SWIFT: Software Implemented Fault Tolerance

George A. Reis Jonathan Chang Neil Vachharajani Ram Rangan David I. August
Departments of Electrical Engineering and Computer Science
Princeton University
Princeton, NJ 08544
{gareis, jcone, nvachhar, ram, august}@princeton.edu

Abstract

To improve performance and reduce power, processor designers employ advances that shrink feature sizes, lower voltage levels, reduce noise margins, and increase clock rates. However, these advances make processors more susceptible to transient faults that can affect correctness. While reliable systems typically employ hardware techniques to address soft-errors, software techniques can provide a lower-cost and more flexible alternative. This paper presents a novel, software-only, transient-fault-detection technique, called SWIFT. SWIFT efficiently manages redundancy by reclaiming unused instruction-level resources present during the execution of most programs. SWIFT also provides a high level of protection and performance with an enhanced control-flow checking mechanism. We evaluate an implementation of SWIFT on an Itanium 2 which demonstrates exceptional fault coverage with a reasonable performance cost. Compared to the best known single-threaded approach utilizing an ECC memory system, SWIFT demonstrates a 51% average speedup.

1 Introduction

In recent decades, microprocessor performance has been increasing exponentially. A large fraction of this improvement is due to smaller and faster transistors with low threshold voltages and tighter noise margins enabled by improved fabrication technology. While these devices yield performance enhancements, they will be less reliable [23], making processors that use them more susceptible to *transient faults*.

Transient faults (also known as *soft errors*), unlike manufacturing or design faults, do not occur consistently. Instead, these intermittent faults are caused by external events, such as energetic particles striking the chip. These events do not cause permanent physical damage to the processor, but can alter signal transfers or stored values and thus cause incorrect program execution.

Transient faults have caused significant failures. In 2000, Sun Microsystems acknowledged that cosmic rays interfered with cache memories and caused crashes in server systems at major customer sites, including America Online, eBay, and dozens of others [3].

To counter these faults, designers typically introduce redundant hardware. For example, some storage structures such as caches and memory include error correcting codes (ECC) and parity bits so the redundant bits can be used to detect or even correct the fault. Similarly, combinational logic within the processor can be protected by duplication. Output from the duplicated combinational logic blocks can be compared to detect faults. If the results differ, then the system has experienced a transient fault and the appropriate recovery or reporting steps can be initiated.

High-availability systems need much more hardware redundancy than that provided by ECC and parity bits. For example, IBM has historically added 20-30% of additional logic within its mainframe processors for fault tolerance [24]. When designing the S/390 G5, IBM introduced even more redundancy by fully replicating the processor's execution units to avoid various performance pitfalls with their previous fault tolerance approach [24]. To alleviate transient faults, in 2003, Fujitsu released its fifth generation SPARC64 with 80% of its 200,000 latches covered by some form of error protection, including ALU parity generation and a mul/divide residue check [1]. Since the intensity of cosmic rays significantly increases at high altitudes, Boeing designed its 777 aircraft system with three different processors and data buses while using a majority voting scheme to achieve both fault detection and recovery [28, 29].

Using these hardware fault tolerant mechanisms is too expensive for many processor markets, including the highly price-competitive desktop and laptop markets. These systems may have ECC or parity in the memory subsystem, but they certainly do not possess double- or triple-redundant execution cores. Ultimately, transient faults in both memory and combinational logic will need to be addressed in *all* aggressive processor designs, not just those used in high-availability applications.

Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)
0-7695-2298-X/05 $ 20.00 IEEE

In this paper, we propose SWIFT, a software-based, single-threaded approach to achieve redundancy and fault tolerance. For brevity's sake, we will be restricting ourselves to a discussion of fault detection. However, since SWIFT performs fault detection in a manner compatible with most reporting and recovery mechanisms, it can be easily extended to incorporate complete fault tolerance.

SWIFT is a compiler-based transformation which duplicates the instructions in a program and inserts comparison instructions at strategic points during code generation. During execution, values are effectively computed twice and compared for equivalence before any differences due to transient faults can adversely affect program output.

A software-based, single-threaded approach like SWIFT has several desirable features. First and foremost, the technique does not require any hardware changes. Second, the compiler is free to make use of slack in a program's schedule to minimize performance degradation. Third, programmers are free to vary transient fault policy within a program. For example, the programmer may choose to check only essential code segments or to vary the manner in which detected errors are handled to achieve the best user experience. Fourth, a compiler orchestrated relationship between the duplicated instructions allows for simple methods to deal with exception-handling, interrupt-handling, and shared memory.

SWIFT demonstrates the following improvements over prior work:

- As a software-based approach, SWIFT requires no hardware beyond ECC in the memory subsystem.
- SWIFT eliminates the need to double the memory requirement by acknowledging the use of ECC in caches and memory.
- SWIFT increases protection at no additional performance cost by introducing a new control-flow checking mechanism.
- SWIFT reduces performance overhead by eliminating branch validation code made unnecessary by this enhanced control flow mechanism.
- SWIFT performs better than all known single-threaded full software detection techniques. Though no direct comparison is made to multithreaded approaches, it performs *on par* with hardware multithreading-based redundancy techniques [19] without the additional hardware cost.
- Methods to deal with exception-handling, interrupt-handling and shared memory programs in software-based, single-threaded make SWIFT deployable in both uniprocessor and multiprocessor environments.

While SWIFT can be implemented on any architecture and can protect individual code segments to varying degrees, we evaluate a full program implementation running on Itanium 2. In these experiments, SWIFT demonstrates exceptional fault-coverage with a reasonable performance cost. Compared to the best known single-threaded approach utilizing an ECC memory system, SWIFT demonstrates a 14% average speedup.

The remainder of this paper is organized as follows. Section 2 discusses the relation to prior work. Section 3 describes the SWIFT technique as an evolution of work found in the literature. The section details improvements to existing software-only techniques that increase performance as well as the reliability of the system. Section 4 addresses various issues related to the implementation and deployment of SWIFT. Section 5 evaluates the reliability and performance of an implementation of SWIFT for IA-64 and presents experimental results. Finally, Section 6 summarizes the contributions of this work.

2 Relation to Prior Work

Redundancy techniques can be broadly classified into two kinds: hardware-based and software-based. Several hardware redundancy approaches have been proposed. Mahmood and McCluskey proposed using a *watchdog* [6] processor to compare and validate the outputs against the main running processor. Austin proposed DIVA [2], which uses a main, high-performance, out-of-order processor core that executes instructions and a second, simpler core to validates the execution. Real system implementations like the Compaq NonStop Himalaya [5], IBM S/390 [24], and Boeing 777 airplanes [28, 29] replicated part or all of the processor and used checkers to validate the redundant computations.

Several researchers have also made use of the multiplicity of hardware blocks readily available on multi-threaded/multi-core architectures to implement redundancy. Saxena and McCluskey [21] were the first to use redundant threads to alleviate soft errors. Rotenberg [20] expanded the SMT redundancy concept with AR-SMT. Reinhardt and Mukherjee [19] proposed simultaneous Redundant MultiThreading (RMT) which increases the performance of AR-SMT and compares redundant streams before data is stored to memory. The SRTR processor proposed by Vijaykumar et al. [26] expand the RMT concept to add fault recovery by delaying commit and possibly rewinding to a known good state. Mukherjee et al. [8] proposed a Chip-level Redundantly Threaded multiprocessor (CRT) and Gomaa et al. [4] expanded that approach with CRTR to enable recovery. Ray et al. [16] proposed modifying an out-of-order superscalar processor's microarchitectural components to implement redundancy. All hardware-based

approaches require the addition of some form of new hardware logic to meet redundancy requirements and thus come at a price.

Software-only approaches to redundancy are attractive because they essentially come *free of cost*. Shirvani et al. [22] proposed a technique to enable ECC for memory data via a software-only technique. Oh and McCluskey [10] analyzed different options for procedure duplication and argument duplication at the source-code level to enable software fault tolerance while minimizing energy utilization. Rebaudengo et al. [17] proposed a source-to-source pre-pass compiler to generate fault detection code in a high level language. The technique increases overhead by 3-5 times and allows 1-5% of faults to go undetected. Oh et al. [12] proposed a novel software redundancy approach (EDDI) wherein all instructions are duplicated and appropriate "check" instructions are inserted to validate. A sphere of replication (SoR) [19] is the logical domain of redundant execution. EDDI's SoR is the entire processor core and the memory subsystem. Oh et al. [11] developed a pure software control-flow checking scheme (CFCSS) wherein each control transfer generates a run-time signature that is validated by error checking code generated by the compiler for every block. Venkatasubramanian et al. [25] proposed a technique called Assertions for Control Flow Checking (ACFC) that assigns an execution parity to each basic block and detects faults based on parity errors. Ohlsson et al. [13] developed a technique to monitor software control flow signatures without building a control flow graph, but requires additional hardware. A coprocessor is used to dynamically compute the signature from the running instruction stream and watchdog timer is used to detect the absence of block signatures.

SWIFT makes several key refinements to EDDI and incorporates a software only signature-based control-flow checking scheme to achieve exceptional fault-coverage. The major difference between EDDI and SWIFT is, while EDDI's SoR includes the memory subsystem, SWIFT moves memory out of the SoR, since memory structures are already well-protected by hardware schemes like parity and ECC, with or without scrubbing [7]. SWIFT's performance greatly benefits from having only half the memory usage and only half as many stores writing to the memory subsystem. This and other optimizations, explained in detail in Section 3, enable SWIFT to significantly outperform EDDI.

Table 1 gives a comparison of various redundancy approaches. The column headings are the different logical entities that need to be protected. The rows contain details about each technique. An "all" in any of the table cells means the technique in the given row offers full protection to the logical state in the corresponding column. A "none" means the technique does not offer any protection and assumes some form of protection from outside. "some" and "most" are intermediate levels of protection, wherein the technique offers protection to a subset of the state for a subset of the time the state is live. More detail for those protection levels is provided in the footnotes.

3 Software Fault Detection

SWIFT is an evolution of the best practices in software-based fault detection. In this section, we will describe the foundation of this work, EDDI [12], discuss extending EDDI with control-flow checking with software signatures [11], and finally introduce the novel extensions that comprise SWIFT.

Throughout this paper, we will be assuming a *Single Event Upset* (SEU) fault model, in which exactly one bit is flipped throughout the entire program. Although the techniques presented will be partially effective at detecting multiple faults, as we shall see in Section 3.7, the probability of a multiple fault event is much smaller than an SEU, making SEU detection by far the first-order concern. We will also assume, in line with most modern systems, that the memory subsystem, including processor caches, are already adequately protected using techniques like parity and ECC.

As presented, the following transformations are used to *detect* faults. However, upon fault detection, arbitrary user-defined fault recovery code can be executed permitting a variety of recovery mechanisms. Because fault detection and fault recovery can be decoupled in this manner, they are often studied independently. Furthermore, because of the relative rarity of faults, recovery code is executed far more infrequently than detection code, which must be run continuously. This makes efficacious and cost-effective fault detection a much more difficult (and interesting) problem than fault recovery.

3.1 EDDI

EDDI [12] is a software-only fault detection system that operates by duplicating program instructions and using this redundant execution to achieve fault tolerance. The program instructions are duplicated by the compiler and are intertwined with the original program instructions. Each copy of the program, however, uses different registers and different memory locations so as to not interfere with one another. At certain synchronization points in the combined program code, check instructions are inserted by the compiler to make sure that the original instructions and their redundant copies agree on the computed values.

Since program correctness is defined by the output of a program, if we assume memory-mapped I/O, then a program has executed correctly if all stores in the program have executed correctly. Consequently, it is natural to use store instructions as synchronization points for compari-

Technique	Category	Opcode	Loads	Stores	Control Transfers	Other Insns	Memory State	Hardware Cost
DIVA	HW	all	all	all	all	all	none	Additional processor
Himalaya	HW	all	all	all	all	all	none	Dual core, checker
RMT	HW	all	all	all	all	all	none	SMT, checker, Sync logic
CRT	HW	all	all	all	all	all	none	CMP, checker, Sync logic
Superscalar	HW	most[a]	most[a]	most[a]	most[a]	most[a]	none	Replicator, Extra logic
CFCSS	SW	some[b]	none	none	most[c]	none	none	None
EDDI	SW	most[d]	all	all	most[e]	all	all	None
ACFC	SW	some[b]	none	none	most[c]	none	none	None
SWIFT	SW	most[f]	all	most[e]	most[c]	all	none	None

[a] instruction replicator and register faults go undetected
[b] coverage only for branch opcodes
[c] incorrect control transfers to within a control block may go undetected in rare circumstances
[d] no coverage for branch opcodes and opcodes that differ from branch opcodes by a Hamming distance of 1
[e] strikes to operands between validation and use by the instruction's functional unit go undetected
[f] no coverage for store opcodes and opcodes that differ from a store opcode by a Hamming distance of 1

Table 1. Comparison of Various Redundancy Approaches

```
ld r12=[GLOBAL]          ld r12=[GLOBAL]
                       1: ld r22=[GLOBAL+offset]
add r11=r12,r13          add r11=r12,r13
                       2: add r21=r22,r23
                       3: cmp.neq.unc p1,p0=r11,r21
                       4: cmp.neq.or p1,p0=r12,r22
                       5: (p1) br faultDetected
st m[r11]=r12            st m[r11]=r12
                       6: st m[r21+offset]=r22
```

(a) Original Code (b) EDDI Code

Figure 1. EDDI Fault Detection

son. Unfortunately, it is insufficient to use store instructions as the only synchronization points since misdirected branches can cause stores to be skipped, incorrect stores to be executed, or incorrect values to ultimately feed a store. Therefore, branch instructions must also be synchronization points at which redundant values are compared.

Figure 1 shows a sample code sequence before and after the EDDI fault-detection transformation. For consistency, throughout the paper we will make use of the IA64 instruction set architecture (ISA) although EDDI was originally implemented for the MIPS ISA. In the example, the load from a global constant address is duplicated as **1**. Notice that the duplicated load reads its data from a different address and stores its result into a different register to avoid conflicting with the original instruction. Similarly, the add instruction is duplicated as instruction **2** to create a redundant chain of computation. The store instruction is a synchronization point, and instructions **3** and **4** compare the store's operands to their redundant copies. If any difference is detected, instruction **5** will report an error. Otherwise the store, and its redundant instruction, **6**, will execute storing values to non-conflicting addresses.

Also, although in the example program an instruction is immediately followed by its duplicate, an optimizing compiler (or dynamic hardware scheduler) is free to schedule the instructions to use additional available ILP thus minimizing the performance penalty of the transformation. Depending on whether the redundant duplicates are executed in parallel or sequentially, two different forms of redundancy, *temporal* and *spatial*, will be exploited. Temporal redundancy computes the same data value at two different times, usually on the same hardware. Spatial redundancy computes the same data value in two different pieces of hardware, usually at the same time.

3.2 Eliminating the Memory Penalty

While EDDI is able to effectively detect transient faults, unfortunately, the transformation incurs a significant memory overhead. As Figure 1 demonstrates, each location in memory needs to have a corresponding shadow location in memory for use with the redundant duplicate. This memory duplication incurs a significant hardware cost, but it also incurs a significant performance cost since cache sizes are effectively halved and additional memory traffic is created.

Recall that under our assumptions, the memory hierarchy is protected with some form of error correction. Consequently, we propose eliminating the use of two distinct memory locations for all memory values eliminating duplicate store instructions. It is still necessary to duplicate load instructions since all values contained in registers require a redundant duplicate. These modifications will *not* reduce the fault detection coverage of the system, but will make the protected code execute more efficiently and require less memory. For the remainder of the paper, we will refer to this as EDDI+ECC.

3.3 Control Flow Checking

In addition to the memory penalty, EDDI also suffers from incomplete protection for control flow faults. With EDDI, although the input operands for branch instructions are verified, there is the possibility that a program's control flow gets erroneously misdirected without detection. The corruption can happen during the execution of the branch, register corruption after branch check instructions, or even due to a fault in the instruction pointer update

```
add r11=r12,r13
cmp.lt.unc
   p11,p0=r11,r12
(p11) br L1
...
L1:
st m[r11]=r12
```

(a) Original Code

```
    add r11=r12,r13
1:  add r21=r22,r23
    cmp.lt.unc
       p11,p0=r11,r12
2:  cmp.lt.unc p21,p0=r21,r22
3:  mov r1=0
4:  (p11) xor r1=r1,1
5:  (p21) xor r1=r1,1
6:  cmp.neq.unc p1,p0=r1,0
7:  (p1) br faultDetected
    (p11) br L1
    ...
    L1:
8:  xor GSR=GSR,L0_to_L1
9:  cmp.neq.unc p2,p0=GSR,sig_1
10: (p2) br faultDetected
11: cmp.neq.unc p3,p0=r11,r21
12: cmp.neq.or p3,p0=r12,r22
13: (p3) br faultDetected
    st m[r11]=r12
```

(b) EDDI+ECC+CF code

Figure 2. Control Flow Checking

logic. To make EDDI more robust to such strikes, additional checks can be inserted to ensure that control flow is being transferred properly. The technique that is described here was originally proposed by Oh, et al. [11]. We will refer to EDDI+ECC with this control flow validation as EDDI+ECC+CF.

To verify that control transfer is in the appropriate control block, each block will be assigned a signature. A designated general purpose register, which we will call the GSR (General Signature Register), will hold these signatures and will be used to detect faults. The GSR will always contain the signature for the currently executing block. Upon entry to any block, the GSR will be xor'ed with a statically determined constant to transform the previous block's signature into the current block's signature. After the transformation, the GSR can be compared to the statically assigned signature for the block to ensure that a legal control transfer occurred.

Using a statically-determined constant to transform the GSR forces two blocks which both jump to a common block (a control flow merge) to share the same signature. This is undesirable since faults which transfer control to or from blocks that share the same signature will go undetected. To avoid this, a run-time adjusting signature can be used. This signature is assigned to another designated register, and at the entry of a block, this signature, the GSR, and a predetermined constant are all xor'ed together to form the new GSR. Since the run-time adjusting signature can be different depending on the source of the control transfer, it can be used to compensate for differences in signatures between source blocks.

This transformation is illustrated in Figure 2. Instruction **1** and **2** are the redundant duplicates for the add and compare instructions, respectively. Recall that in the EDDI transformation, branches are synchronization points. Instructions **3** through **7** are inserted to compare the predicate `p11` to its redundant duplicate `p21` and branch to error code if a fault is detected. The control flow additions begin with instruction **8**. This instruction transforms the GSR from the previous block to the signature for this block. Instructions **9** and **10** ensure that the signature is correct, and if an incorrect signature is detected error code is invoked. Finally, instructions **11** through **13** are inserted to handle the synchronization point induced by the later store instruction.

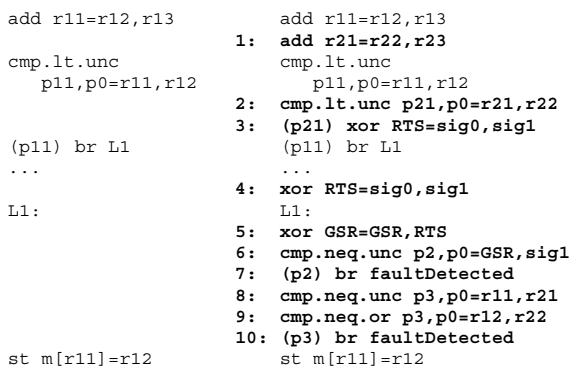
The transformation will detect any fault that causes a control transfer between two blocks that should not jump to one another. Any such control transfer will yield incorrect signatures even if the erroneous transfer jumps to the middle of a basic block. The control flow transformation does not ensure that the correct direction of the conditional branch is taken, only that the control flow is diverted to the taken or untaken path. The base EDDI transformation provides reasonable guarantees since the branches input operands are verified prior to its execution, however, faults that occur during the execution of a branch instruction which influence the branch direction will not be detected by EDDI+ECC+CF.

3.4 Enhanced Control Flow Checking

To extend fault detection coverage to cases where branch instruction execution is compromised, we propose an enhanced control flow checking transformation, EDDI+ECC+CFE. The transformation is similar to EDDI+ECC+CF for blocks using run-time adjusting signatures, but our contribution increases the reliability of the control flow checking. The enhanced mechanism uses a dynamic equivalent of a run-time adjusting signature for all blocks, including those that are not control flow merges. Effectively, each block asserts its target using the run-time adjusting signature, and each target confirms the transfer by checking the GSR. Conceptually, the run-time adjusting signature combined with the GSR serve as a redundant duplicate for the program counter (PC).

The transformation is best explained through an example. Consider the program shown in Figure 3. Just as before, instructions **1** and **2** are the redundant duplicates for the add and compare instructions, respectively. In this example, for brevity, the synchronization check before the branch instruction has been omitted. Instruction **3** computes the run-time signature for the target of the branch. The run-time signature is computed by xor'ing the signature of the current block, with the signature of the target block. Since the branch is predicated, the assignment to RTS is also predicated using the redundant duplicate for the predicate register. Instruction **4** is the equivalent of instruction **3** for the fall through control transfer.

Instruction **5**, at the target of a control transfer, xors RTS



(a) Original Code (b) EDDI+ECC+CFE Code

Figure 3. Enhanced Control Flow Checking

with the GSR to compute the signature of the new block. This signature is compared with the statically assigned signature in instruction **6** and if they mismatch error code is invoked with instruction **7**. Just as before, instructions **8** through **9** implement the synchronization checks for the store instruction.

Notice that with this transformation, even if a branch is incorrectly executed, the fault will be detected since the RTS register will have the incorrect value. Therefore, this control transformation more robustly protects against transient faults. As a specific example, again consider the code from Figure 2. If a transient fault occurred to the guarding predicate of the original branch (`p11`) after it was read for comparison, (i.e. after instruction **4**), then execution would continue in the wrong direction, but EDDI+ECC+CF would not detect that error. The EDDI+EDD+CF control flow checking only ensures that execution is transfered to a valid control block, such as the taken branch label or fall through path, but does not ensure that the correct conditional control path is taken. The enhanced control flow checking detects this case by dynamically updating the target signature based on the redundant conditional instructions (**3**) and checking at the beginning of each control block (**5,6,7**).

3.5 SWIFT

This section will describe optimizations to the EDDI+ECC+CFE transformation. These optimizations applied to the EDDI+ECC+CFE transformation comprise SWIFT. The section will conclude with a qualitative analysis of the SWIFT system including the faults that the system cannot detect.

3.5.1 Control Flow Checking at Blocks with Stores

The first optimization comes from the observation that it is only the store instructions that ultimately send data out of the SoR. As long as we can ensure that stores execute only if they are "meant to" and stores write the correct data to the correct address, the system will run correctly. We use this observation to restrict enhanced control flow checking only to blocks which have stores in them. The updates to GSR and RTS are performed in all blocks, but signature comparisons are restricted to blocks with stores. Removing the signature check instructions with this optimization, abbreviated `SCFOpti`, can further reduce the overhead for fault tolerance at no reduction in reliability. Since signature comparisons are computed at the beginning of every block that contains a store instruction, any deviation from the valid control flow path to that point will be detected before memory and output is corrupted. This optimization slightly increases performance and reduces the static size, as will explained in Section 5, for no reduction in reliability.

3.5.2 Redundancy in Branch/Control Flow Checking

Another optimization is enabled by realizing that branch checking and enhanced control flow checking are redundant. While branch checking ensures that branches are taken in the proper direction, enhanced control flow checking ensures that all control transfers are made to the proper address. Note that verifying *all* control flow *subsumes* the notion of branching in the right direction. Thus, doing control flow checking alone is sufficient to detect all control flow errors. Removing branch checking via this optimization, abbreviated `BROpti`, can significantly reduce the performance and static size overhead for fault detection and will be evaluated in Section 5. Since the control flow checking, instructions **3,5,6,7** of Figure 3, *subsume* the branch direction checking, instructions **3,4,5,6,7** of Figure 2, there is no reduction in reliability by removing the branch direction checking.

3.6 Undetected Errors

There are *two* primary points-of-failure in the SWIFT technique. Since redundancy is introduced solely via software instructions, there can be a delay between validation and use of the validated register values. Any strikes during this gap might corrupt state. While all other instructions have some form of redundancy to guard them against such strikes, bit flips in store address or data registers are uncaught. This can cause incorrect program execution due to incorrect writes going outside the SoR. These can be due to incorrect store values or incorrect store addresses.

The second point-of-failure occurs if an instruction opcode is changed to a store instruction by a transient fault.

These stores are unprotected since the compiler did not see this instruction. The store will be free to execute and the value it stores will leave the SoR.

3.7 Multibit Errors

The above code transformations are sufficient to catch single-bit faults in all but a few rare corner cases. However, it is less effective at detecting multibit faults. There are two possible ways in which multibit faults can cause problems. The first is when the same bit is flipped in both the original and redundant computation. The second occurs when a bit is flipped in either the original or redundant computation and the comparison is also flipped such that it does not branch to the error code. Fortunately, these patterns of multibit errors are unlikely enough to be safely ignored.

We may estimate the probability of each of these types of multibit errors. Suppose that instead of a single-upset fault model, we use a dual-upset fault model, wherein two faults are injected into each program with a uniformly random distribution. Let us first consider the case where a bit is flipped in the original as well as the redundant computation. If we assume that the same fault must occur in the same bit of the same instruction for the fault to go undetected, then the probability can be easily computed as $\mathrm{P}(\mathrm{error}_{\mathrm{redundant}}|\mathrm{error}_{\mathrm{original}}) = \frac{1}{64} \cdot \frac{1}{\#_{\mathrm{instructions}}}$, simply the probability of that particular instruction being chosen times the probability of a particular bit being chosen (in this case, we assume 64-bit registers). Since the average SPEC benchmark typically has on the order of 10^9 to 10^{11} dynamic instructions, the probability of this sort of fault occurring will be in the neighborhood of one in a trillion.

Now, let us consider the case in which a bit is flipped along one of the computation paths and another bit is flipped in the comparison. If we assume that there is only one comparison for every possible fault, then the probability of error is simply $\mathrm{P}(\mathrm{error}_{\mathrm{comparison}}|\mathrm{error}_{\mathrm{original}}) = \frac{1}{\#_{\mathrm{instructions}}}$. This probability will be about one in ten billion on average. Note that this is a gross *overestimate* of this sort of error because it assumes only one comparison for each fault, whereas in reality, there may be many checks on a faulty value as a result of its being used in the computation of multiple stores or branches.

4 Implementation Details

This section presents details specific to the implementation and deployment of SWIFT. In particular, we consider different options for calling convention, implementations on multiprocessor systems, and the effects of using an ISA with predication (IA64).

4.1 Function calls

Since function calls may affect program output, incorrect function parameter values may result in incorrect program output. One approach to solve this is to simply make function calls as synchronization points. Before any function call, all input operands are checked against their redundant copies. If any of them mismatch, a fault is detected. Otherwise, the original versions are passed as the parameters to the function. At the beginning of the function, the parameters must be reduplicated into original and redundant versions. Similarly, on return, only one version of the return values will be returned. These must be duplicated into redundant versions for the remaining redundant code to function.

All of this adds performance overhead and it introduces points of vulnerability. Since only one version of the parameters is sent to the function, faults that occur on the parameters after the checks made by the caller and before the duplication by the callee will not be caught. To handle function calls more efficiently and effectively, the calling convention can be altered to pass multiple sets of computed arguments to a function and to return multiple return values from a function. Note that only arguments passed in registers need be duplicated. Arguments that are passed via memory do not need to be replicated, since memory is outside the SoR. Multiple return values simply require that an extra register be reserved for the replicated return value. This incurs the additional pressure of having twice as many input and output registers, but it ensures that fault detection is preserved across function calls.

Note that interaction with unmodified libraries is possible, provided that the compiler knows which of the two calling conventions to use.

4.2 Shared Memory, Interrupts, and Exceptions

When multiple processes communicate with each other using shared memory, the compiler cannot possibly enforce an ordering of reads and writes across processes. Thus, the two loads of a duplicated pair of loads are not guaranteed to return the same value, as there is always the possibility of intervening writes from other processes. While this does not reduce the fault-coverage of the system in any way, it will increase the detected fault count by contributing to the number of detected fault that would not have caused a failure. This is true in both uniprocessor and multiprocessor systems.

Shared-memory programs are only a part of the problem though. We find ourselves in a very similar situation when an interrupt or exception occurs between the two loads of a duplicated pair and the interrupt or exception handler changes the contents at the load address.

1. **Hardware Solutions**

   This is a problem that affects RMT machines too. Thus, we can appeal to "safe" hardware-based load value duplication techniques like the Active Load Address Buffer (ALAB) or the Load Value Queue (LVQ) used in RMT machines [19] and adapt them to a SWIFT system. However, these are hardware techniques and come at a cost.

2. **No Duplication for Loads**

   One software-only solution to this problem is for the compiler to do only one load instead of doing two loads and then duplicate the loaded value for both the original and redundant version consumers. While this is a simple solution, it has the disadvantage of removing redundancy from the load execution, thereby making loads yet another single point-of-failure.

3. **Dealing with Potentially-Excepting Instructions**

   If the compiler knows *a priori* that certain instructions may cause faults, it may choose to enforce a schedule in which pairs of loads are not split across such instructions. This can prevent most exceptions from being raised in between two versions of a load instruction and more importantly allows us to have redundancy in load execution (as opposed to using just a single load and losing load redundancy). Asynchronous signals and interrupts cannot be handled in this manner and we might have to fall back on a hardware solution or the single-load solution to deal with those.

4.3 Logical Masking from Predication

Branches in a typical RISC architecture take in two register values and branch based upon the result of the comparison of the two values. A software fault detection mechanism inserts instructions to compare the original and redundant values of these two operands just before branches, and if the values differ an error is signaled. However, not all faults detected by such a comparison need be detected. For example, consider the branch `br r1!=r2`. If, in the absence of a fault, the branch were to be taken, it is likely that even after a strike to either `r1` or `r2`, the condition would still hold true. If the branch would have had the same outcome despite the error, the error can be safely ignored. This logical masking [19], allows the fault detection mechanism to be less conservative in detecting errors, thus reducing the overall false detected unrecoverable error [19] count, while still maintaining the same coverage.

While special checks would have to be used to check for logical masking in a conventional ISA, predicated architectures naturally provide logical masking. For example, in the IA64 ISA, conditional branches are executed based upon a predicate value which is computed by prior predicate-defining instructions. Since there is no validation before predicate-defining instructions, they go ahead and compute the predicates and any bit flips due to strikes are not noticed unless they produce unequal predicate values.

5 Evaluation

This section evaluates the techniques from Section 3 in order to determine the performance of each technique and to verify their ability to detect faults.

5.1 Performance

To evaluate the performance impact of our techniques, a pre-release version of the OpenIMPACT compiler [14] was modified to add redundancy and was targeted at the Intel Itanium 2 processor. A version was created for each of the EDDI+ECC+CFE and SWIFT techniques. To see the effect of each optimization individually, versions were also created with each of the specific optimizations removed: SWIFT-SCFopti to analyze the control flow checking only at blocks with stores from Section 3.5.1 and SWIFT-BRopti to analyze branch checking optimization from Section 3.5.2.

The modified compilers were used to evaluate the techniques on SPEC CINT2000 and several other benchmark suits, including SPEC FP2000, SPEC CINT95 and MediaBench. These executions were compared against binaries generated by the original OpenIMPACT compiler which have no fault detection. The fault detection code was inserted into the low level code immediately before register allocation and scheduling. Optimizations that would have interfered with the duplicated and detection code, like Common Subexpression Elimination, were modified to respect the fault detecting code.

Performance metrics were obtained by running the resulting binaries with all reference inputs on an HP workstation zx6000 with 2 900Mhz Intel Itanium 2 processors running Redhat Advanced Workstation 2.1 with 4Gb of memory. The `perfmon` utility [15] was used to measure the CPU time, instructions committed and NOPs committed.

The results in Figure 4(a) show that the normalized execution time of the EDDI+ECC+CFE technique had a geometric mean of 1.62 compared to the baseline, no fault detection IMPACT binaries. For the SWIFT version, the execution time was 1.41. As explained earlier, the EDDI+ECC+CFE version does comparisons of the values used before every branch while the SWIFT version does not. As can also be seen from Figure 4(a), the optimization due to control flow checking accounts for almost all of the 0.21 difference from the EDDI+ECC+CFE to the SWIFT version.

If the program were just run twice, the normalized execution time would be exactly 2.00. Since additional

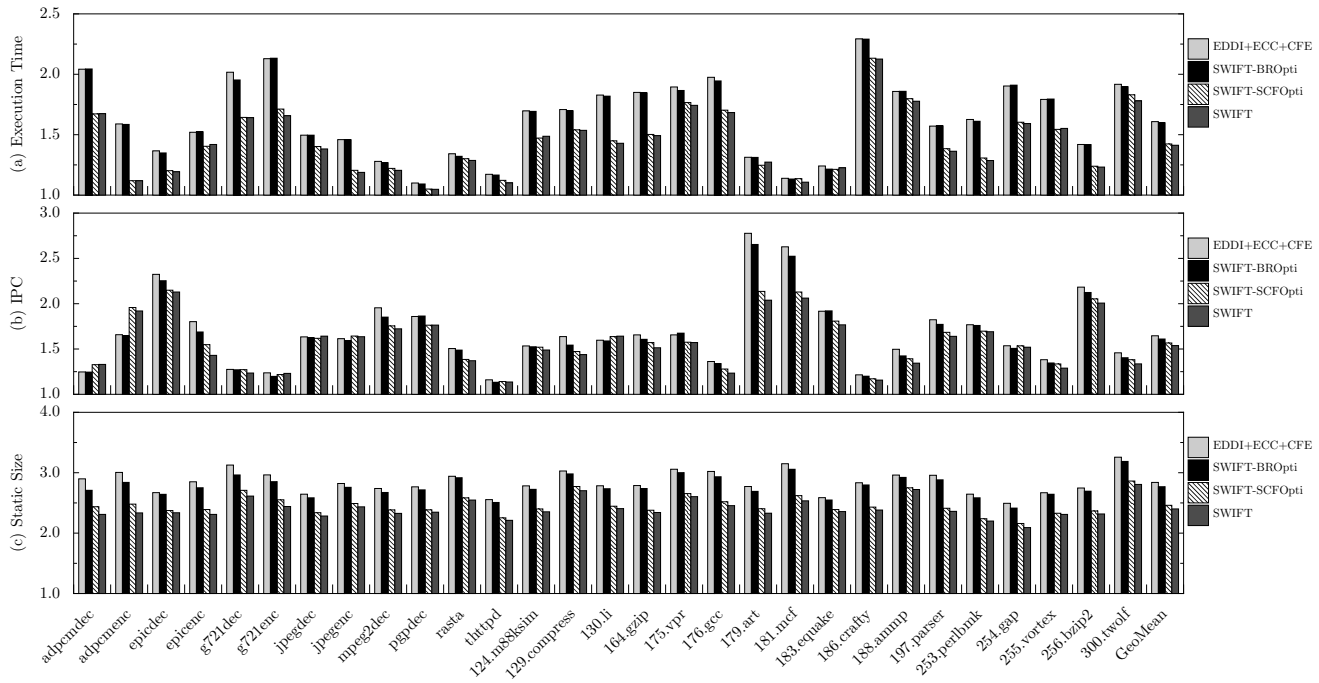


Figure 4. EDDI+ECC+CFE, SWIFT-BROpti, SWIFT-SCFOpti, and SWIFT performance and code characteristics normalized to unprotected code.

code would need to be executed to compare and validate the two program outputs the overall degradation would be much greater than 2.00. The 1.62 and 1.41 normalized execution times of EDDI+ECC+CFE and SWIFT indicate that these methods are exploiting the unused processor resources present during the execution of the baseline program. This is corroborated by the IPC numbers shown in Figure 4(b). The geometric mean of the normalized IPC for EDDI+ECC+CFE was 1.53 and for SWIFT, it was 1.48. The additional branch checks that are present in EDDI+ECC+CFE technique enable more independent work and thus increase the IPC. Scheduling both version of the program together enables a normalized IPC of roughly 1.5 when compared with the non-detecting executions.

Figure 4(c) shows the static sizes of the binaries normalized to baseline with no detection ability. EDDI+ECC+CFE is 2.83x larger than the baseline while SWIFT is 2.40x larger. Both techniques duplicate all instructions except for NOPs, stores, and branches and then insert detection code. The SWIFT technique generates binaries that are, on average, 15% smaller than the EDDI+ECC+CFE technique because it does not generate the extra instructions that are eliminated by each optimization. The store control block optimization alone reduces the static size by 2% while the branch checking optimization alone reduces the static size by 13%. The optimizations capture different opportunities as can be seen by their additive effect in SWIFT.

Figure 5 shows the instruction counts for all four techniques normalized to the baseline, no fault detection instruction count. Note that the light-grey region of Figure 5 represents the fraction of total dynamic instructions that are NOP instruction. The normalized instruction counts have a geometric mean of 2.73 for EDDI+ECC+CFE and 2.23 for SWIFT. These numbers follow the same trend as the static binary size numbers in Figure 4(c). However, the dynamic instruction counts grow disproportionally to the static binary size increases, which would indicate that programs spend, on the balance, slightly more of their execution time in branch-heavy or store-heavy routines.

5.2 Fault Detection

The techniques' abilities to detect faults were also evaluated. We used Pin [18] to instrument our binaries. For the purpose of these experiments, our instrumentation tools ignored the regions in the binaries corresponding to code linked in from the `libc` and `libm` libraries, as those regions were not protected.

The binaries were first profiled to see how many times each static instruction is executed. We then used the `libc` `rand` function to select the number of faults to insert into the program. The fault injection rate per dynamic instruction was normalized so that there would be exactly one fault per run on baseline builds. Once the number of faults was chosen, our instrumentation chose, for each fault, a number

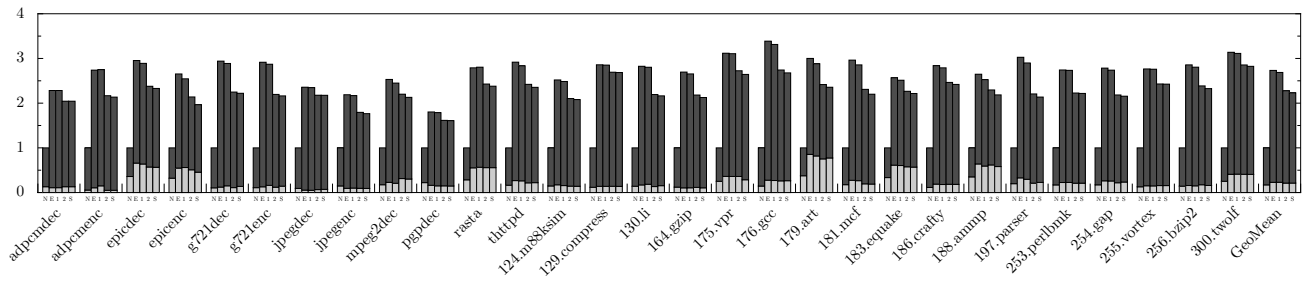


Figure 5. Normalized Dynamic Instruction Counts for EDDI+ECC+CFE (E), SWIFT-BROpti (1), SWIFT-SCFOpti (2), and SWIFT (S) binaries of the benchmarks. The fraction of instructions which are NOPs is denoted by the light-grey regions.

	Correct	Fault Det	Incorrect	Segfault
NOFT	63.10	0.00	15.04	21.86
EDDI+ECC+CFE	15.41	75.38	0.00	9.21
SWIFT	18.04	70.31	0.00	11.65

Table 2. Comparison of fault-detection rates between NOFT, EDDI+ECC+CFE, and SWIFT. The results have a 95% confidence interval of $\pm$1.1%

between zero and the number of total dynamic instructions. We used this number to choose a static instruction using the weights from the profile, and then we choose a specific instance of this static instruction in the dynamic instruction stream to instrument.

The dynamic instruction is instrumented as follows: one of the outputs of the instruction is chosen at random. In these experiments, we restricted ourselves to modifying general purpose registers, floating point registers, and predicate registers. A random bit of this output register is flipped (predicates are considered 1-bit entities). Afterwards, the execution continues normally, and the result of the execution is recorded.

The execution output is also recorded and compared against known good outputs. If both the execution and the check were successful, then the run is entered into the "Correct" column. If the execution fails due to `SIGSEGV` (Segmentation fault due to the access of an illegal address), `SIGILL` (Illegal instruction due to the consumption of a NaT bit generated by a faulting speculative load instruction) or `SIGBUS` (Bus error due to unaligned access) then the run is entered into the "Segfault" column. If the execution fails due to a fault being detected, the run is entered into the "Fault Detected" column. The runs that do not satisfy any of the above are entered into the "Incorrect" column. Each benchmark was run 300 times on test inputs.

These results are tabulated in Figure 5.2. These figures show that the EDDI+ECC+CFE and SWIFT techniques detect all of the faults which yield incorrect outputs. This confirms our earlier claims that EDDI+ECC+CFE and SWIFT will detect all but the most pathological single-upset faults. This is a marked improvement over the NOFT (No Fault Tolerance) builds which can give incorrect results up to almost 50% of the time in some cases (`pgpdec`). Furthermore, the number of segfaults are reduced from 21.86% in NOFT to 9.21% and 11.65% in EDDI+ECC+CFE and SWIFT respectively. This is because some faults which would have resulted in segfaults in the NOFT builds are detected in the SWIFT and EDDI+ECC+CFE builds before the segfault can actually occur.

Despite the injection of a faults into every run, binaries still ran successfully 63%, 15% and 18% of the time for NOFT, EDDI+ECC+CFE, and SWIFT respectively. These results are in accordance with previous research [27, 9] which observed that many faults injected into programs do not result in incorrect output. Note that EDDI+ECC+CFE and SWIFT have significantly lower rates of success because the number of faults injected into each run is higher for the builds with fault detection due to their larger dynamic instruction count.

However, the difference in the correct rates between NOFT and the builds with fault detection is statistically significant, which would indicate that our techniques are a bit overzealous in their fault detection. This fact is very evident in the `129.compress` where the rate of correct execution fell from 65% in NOFT builds to about 1% for EDDI+ECC+CFE and SWIFT builds, the balance largely being made up of faults detected. The reason for this calamitous fall in correct execution is that a large portion of execution time is spent initializing a hash table which is orders of magnitude larger than the input, and so many of the stores are superfluous in that they do not affect the output, but our technique must nevertheless detect faults on these stores, since it cannot know *a priori* whether or not the output will depend on them, as it is an undecidable problem.

There is also a statistically significant difference between the fault detection rate of EDDI+ECC+CFE versus SWIFT. These can be partially attributed to faults in-

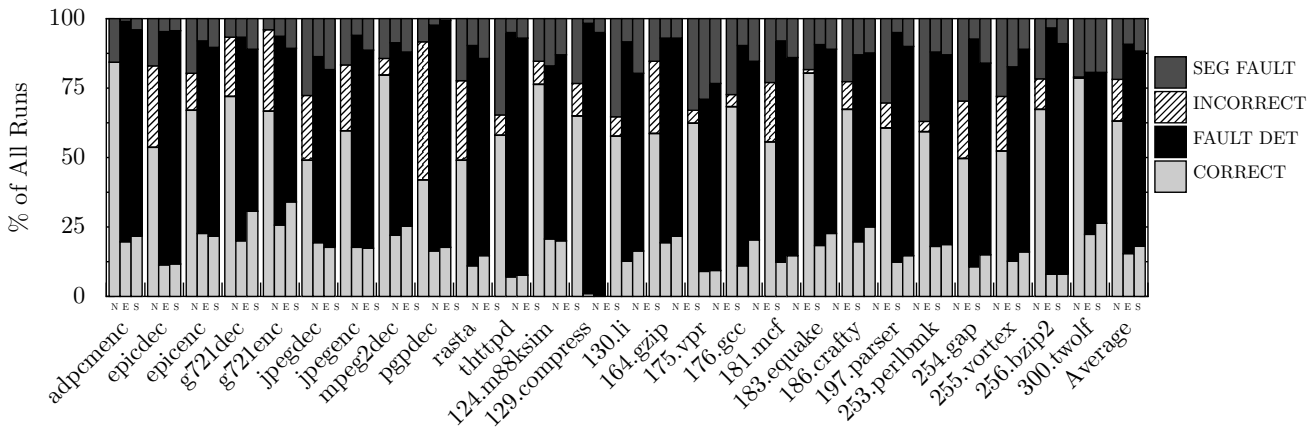


Figure 6. Fault-detection rates by benchmark between NOFT (N), EDDI+ECC+CFE(E) and SWIFT (S). Bars are broken down by percent where a fault was detected (FAULT DET), a segmentation fault occurred (SEG FAULT), incorrect output was produced (INCORRECT), or correct output was produced (CORRECT).

jected on the extra comparison instruction generated by the EDDI+ECC+CFE technique. A fault on these instructions *always* generates a fault detected, whereas a fault on the general population of instruction has a nonzero probability of generating correct output (or a segmentation fault) in lieu of a fault detected. Therefore, SWIFT can be expected to have a slightly lower fault detection rate than EDDI+ECC+CFE because SWIFT binaries do not have extra comparison instructions to fault on.

In addition, the difference in fault detection rates is made up partially by the larger segmentation fault rate in the SWIFT binaries. This can be explained by the larger number of speculative loads in the EDDI+ECC+CFE binaries which result from the larger number of branches around which the compiler must schedule loads. An injected fault which would cause a segfault in SWIFT is transformed in EDDI+ECC+CFE into the insertion of a NaT bit on the output register. This extra NaT bit is then checked at the comparison code and detected as a fault rather than a segfault.

6 Conclusion

This paper demonstrated that detection of most transient faults can be accomplished without the need for specialized hardware. In particular, this paper introduces SWIFT, the best performing single-threaded software-based approach for full out fault detection. SWIFT is able to exploit unused instruction-level parallelism resources present in the execution of most programs to efficiently manage fault detection. Through enhanced control flow checking, validation points needed by previous software-only techniques become unnecessary and SWIFT is able to realize a performance increase. When compared to the best known singled-threaded approach to fault tolerance, SWIFT achieves a 14% speedup. The SWIFT technique described in this paper can be integrated into production compilers to provide fault detection on today's commodity hardware.

Acknowledgments

We thank the entire Liberty Research Group as well as Shubhendu Mukherjee, Santosh Pande, John Sias, Robert Cohn, and the anonymous reviewers for their support during this work. This work has been supported by the National Science Foundation (CNS-0305617 and a Graduate Research Fellowship) and Intel Corporation. Opinions, findings, conclusions, and recommendations expressed throughout this work are not necessarily the views of the National Science Foundation or Intel Corporation.

References

[1] H. Ando, Y. Yoshida, A. Inoue, I. Sugiyama, T. Asakawa, K. Morita, T. Muta, T. Motokurumada, S. Okada, H. Yamashita, Y. Satsukawa, A. Konmoto, R. Yamashita, and H. Sugiyama. A 1.3GHz fifth generation SPARC64 Microprocessor. In *International Solid-State Circuits Conference*, 2003.

[2] T. M. Austin. DIVA: a reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 196–207. IEEE Computer Society, 1999.

[3] R. C. Baumann. Soft errors in commercial semiconductor technology: Overview and scaling trends. In *IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals*, pages 121_01.1 – 121_01.14, April 2002.

[4] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. In

Proceedings of the 30th annual international symposium on Computer architecture, pages 98–109. ACM Press, 2003.

[5] R. W. Horst, R. L. Harris, and R. L. Jardine. Multiple instruction issue in the NonStop Cyclone processor. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 216–226, May 1990.

[6] A. Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors-a survey. *IEEE Transactions on Computers*, 37(2):160–174, 1988.

[7] S. S. Mukherjee, J. Emer, T. Fossum, and S. K. Reinhardt. Cache scrubbing in microprocessors: Myth or necessity? In *Proceedings of the 10th IEEE Pacific Rim International Symposium on Dependable Computing*, March 2004.

[8] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the 29th annual international symposium on Computer architecture*, pages 99–110. IEEE Computer Society, 2002.

[9] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 29. IEEE Computer Society, 2003.

[10] N. Oh and E. J. McCluskey. Low energy error detection technique using procedure call duplication. 2001.

[11] N. Oh, P. P. Shirvani, and E. J. McCluskey. Control-flow checking by software signatures. volume 51, pages 111–122, March 2002.

[12] N. Oh, P. P. Shirvani, and E. J. McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, 51(1):63–75, March 2002.

[13] J. Ohlsson and M. Rimen. Implicit signature checking. In *International Conference on Fault-Tolerant Computing*, June 1995.

[14] OpenIMPACT. Web site: http://gelato.uiuc.edu.

[15] Perfmon: An IA-64 performance analysis tool. http://www.hpl.hp.com/research/linux/perfmon.

[16] J. Ray, J. C. Hoe, and B. Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 214–224. IEEE Computer Society, 2001.

[17] M. Rebaudengo, M. S. Reorda, M. Violante, and M. Torchiano. A source-to-source compiler for generating dependable software. pages 33–42, 2001.

[18] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn. PIN: A binary instrumentation tool for computer architecture research and education. In *Proceedings of the 2004 Workshop on Computer Architecture Education (WCAE)*, pages 112–119, June 2004.

[19] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 25–36. ACM Press, 2000.

[20] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, page 84. IEEE Computer Society, 1999.

[21] N. Saxena and E. J. McCluskey. Dependable adaptive computing systems – the ROAR project. In *International Conference on Systems, Man, and Cybernetics*, pages 2172–2177, October 1998.

[22] P. P. Shirvani, N. Saxena, and E. J. McCluskey. Software-implemented EDAC protection against SEUs. volume 49, pages 273–284, 2000.

[23] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 389–399, June 2002.

[24] T. J. Slegel, R. M. Averill III, M. A. Check, B. C. Giamei, B. W. Krumm, C. A. Krygowski, W. H. Li, J. S. Liptay, J. D. MacDougall, T. J. McPherson, J. A. Navarro, E. M. Schwarz, K. Shum, and C. F. Webb. IBM's S/390 G5 Microprocessor design. In *IEEE Micro*, volume 19, pages 12–23, March 1999.

[25] R. Venkatasubramanian, J. P. Hayes, and B. T. Murray. Low-cost on-line fault detection using control flow assertions. In *Proceedings of the 9th IEEE International On-Line Testing Symposium*, pages 137–143, July 2003.

[26] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery using simultaneous multithreading. In *Proceedings of the 29th annual international symposium on Computer architecture*, pages 87–98. IEEE Computer Society, 2002.

[27] N. Wang, M. Fertig, and S. J. Patel. Y-branches: When you come to a fork in the road, take it. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 56–67, September 2003.

[28] Y. Yeh. Triple-triple redundant 777 primary flight computer. In *Proceedings of the 1996 IEEE Aerospace Applications Conference*, volume 1, pages 293–307, February 1996.

[29] Y. Yeh. Design considerations in Boeing 777 fly-by-wire computers. In *Proceedings of the Third IEEE International High-Assurance Systems Engineering Symposium*, pages 64 – 72, November 1998.