, there are only three invariants that must be maintained: the computation (addition, multiplication, etc.), control flow, and dataflow must all be correct. There already exist cheap computation checkers and control flow checkers, but we developed Dynamic Dataflow Verification (DDFV) as the first dataflow checker.

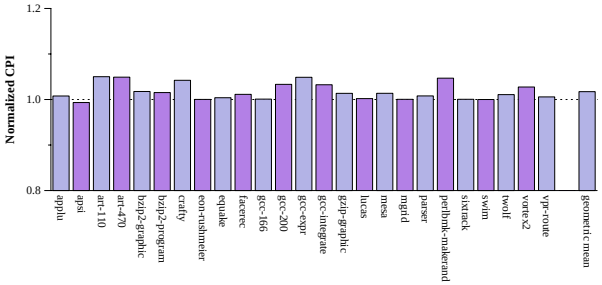

Fig. 2. DDFV Performance Overhead

The basic idea behind DDFV is to compute signatures of portions of the dataflow graph of a program and then compare then to the signatures that are computed dynamically at runtime. If the signatures differ, then there was an error in the execution and DDFV will detect it. DDFV is powerful because it can detect errors in the large portion of the core that is devoted to dynamically reconstructing the dataflow graph at runtime. This portion of the core includes: fetch, decode, register renaming, register reading, instruction scheduling, and data communication between instructions. Once DDFV detects an error, it triggers a core recovery using one of many pre-existing core checkpoint mechanisms.

DDFV incurs modest performance degradation due to embedding signatures into the program (so that they can be compared to the runtime signatures). The performance results in Figure 2 show that the overhead, measured in clock cycles per instruction (CPI), is small across a wide range of benchmarks.

We are currently working to combine DDFV with existing control flow and computation checkers. This combination will provide a very low cost, comprehensive method for detecting all core errors.

III. Fault Diagnosis and Reconfigurability

In this section, we discuss our recent contributions in the areas of fault diagnosis and reconfiguration around hard faults.

A. Microprocessor Core Diagnosis and Reconfiguration

Our core diagnosis scheme [3, 4] dynamically attributes errors to field deconfigurable units (FDUs) as the system is running. Given an error detection mechanism, if an instruction (or micro-op, in the case of IA-32) is determined to be in error, the system records which FDUs that instruction used during its lifetime. If, over a period of time, more than a pre-specified threshold of errors has been attributed to a given FDU, it is very likely that this resource has a hard fault.

Our diagnosis scheme does not rely on any specific error detection mechanism. For purposes of this paper, we assume that we are using DIVA [1], which is a previ-

ously developed scheme for detecting errors in cores. DIVA detects errors by adding a small checker core to each core that is to be checked. It is more expensive, in terms of power and hardware, than the scheme we discussed in Section 2.B, but we have not yet completed the development of our scheme.

The choice of FDU is a design decision for a given implementation. In this paper, the identified FDUs for which we track diagnosis information are: individual entries in the instruction fetch queue (IFQ), individual reservation stations (RS), individual entries in the load-store queue (LSQ), individual entries in the re-order buffer (ROB), individual arithmetic logic units (ALU), and the individual DIVA checkers. We have chosen a fairly fine FDU granularity, but one could choose coarser or even finer granularities if so desired. The hardware bounds of our diagnosis mechanism are the components in which the selected error checker (in our design, DIVA) can detect a fault. Therefore, we do not consider the register file, because DIVA cannot recover from errors in it.

To track each instruction's FDU usage, bits are carried with each instruction from the point of FDU usage to commit. For those structures that the instruction owns at commit, this information is already implicitly available and no extra wires are needed to carry this resource usage info through the pipeline. In our modeled processor, the ROB entries and DIVA checkers use implicit tracking. For the remaining FDUs, the number of bits required is a function of the size of the structure and the granularity into which we are allowing it to be subdivided for later deconfiguration. This represents an engineering trade-off in our design that will allow implementations to select the appropriate FDU granularity/overhead trade-off. With the configuration used in our paper [3], each instruction carries 19 bits of usage information: 5 bits for RS, 6 bits for LSQ, 6 bits for IFQ, and 2 bits for ALUs. For each FDU we track, the processor maintains a small, saturating error counter.

After an FDU has been diagnosed as having a hard fault present, deconfiguring the faulty FDU is desired to avoid the frequent pipeline flushes that DIVA would trigger due to continued manifestation of the fault. In this section, we describe several pre-existing methods for deconfiguring typical microprocessor structures, plus a new way to deconfigure a faulty DIVA checker.

For circular access array structures—such as the IFQ, ROB, and LSQ—we have shown how to add a level of indirection to allow for de-configuration of a single entry with little additional latency added to access time for the structure [2]. In our technique [2], each structure maintains a fault map. This fault map information feeds into the head and tail pointer advancement

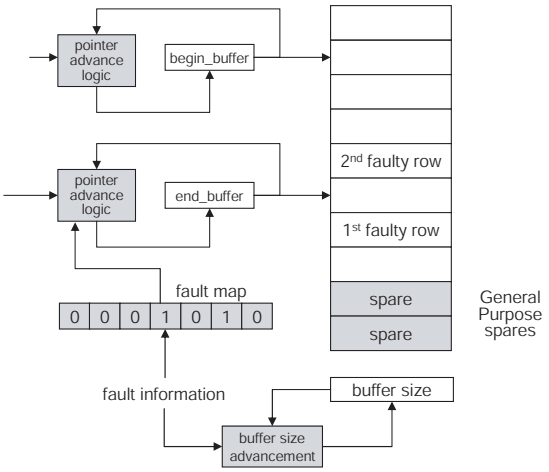

Fig. 3. Deconfiguration of entries in a circular buffer (e.g., reorder buffer). Shading indicates hardware added for entry deconfiguration purposes.

logic, causing the advancement logic to skip an entry that is marked as faulty. If cold spares are available, as we assume and as shown in Figure 3, the structure size can be maintained at the original processor design point. If no spares are provisioned, then the structure size must be updated when the fault map is updated.

For some tabular (i.e., directly addressed) structures—such as reservation stations, register files, etc.—a simple solution is to permanently mark the resource as in-use, thus removing it from further operation [8].

For a functional unit (ALU, etc.), similar to a reservation station, we can mark the resource as permanently busy, preventing further instructions from issuing to it [8]. Cold sparing of functional units is possible, but it may require too much die space, as functional units are relatively large compared to individual ROB entries or reservation stations. We focus on using existing redundancy, since the cost of adding extra redundancy may be too great for commodity microprocessors.

For one of the multiple DIVA checkers, we can map it out if we diagnose it as being permanently faulty. Depending on how DIVA checkers are scheduled, deconfiguration is just as simple as for ALUs; just marking a faulty checker as permanently busy will deconfigure it. Prior work has not looked into deconfiguring DIVA checkers, because no fault diagnosis schemes prior to ours could diagnose hard faults in a checker.

B. Self-Detecting and Reconfiguring Multiplier

In the previous section, we considered ALUs and multipliers to be FDUs. This is a fairly coarse granularity, particularly for large structures like multipliers. Also, each core is likely to have only one multiplier, so deconfiguring it may not be acceptable. Thus, we devel-

oped a multiplier that can diagnose hard faults within its logic and then reconfigure itself to avoid using the faulty logic [13].

C. Delay Fault Diagnosis for Functional Units

Most fault diagnosis schemes concern themselves with stuck-at faults. This fault model represents many underlying physical phenomena and it is commonly used by researchers in the areas of fault tolerance and fault testing. However, the stuck-at fault model does not represent the scenario in which a component is starting to wear out. In this case, the value on a wire is not stuck at a particular value; rather, the value on the wire is generated correctly, but more slowly than in the fault-free case. The beginning of physical wearout often manifests itself as a delay fault, and then complete wearout manifests as a stuck-at fault.

Our goal is to diagnose delay faults before they lead to permanent wearout, because then we can avoid the side effects of wearout, such as failure of nearby circuitry. Our initial work has focused on functional units, like ALUs and multipliers. After we detect an error in a computation, we want to determine if it is permanent and if it is a stuck-at or a delay fault. Simply replaying the inputs to the functional unit is sufficient for diagnosing stuck-at faults, but we must replay the most recent *sequence* of inputs to diagnose delay faults. Thus, we add a small buffer to remember the most recent input pairs, and we replay them after detecting an error. This procedure puts the functional unit back in the state before it received the inputs that triggered the error. Then we replay the error-inducing inputs; if an error occurs this time, then we have diagnosed either a stuck-at or a delay fault. In either case, we have diagnosed a faulty component that we want to stop using.

IV. Conclusions

We are addressing NASA's need for autonomic microprocessor execution in the presence of hard faults. We have developed novel, low-cost, low-power solutions for detecting errors, diagnosing hard faults, and reconfiguring around permanently faulty components. This work addresses microprocessor cores, as well as multicore processors. We believe that our contributions will enable NASA to achieve its desired microprocessor reliability without resorting to expensive, power-hungry macro-scale redundancy.

Acknowledgment

This material is based upon work supported by the National Aeronautics and Space Administration under Grant NNG04GQ06G, the National Science Foundation under grants CCR-0309164 and CCF-0444516, a Duke Warren Faculty Scholarship (Sorin), and an equipment donation from Intel Corporation.

References

[1] T. M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 196–207, Nov. 1999.

[2] F. A. Bower, P. G. Shealy, S. Ozev, and D. J. Sorin. Tolerating Hard Faults in Microprocessor Array Structures. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 51–60, June 2004.

[3] F. A. Bower, D. J. Sorin, and S. Ozev. A Mechanism for Online Diagnosis of Hard Faults in Microprocessors. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 2005.

[4] F. A. Bower, D. J. Sorin, and S. Ozev. Online Diagnosis of Hard Faults in Microprocessors. *ACM Transactions on Architecture and Code Optimization*, To Appear 2007.

[5] A. Meixner and D. J. Sorin. Dynamic Verification of Sequential Consistency. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 482–493, June 2005.

[6] A. Meixner and D. J. Sorin. Dynamic Verification of Memory Consistency in Cache-Coherent Multithreaded Computer Architectures. In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2006.

[7] A. Meixner and D. J. Sorin. Error Detection via Online Checking of Cache Coherence with Token Coherence Signatures. In *Proceedings of the Twelfth International Symposium on High-Performance Computer Architecture*, pages 145–156, Feb. 2007.

[8] P. Shivakumar, S. W. Keckler, C. R. Moore, and D. Burger. Exploiting Microarchitectural Redundancy For Defect Tolerance. In *Proceedings of the 21st International Conference on Computer Design*, Oct. 2003.

[9] D. J. Sorin, M. M. Martin, M. D. Hill, and D. A. Wood. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 123–134, May 2002.

[10] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The Case for Lifetime Reliability-Aware Microprocessors. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004.

[11] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. Exploiting Structural Duplication for Lifetime Reliability Enhancement. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, June 2005.

[12] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 191–202, May 1996.

[13] M. Yilmaz, D. R. Hower, S. Ozev, and D. J. Sorin. Self-Detecting and Self-Diagnosing 32-bit Microprocessor Multiplier. In *International Test Conference*, Oct. 2006.