Software-Controlled Fault Tolerance

George A. Reis[1]   Jonathan Chang[1]   Neil Vachharajani[1]
Ram Rangan[1]   David I. August[1]   Shubhendu S. Mukherjee[2]

Traditional fault tolerance techniques typically utilize resources ineffectively because they cannot adapt to the changing reliability and performance demands of a system. This paper proposes software-controlled fault tolerance, a concept allowing designers and users to tailor their performance and reliability for each situation. Several software-controllable fault detection techniques are then presented: SWIFT, a software-only technique, and CRAFT, a suite of hybrid hardware/software techniques. Finally, the paper introduces PROFiT, a technique which adjusts the level of protection and performance at fine granularities through software control. When coupled with software-controllable techniques like SWIFT and CRAFT, PROFiT offers attractive and novel reliability options.

1. INTRODUCTION

In recent decades, microprocessor performance has been increasing exponentially, due in large part to smaller and faster transistors enabled by improved fabrication technology. While such transistors yield performance enhancements, their lower threshold voltages and tighter noise margins make them less reliable [Shivakumar et al. 2002; O'Gorman et al. 1996; Baumann 2001], rendering processors that use them more susceptible to *transient faults*. Transient faults are intermittent faults caused by external events, such as energetic particles striking the chip. These faults do not cause permanent damage, but may result in incorrect program execution by altering signal transfers or stored values. If a fault of this type ultimately affects program execution, it is considered a *soft error*.

Incorrect execution in high-availability and real-time applications can potentially result in serious damage and thus these systems have the highest reliability requirements. These systems will often resort to expensive hardware redundancy to ensure maximum reliability. While several fault tolerance solutions have been proposed for high-end systems, the high hardware costs of these solutions make them less than ideal for the desktop and embedded computing markets.

These lower-end markets do not have reliability requirements that are as stringent as

---

[1] Liberty Research Group
Princeton University
Princeton, NJ 08544
{gareis,jcone,nvachhar,ram,august}@princeton.edu

[2] FACT Group
Intel Corporation
Hudson, MA 01749
shubu.mukherjee@intel.com

those for high-end, high-availability markets. However, during specific periods of time, these systems may be required to perform critical tasks that require high reliability. For instance, end-users will typically spend most of their time on applications that do not need high reliability (reading the news, playing games), but occasionally, they will also need high reliability (managing finances). It is only during this short period of time that high reliability is needed.

Given such infrequent requirements for high reliability, it would be exceedingly prodigal to use the massive hardware redundancy mechanisms used in high-availability systems. Such schemes would only degrade performance, superfluously consume power, and increase manufacturing costs beyond what is needed. *Software-controlled reliability*, which allows software to direct the level of reliability, reduces cost by enabling fault detection only when necessary. This allows the resources that would be unnecessarily applied to fault detection to be leveraged for other computations, thus increasing performance, or shut down, thus saving power.

We begin by presenting four fault detection techniques which allow for software control. Each technique is a point in the design space representing a trade-off between hardware cost, performance, and reliability of the system. The first technique, called SWIFT (SoftWare Implemented Fault Tolerance) [Reis et al. 2005a], increases reliability by inserting redundant code to compute duplicate versions of all register values and inserting validation instructions before control flow and memory operations. The redundant and validation instructions are inserted by the compiler and are used to increase the reliability of systems without any hardware requirements.

The last three techniques are hardware-software hybrid techniques collectively called CRAFT (CompileR Assisted Fault Tolerance) [Reis et al. 2005b]. The CRAFT suite is based on the SWIFT technique augmented with structures inspired by the hardware-only Redundant MultiThreading (RMT) technique [Reinhardt and Mukherjee 2000]. The CRAFT hybrid techniques provide increased reliability and performance over software-only techniques, while incurring significantly lower hardware costs than hardware-only techniques.

While the SWIFT and CRAFT techniques immediately provide application-level control, they also provide the ability to control reliability at a finer level. Just as it may be desirable to only protect a single application on a system, it may be even more desirable to only protect critical sections of that application. Therefore, we also explore techniques to control the amount and type of protection offered by a program at fine granularities in order to best meet the performance and reliability demands of a system as expressed via a utility function. A profile is created to determine the vulnerability and performance trade-offs for each program region and this profile is used to decide where to turn on and off redundant execution. We call this software-controlled fault tolerance technique PROFiT, for profile-guided fault tolerance

For fair, quantitative comparisons between SWIFT, CRAFT, both with and without PROFiT, and other fault-tolerance systems, we measure the *Mean Work To Failure* (MWTF) for each of these systems, using the evaluation methodology first presented in previous work [Reis et al. 2005b]. We analyzed the reliability of the integer register file of an Intel$^{®}$ Itanium$^{®}$ 2 processor for each of the techniques.

We show that the SWIFT software-only technique reduces output-corrupting faults by 9x over a system with no fault detection, from 17.75% to 1.95%, while increasing exe-

cution time by 42.9%. The CRAFT hybrid techniques reduces output-corrupting faults to 0.71%, a 25x decrease, while increasing execution time by 31.4%. Adding PROFiT to these techniques reduces the execution time by an additional 21.2% over the software-only technique and 3.3% over the hybrid techniques (corresponding to normalized execution times of 1.217 and 1.281 relative to no fault tolerance) while maintaining the same level of reliability.

While the SWIFT and CRAFT techniques have been previously introduced [Reis et al. 2005a; Reis et al. 2005b], this paper provides a more accurate and in-depth analysis of the performance and reliability of these systems. This paper introduces software control in fault tolerance and exploits it using the PROFiT technique, which allows designers to further fine-tune the trade-off between performance and reliability. PROFiT demonstrates that software control can be used to build systems tailored to the performance, reliability, and hardware requirements of the application.

The rest of the paper is organized as follows. Section 2 describes related work and Section 3 provides background information for transient fault detection. Section 4 describes the SWIFT software fault-detection system. Section 5 gives a detailed description of the three hybrid CRAFT techniques. Section 6 describes the PROFiT technique for adjusting the level of protection according to profiles. Section 7 describes the new framework for evaluating reliability and introduces the MWTF metric. Section 8 evaluates the SWIFT and CRAFT techniques, both with and without PROFiT. The paper concludes with Section 9.

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 [1988] proposed using a *watchdog* processor to compare and validate the outputs against the main running processor. Austin [1999] proposed DIVA, which uses a main, high-performance, out-of-order processor core that executes instructions and a second, simpler core to validate the execution. Real system implementations like the Compaq NonStop Himalaya [Horst et al. 1990], IBM S/390 [Slegel et al. 1999], and Boeing 777 airplanes [Yeh 1996; 1998] 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 [1998] were the first to use redundant threads to alleviate soft errors. Rotenberg [1999] presented AR-SMT, which used Simultaneous MultiThreading (SMT) for redundancy and leveraged the computations of the leading thread to increase the performance of the trailing thread without loss of reliability. Reinhardt and Mukherjee [2000] 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. [2002] adds fault recovery to RMT by delaying commit and possibly rewinding to a known good state. Mukherjee et al. [2002] proposed a Chip-level Redundantly Threaded multiprocessor (CRT) and Gomaa et al. [2003] expanded upon that approach with CRTR to enable recovery. Ray et al. [2001] proposed modifying an out-of-order superscalar processor's microarchitectural components to implement redundancy. At the functional-unit level Patel and Fung [1982] proposed a method to redundantly compute with shifted operands (RESO) to detect faults to a datapath or arithmetic unit.

All hardware-based approaches require the addition of some form of new hardware logic to meet redundancy requirements. Software-only approaches, on the other hand, are attractive because they come free of hardware cost. Shirvani et al. [2000] proposed a technique to enable ECC for memory data via a software-only technique. Oh and McCluskey [2001] 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. [2001] proposed a source-to-source pre-pass compiler to generate fault detection code in a high level language. Bolchini and Salice [2001] proposed a software-only method for VLIW architectures which used excess instruction slots for redundant execution, but which lacked control-flow checking. Holm and Banerjee [1992] also proposed a software-only technique for VLIW architectures which lacked control-flow checking.

Oh et al. [2002c] proposed a novel software redundancy approach (EDDI) wherein all instructions are duplicated and appropriate "check" instructions are inserted for validation. Oh et al. [2002b] extended this approach with ED^4I which creates a different, but functionally equivalent program by mapping values in the original program to different values in the duplicated program. Dean and Shen [1998] showed how to interleave different threads of execution into a single thread to eliminate context switching overhead. The technique was used to increase multi-application performance, but it could also be used for reliability if the threads to be integrated were an original and redundant version of the same program, and specific comparison instructions were included to check for faults.

Oh et al. also 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 [2002a]. Venkatasubramanian et al. [2003] 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. Schuette and Shen [1994] explored control-flow monitoring (ARC) to detect transient faults affecting the program flow on a Multiflow TRACE 12/300 machine with little extra overhead. Ohlsson and Rimen [1995] 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, unlike the related work, is a complete fault detection technique, incorporating instruction duplication and data validation in a single-threaded context as well as a novel form of control flow verification. The CRAFT techinques, which are based on the SWIFT technique, are the first hybrid validation techniques to target the entire processor core.

3.   PRELIMINARIES

Throughout this paper, we will assume a *Single Event Upset* fault model. That is, we will assume that exactly one bit is flipped exactly once during a program's execution. In this model, any bit in the system at any given execution point can be classified as one of the following [Mukherjee et al. 2003]:

ACE.   These bits are required for *Architecturally Correct Execution* (ACE). A transient fault affecting an ACE bit will cause the program to execute incorrectly.

unACE.   These bits are not required for ACE. A transient fault affecting an unACE bit will not affect the program's execution. For example, unACE bits occur in state elements that hold dynamically dead information, logically masked values, or control flows that are

Y-branches [Wang et al. 2003]

Transient faults in ACE bits can also be further classified by how they manifest themselves in program output.

DUE. A transient fault on an ACE bit that is caught by a fault detection mechanism is a *Detected Unrecoverable Error* (DUE). A detected error can only be considered DUE if it is fail-stop, that is, if the detection occurs before any errors propagate outside a boundary of protection. Obviously, no fault is a DUE in a non-fault-detecting system.

SDC. A transient fault on an ACE bit that is *not* caught by a fault-detection mechanism will cause *Silent Data Corruption* (SDC). This could manifest itself as a spurious exception, an incorrect return code, or corrupted program output. We can further sub-categorize SDC into *potential SDC* (pSDC), faults that cause an exception or an incorrect return code, and *definite SDC* (dSDC), faults that silently corrupt program output [Reis et al. 2005b]. pSDC faults can possibly be detected if the program terminates in a manner that cannot happen under normal execution and the execution did not corrupt any data.

Note that detected, recoverable errors are not considered errors. In this paper, we will sometimes refer to a bit as being DUE or SDC. A DUE bit is an ACE bit which, if flipped by a transient fault, would result in a DUE. Similarly, an SDC bit is an ACE bit which, if flipped, would result in an SDC.

The goal of any fault-detection system is to convert a system's SDC into DUE. Unfortunately, fault-detection systems will have a higher *soft error rate* (SER), the sum of SDC and DUE, than the original system. There are two principal reasons for this. First, most practical fault-detection schemes may exhibit *false DUE*, which arise whenever the system detects a fault in an unACE bit. This occurs because the system may not be able to determine whether a flipped bit is unACE, and thus may have to conservatively signal a fault. Second, any type of fault detection necessarily introduces redundancy, and this increases the number of bits present in the system. Since all bits in the system are susceptible to transient faults, this also leads to a higher soft error rate.

Although redundancy techniques often increase the overall SER, they reduce SDC faults, which are more deleterious than DUE. Consequently, system designers tolerate higher incidents of DUE in order to reduce SDC. A typical SDC rate target is one fault per 1000 years, while a corresponding DUE rate target is two orders of magnitude larger (one fault per 10 years) [Bossen 2002].

To measure a microarchitectural structure's susceptibility to transient faults, the notion of an *architectural vulnerability factor* (AVF) is used and is defined as follows:

$$\text{AVF} = \frac{\text{number of ACE bits in the structure}}{\text{total number of bits in the structure}}$$

Just as ACE bits were further categorized into DUE bits and SDC bits, AVF can be broken up into AVF_{DUE} and AVF_{SDC} by computing the ratio of DUE or SDC bits over the total bits in the structure respectively.

The rest of this paper focuses on fault-detection techniques, although fault recovery may also be desirable. Fault-detection techniques can often be extended to enable fault recovery, as shown by the recovery techniques, SRTR and CRTR, that have been derived from the detection-only redundant threading techniques SRT and CRT.

The techniques in this paper can tolerate transient faults that occur in the processor core, including the processor's pipeline, functional units, register state, etc. We define a

Sphere of Replication (SoR) [Reinhardt and Mukherjee 2000] to be the region in which our technique tolerates faults. The Sphere of Replication in our system is the boundary between the processor and memory, including the cache. The memory subsystem and caches are not protected by these techniques, as they can be effectively protected by error correcting codes (ECC).

4. SOFTWARE-ONLY FAULT DETECTION

In this section, we will describe our first low-cost fault-tolerance technique, SWIFT, a software-only redundancy scheme. In this technique, the compiler enables program protection by building redundancy directly into the compiled code. The redundant codes do not require any special microarchitectural hardware for their execution. The SWIFT technique has two components - instruction duplication with validation and control flow checking.

4.1 Instruction Duplication

The SWIFT-enabled compiler duplicates the original program's instructions and schedules them along with the original instructions in the same execution thread. The original and duplicate versions of instructions are register-allocated so that they do not interfere with one another. At certain synchronization points in the combined program, validation code sequences are inserted by the compiler to ensure that the data values being produced by the original and redundant instructions agree with each other.

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 necessary to use store instructions as synchronization points for comparison.

Figure 1 shows a sample code sequence before and after the SWIFT fault-detection transformation. The add instruction is duplicated and inserted as instruction **3**. The duplicate instruction uses redundant versions of the values in registers r2 and r3, denoted by r2' and r3' respectively. The result is stored in r1's redundant version, r1'.

Instructions **1** and **2** are inserted to validate and replicate the data of the load instruction. Instruction **1** is a comparison inserted to ensure that the address of the subsequent load matches its duplicate version, while instruction **2** copies the result of the load instruction into a duplicate register.

The values of r1 and r2 are used at the store instruction at the end of the example. Since it is necessary to avoid storing incorrect values into memory or storing values to incorrect addresses, we must check that both the address and value match their redundant copy. If a difference is detected, then a fault has occurred and the appropriate handling code, whether that be exiting or restarting the program or simply notifying another process, is executed at instructions **4** or **5**. Otherwise, the store may proceed as normal.

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 Instruction Level Parallelism (ILP) thus minimizing the performance penalty of the transformation.

4.2 Control-Flow Checking

Unfortunately, it is insufficient to only compare the inputs of store instructions since misdirected branches can cause stores to be skipped, incorrect stores to be executed, or incorrect values to be ultimately fed to a store. To extend fault detection coverage to cases where

```
                                       1:  br faultDet, r4 != r4'
    ld r3 = [r4]                           ld r3 = [r4]
                                       2:  mov r3'= r3
    add r1 = r2, r3                        add r1 = r2, r3
                                       3:  add r1'= r2',r3'
                                       4:  br faultDet, r1 != r1'
                                       5:  br faultDet, r2 != r2'
    st [r1] = r2                           st [r1] = r2
```

(a) Original Code   (b) SWIFT Code

Fig. 1.   Duplication and Validation

branch instruction execution is compromised, we propose a control flow checking transformation.

Explicit control-flow checking is not necessary in redundant multi-threading approaches because each thread is executed with an independent program counter (PC). A fault that causes one thread to follow a different control path will not divert the other thread. Therefore, the fault will be detected when different store instructions or values are about to be written to memory. For techniques that use redundant instructions within a single thread, such as SWIFT, additional control-flow validation is needed because there is no redundant hardware PC.

To verify that control is transferred to the appropriate control block, each block will be assigned a unique signature. A designated general purpose register, which we will denote pc', will hold the signature of the current block and will be used to detect control flow faults. For every control transfer, the source block asserts the offset to its target using another register, sigoff, and each target confirms the transfer by computing the pc' from this offset and comparing it to the statically assigned signature. Conceptually, the pc' serves as a redundant copy of the program counter, while sigoff functions as a redundant computation of the control transfer. This control flow checking will catch faults which divert the control flow to the incorrect direction of a branch, or to a completely invalid branch target.

Consider the program shown in Figure 2. Instructions **6** and **12** are the redundant computations of the original add instructions. Instruction **8** uses an xor instruction to compute the relationship between the signature of the current block (sig0) and the signature of the branch target (sig1) and stores this result in sigoff.

Since the original branch is guarded by the condition r1==r5, the sigoff redundant control-flow computation should be guarded by the redundant condition r1'==r5'. This is handled via instruction **7**. Instruction **10**, at the target of a control transfer, xors sigoff with the pc' to compute the signature of the new block (sig1). This signature is compared with the statically assigned signature in instruction **11** and a fault is detected if they mismatch.

Notice that with this transformation, any faults that affect branch execution will be detected since pc' will eventually contain an incorrect value. Therefore, this control transformation robustly protects against transient faults. As a specific example, suppose a transient fault occurred to r5 so that r5 incorrectly equals r1 and the branch instruction spuriously executes. The control flow checking detects this case because the duplicate versions, r1' and r5', will not be equal and so the sigoff register will not be updated. After instruction **10** executes, the pc' will contain the wrong value and an error will be

```
        add r1 = r1, r3                     add r1 = r1, r3
                                    6:      add r1'= r1',r3'
                                    7:      br L1', r1' != r5'
                                    8:      xor sigoff = sig0,sig1
                                    9:      L1':
        br L1, r1 == r5                     br L1, r1 == r5
        ...                                 ...
        L1:                                 L1:
                                    10:     xor pc' = pc',sigoff
                                    11:     br faultDet, pc' != sig1
        add r6 = r1, r2                     add r6 = r1, r2
                                    12:     add r6'= r1',r2'

            (a) Original Code                   (b) SWIFT Code
```

Fig. 2.   Control Flow Checking

signaled at instruction **12**.

4.3 Undetected Errors

Although SWIFT's protection is quite robust, there are two *fundamental* limitations to SWIFT:

(1) Since redundancy is introduced solely via software instructions, any time a fault affects the value used by the store without affecting the value used by the compare, an error may go undetected. For simple microarchitectures, this is largely equivalent to saying that an error may go undetected if a fault occurs after the validation instruction and before the store instruction. In more complicated microarchitectures, errors to a register may manifest themselves on some instructions and not others due to the modern design of instruction windows and bypass networks. In these situations, it is more difficult to enumerate the conditions under which a store can be affected without its corresponding validation instructions being affected.

(2) A strike to an instruction's opcode bits may change a non-store instruction into a store instruction. Since store instructions are not duplicated by the compiler, there is no way of detecting and recovering from this situation. The incorrect store instruction will be free to execute and the value it stores will corrupt memory and can result in silent data corruption. A non-branch instruction may be converted to a branch instruction by a single bit flip, in which case the program is susceptible to the control-flow issues discussed below.

   The aforementioned errors are fundamental errors in the sense that every software-only scheme (and even many hardware-only schemes) will be plagued by them to some degree. In addition to these errors, there are four vulnerabilities which are a function of SWIFT's implementation:

(3) If a control-flow error occurs such that the program branches directly to a store instruction, system call instruction, or other potentially disastrous instruction before the signature can be verified, then a potentially erroneous value may propagate to memory. The effect of this can be minimized by performing validation checks as close as possible to every potentially output-corrupting instruction. This will reduce the window of vulnerability, although not eliminate it, as well as increase performance overhead.

(4) Intra-block control-flow errors may also go undetected. For example, in Figure 2, if instruction **7** is affected such that it jumps to the first add instruction of the block, immediately before **6**, the control signatures will be unchanged and still valid, but the original and redundant additions will be incorrectly computed one extra time, corrupting the values of `r1` and `r1'`. This type of limitation cannot be avoided by adding extra validation instructions, but can be minimized by the compiler's scheduling algorithm [Oh et al. 2002a].

(5) Our control-flow scheme may also not be able to protect against all errors on indirect call/branch computations. To ameliorate the situation somewhat, one could simply duplicate the computation path of the call target and compare the original and redundant call targets before executing the branch. In other situations, this vulnerability can be easily closed such as when the call target is computed via a jump table (such as those used to implement `switch` statements). One simply needs to have the redundant computation consult a table of signatures instead of the original jump table.

(6) Instead of duplicating load instructions, SWIFT simply adds a move instruction to copy the loaded value into a redundant register, as in instruction **2** of Figure 1. Before the copy is performed, the program only has one version of the loaded value. If a fault occurs to this value before it is copied, then the incorrect value will be propagated to the redundant register. Since both copies will now be corrupted, the error will go undetected.

4.4 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.[1]

4.5 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 simply to make function calls synchronization points. Before any function call, all input operands are checked against their redundant copies. If any 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 then be duplicated into redundant versions for the remaining redundant code to function.

This synchronization adds performance overhead and 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.

---

[1] Although we do not give the details here, it is clear that the probability of n upset events decreases exponentially with n.

```
1:  br faultDet, r1 != r1'
2:  br faultDet, r2 != r2'                         3:  st [r1] = r2
3:  st [r1] = r2                                    4:  st'[r1']= r2'

        (a) SWIFT Code                                  (b) CRAFT:CSB Code
```

Fig. 3.   The CRAFT:CSB transformation.

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. However, only arguments passed in registers need be duplicated. Arguments that are passed via memory do not need to be replicated because the loads and stores will be validated during the normal course of execution as explained in Section 4.1.

Doubling the number of arguments and return values 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.

5.   HYBRID REDUNDANCY TECHNIQUES

This section presents three low-cost hybrid hardware/software redundancy techniques called CRAFT (*CompileR-Assisted Fault Tolerance*). CRAFT is built on top of SWIFT, with minimal hardware adaptations from RMT to create systems with near-perfect reliability, low performance degradation, and low hardware cost. A more detailed description of these techniques can be found in previous work [Reis et al. 2005b].

The CRAFT techniques, besides benefiting from the redundant execution provided by SWIFT, use low-cost microarchitectural enhancements to tackle the cases where software-only techniques cannot guarantee fault coverage.

5.1   CRAFT: Checking Store Buffer (CSB)

As noted in Section 4.3, in SWIFT, stores are single points-of-failure and make the application vulnerable to strikes in the time interval between the validation and the use of a register values. For example, consider the code snippet given in Figure 3(a). If r1 receives a strike after instruction 1 is executed but before instruction 3 is executed, then the value will be stored to the incorrect address. Similarly, if a fault occurs on r2 after instruction 2 but before instruction 3 then an incorrect value will be stored to memory.

In order to protect data going to memory, in the CRAFT:CSB technique, the compiler duplicates store instructions in the same way that it duplicates all other instructions, except that store instructions are also tagged with a single-bit version name, indicating whether a store is an original or a duplicate. Figure 3(b) shows the result of this transformation. The original store (instruction 3) is duplicated and tagged to form instruction 4.

Code thus modified is then run on hardware with an augmented store buffer called the *Checking Store Buffer* (CSB). The CSB functions much as a normal store buffer, except that it does not commit entries to memory until they are validated. An entry becomes validated once the original and the duplicate version of the store have been sent to the store buffer, and the addresses and values of the two stores match perfectly.

```
1:  br faultDet, r4 != r4'
2:  ld r3 = [r4]                          2:  ld r3 = [r4]
3:  mov r3' = r3
                                          4:  ld r3'= [r4']
```

       (a) SWIFT Code   (b) CRAFT:LVQ Code

Fig. 4.   The CRAFT:LVQ transformation.

The CSB can be implemented by augmenting the normal store buffer with a tail pointer for each version and a `validated` bit for each buffer entry. An arriving store first reads the entry pointed to by the corresponding tail pointer. If the entry is empty, then the store is written in it, and its `validated` bit is set to false. If the entry is not empty, then it is assumed that the store occupying it is the corresponding store from the other version. In this case, the addresses and values of the two stores are validated using simple comparators built into each buffer slot. If a mismatch occurs, then a fault is signaled. If the two stores match, the incoming store is discarded, and the `validated` bit of the already present store is turned on. The appropriate tail pointer is incremented modulo the store buffer size on every arriving store instruction. When a store reaches the head of the store buffer, it is allowed to write to the memory subsystem if and only if its `validated` bit is set. Otherwise, the store stays in the store buffer until a fault is raised or the corresponding store from the other version comes along.

The store buffer is considered *clogged* if it is full and the validated bit at the head of the store buffer is unset. Note that both tail pointers must be checked when determining whether the store buffer is full, since either version 1 or version 2 stores may be the first to appear at the store buffer. A buffer clogged condition could occur because of faults resulting in bad control flow, version bit-flips, or opcode bit-flips, all of which result in differences in the stream of stores from the two versions. If such a condition is detected at any point, then a fault is signaled. To prevent spurious fault signaling, the compiler must ensure that the difference between the number of version 1 stores and version 2 stores at any location in the code does not exceed the size of the store buffer.

The use of hardware to do the validation allows us to optimize the generated code as we can now remove the software validation code (instructions 1 and 2). These modest hardware additions allow the system to detect faults in store addresses and data, as well as dangerous opcode bit-flips, thus protecting against vulnerabilities (1) and (2) mentioned in Section 4.3. Although this technique duplicates all stores, no extra memory traffic is created, since there is only one memory transaction for each pair of stores in the code. Furthermore, CRAFT:CSB code also exhibits greater scheduling flexibility since each pair of stores and the instructions they depend on can now be scheduled independently, whereas in SWIFT, store instructions are synchronization points. The net result is a system with enhanced reliability, higher performance, and only modest additional hardware costs.

As we will further explore in Section 8, with the CRAFT:CSB transformation, the AVF_{SDC} is reduced by 96.0% versus 89.0% for SWIFT, while performance degradation is reduced to 33.4% versus 42.9% for SWIFT.

5.2 CRAFT: Load Value Queue (LVQ)

In SWIFT, load values need to be duplicated to enable redundant computation. SWIFT accomplishes this by generating a move instruction after every load. This is shown as instruction 3 in Figure 4(a). Furthermore, the load address must be validated prior to the load; this validation is done by instruction 1.

This produces two windows of vulnerability, namely vulnerabilities (1) in (6) in Section 4.3. If a fault occurs to r4 after instruction 1 but before instruction 2, then the load instruction will load from an incorrect address. If a fault occurs to r3 after instruction 2 but before instruction 3, then a faulty value will be duplicated into both streams of computation and the fault will go undetected.

In order to remove these windows of vulnerability, we must execute a redundant load instruction and apply the transformation shown in Figure 4(b). Instead of copying the value from a single load (instruction 3), we would ideally simply execute a second, redundant load (instruction 4). Unfortunately, merely duplicating load instructions will not provide us with correct redundant execution in practice. Treating both versions of a load as normal loads will lead to problems in multi-programmed environments as any intervening writes by another process to the same memory location can result in a false DUE. In such cases, the technique prevents the program from running to completion even though no faults have been actually introduced into the program's execution.

We make use of a protected hardware structure called the *Load Value Queue* (LVQ) to enable redundant load execution. The LVQ only accesses memory for the original load instruction and bypasses the load value for the duplicate load from the LVQ. An LVQ entry is deallocated if and only if both the original and duplicate versions of a load have executed successfully. A duplicate load can successfully bypass the load value from the LVQ if and only if its address matches that of the original load buffered in the LVQ. If the addresses mismatch, a fault has occurred and we signal a fault. Loads from different versions may be scheduled independently, but they must maintain the same relative ordering across the two versions. Additionally, for out-of-order architectures, the hardware must ensure that loads and their duplicates both access the same entry in the load value queue.

The duplicated loads and the LVQ provide completely redundant load instruction execution. Since the address validation is now done in hardware, the software address validation in instruction 1 of Figure 4 can now be removed. The LVQ also allows the compiler more freedom in scheduling instructions around loads, just as the checking store buffer allows the compiler to schedule stores more freely. Since the duplicate load instruction will always be served from the LVQ, it will never cause a cache miss and or additional memory bus traffic.

With the CRAFT:LVQ transformation, the AVF_{SDC} is reduced by 89.4% versus 89.0% for SWIFT, while performance degradation is reduced to 37.6% versus 42.9% for SWIFT.

5.3 CRAFT: CSB + LVQ

The third and final CRAFT technique duplicates both store and load instructions and adds both the checking store buffer and the load value queue enhancements simultaneously to a software-only fault detection system such as SWIFT.

Combining the CSB and the LVQ yields a system which reduces SDC by 90.5% compared to 89.0% for SWIFT. Furthermore, CRAFT:CSB+LVQ's performance degradation is reduced to 31.4% versus 42.9% for SWIFT.

6. PROFIT

SWIFT and CRAFT, while being able to greatly reduce the number of undetected errors, also have the feature of being software-controllable. This makes it possible for a software-controlled fault tolerance technique to precisely manage the level of performance and reliability within a program. An instance of such a technique is PROFiT, an algorithm which uses a program's reliability profile to fine-tune the tradeoff between protection and performance. PROFiT relies on the fact that regions of code may differ intrinsically in terms of reliability. Regions can differ in their natural robustness against transient faults, and they can differ in their response to various fault-detection schemes. For example, a function which computes a high degree of dynamically dead or logically masked data may naturally mask many transient faults and not require much protection. On the other hand, a function which is dominated by control flow and which contains few dynamic instances of store instructions may not need any protection for stores but may instead require large amounts of control-flow checking.

In general, programs will exhibit a wide variety of reliability behaviors. A one-size-fits-all application of any given reliability technique will be either inefficient and over-protect some regions or will leave other regions unnecessarily vulnerable. Customizing reliability for each of these regions is the goal of the PROFiT algorithm. By tailoring the reliability to the particular qualities of the region in question, we can obtain reliability comparable to any of the aforementioned protection techniques while simultaneously improving performance.

PROFiT operates under the supposition of the existence of a user-defined *utility function*. This utility function should serve as a metric for how desirable a particular version of a program is. For this paper, we will consider utility functions that are functions of the execution time of the program, the AVF_{SDC} of the program, and the AVF_{DUE} of the program. These three factors are used because they are usually regarded as the most critical design decisions of a system with fault detection. However, the techniques we will illustrate can be easily extended to utility functions that are functions of other factors such as static size and power.

6.1 Optimization

PROFiT is an optimization problem; a variety of optimization techniques have been devised to solve such problems. For the purposes of this paper, we will make use of gradient descent. Gradient descent in general, like most optimization algorithms may become trapped in a local minimum and consequently not find the global minimum. The general procedure works as follows. For each iteration, given $\mathbf{x_n}$, the previous iteration's result, an $\mathbf{x}'$ is found such that

$$\nabla_{\mathbf{x}'} U(\mathbf{x_n}) = \max_{\mathbf{x}} \nabla_{\mathbf{x}} U(\mathbf{x_n})$$

where $\mathbf{x}$ is chosen from some set of vectors, usually the unit circle. If $\nabla_{\mathbf{x}'} U(\mathbf{x_n}) \leq 0$ for all $\mathbf{x}'$, then the procedure is complete and $\mathbf{x_n}$ is the local maximum. Otherwise, $\mathbf{x_n}$ is "adjusted" by $\mathbf{x}'$ to yield the next iteration's result, $\mathbf{x_{n+1}}$, and the procedure is repeated.

To see how this applies to optimization of reliability and performance, we will introduce the notation $X \setminus F_Y$ to represent version X of a program with all instances of function F replaced with F from version Y. For example, $X \setminus \texttt{main}_V$ will mean "replace `main` in X with the `main` from binary V." In our specific application, different versions of a function will correspond to functions with varying levels of protection. We will also use

(1) $X \leftarrow P_{\text{NOFT}}$
(2) $\nabla U \leftarrow 0$
(3) foreach $(F, V) \in (\text{functions}(P), \text{versions}(P))$
(4)   if $U(X \setminus F_V) - U(X) < \nabla U$
(5) $\nabla U \leftarrow U(X \setminus F_V) - U(X)$
(6) $F^*_{V*} \leftarrow F_V$
(7) end foreach
(8) if $\nabla U > 0$ then
(9) $X \leftarrow X \setminus F^*_{V*}$
(10)   goto 2
(11) otherwise
(12) $P_{\text{PROFiT}} \leftarrow X$

Fig. 5.   The PROFiT algorithm.

"functions(P)" to denote the set of functions in P, and "versions(P)" to denote the set of versions of P. Our algorithm for a program P is given in Figure 5.

Every gradient descent must begin with an initial guess at the solution. In our case, a simple choice for the initial guess would be the normal, unprotected version of the binary. Finding the point at which the gradient is minimized is simple in this case, because the set of possible directions in which one can move is always finite. We can simply traverse the entire space searching for the adjacent vector which nets the largest increase in utility, which occurs in steps 3 through 7. Line 2 initializes a ∇U which is used to track the largest increase in utility among all functions and versions. For each tuple consisting of a function and a version, the utility of swapping in that tuple is computed. The increase in utility is compared against ∇U in line 4. If the proposed swap is better than any heretofore seen swap, then line 5 records the new change in utility while line 6 records the swap. Finally, in line 8, we check if there were any proposed swaps which actually improved utility. If not, we can terminate the loop. Otherwise, we reiterate.

6.2 Estimating Utilities

The inner loop of the above procedure compares the utility of swapping each function with one function from another version of the program. To be tractable, this relies on our ability to quickly estimate the change in utility, namely $U(X \setminus F_V) - U(X)$. Or more precisely, since U is a function of the execution time, AVF_{SDC}, and AVF_{DUE} of the program, we need only to be able to estimate those three values quickly at the function granularity.

Such an estimation can be done quickly if we assume that all functions are decoupled with respect to those three variables. In other words, replacing a function with a different version of that function should only affect the performance, AVF_{SDC}, and AVF_{DUE} *of that function*. When we speak of the AVF of a function, we refer to the contribution to the system's overall AVF of those microarchitectural state bits that exist during the execution of that function.

For performance, the major inter-function effects on performance are the state of the cache and the branch predictors. However, since all accesses to memory are identical in all versions of the program, the cache will remain largely unaffected. Branch prediction cannot be decoupled so easily, but we can assume that its effects are small and that the change in the runtime of a function between different levels of protection is dominated by the execution of the duplicate instructions.

In the software and hybrid techniques, functions are synchronization points, so the AVF_{SDC} and AVF_{DUE} will be largely decoupled as well. As explained in Section 4.5, the caller of a function compares the arguments of the function to the duplicate version and reports a fault if a mismatch occurs. Otherwise, it passes the original version of the arguments to the function. The body of the function then duplicates its operands so that a redundant thread of execution can occur within the function itself. Therefore, if a fault occurs in the operand to a function, it must be detected by the caller, otherwise the fault will go undetected. Similarly, if a fault occurs in a return value of a function, it must be detected by the function itself, otherwise the fault will go undetected. Since memory does not have a redundant duplicate, any errant stores within the function will not be detected outside the function.

The same can be said for erroneous control-flow changes which branch to an address outside of the current function. If a function erroneously branches to a function with control-flow checking, then the fault will be caught by the second function. On the other hand, if a function erroneously branches to a function without control-flow checking, then the fault may never be caught. In either case, the result is independent of whether or not the first function has control-flow checking. In summary, if a faulty value in a function is not detected by that function, it will never be detected by any other function, which means that the AVF_{SDC} and AVF_{DUE} of each function is decoupled from all other functions.

Due to this decoupling, we can write the following simple relations:

$$t_{X \setminus F_V} = t_X + t_{F_V} - t_{F_X}$$
$$SDC_{X \setminus F_V} = SDC_X + SDC_{F_V} - SDC_{F_X}$$
$$DUE_{X \setminus F_V} = DUE_X + DUE_{F_V} - DUE_{F_X}$$

where F_X denotes the version of function F in X. Therefore, we can quickly calculate the utility of each permutation so long as we know t_{F_V}, SDC_{F_V}, and DUE_{F_V} for each function and reliability version. In the following section, we will present a methodology for rapidly measuring the SDC_{F_V} and DUE_{F_V} of each function.

7. A METHODOLOGY FOR MEASURING RELIABILITY

Mean Time To Failure (MTTF) and AVF are two commonly used metrics to encompass reliability. However, MTTF and AVF are not appropriate in all cases. In this section, to make the evaluation of the techniques in this paper possible, we present Mean Work To Failure, a metric that generalizes MTTF to make it applicable to a wider class of fault-detection systems. We also provide a new framework to accurately and rapidly measure reliability using fault injection and programs run to completion.

7.1 Mean Work To Failure

MTTF is generally accepted as the appropriate metric for system reliability. Unfortunately, this metric does not capture the trade-off between reliability and performance. For example, suppose that system A and system B were being compared, and that system A were twice as fast but half as "reliable" as system B. Specifically, let

$$\text{MTTF}_A = \frac{1}{2} \cdot \text{MTTF}_B \qquad\qquad t_A = \frac{1}{2} \cdot t_B$$

We ask what the probability of a fault occurring during the execution of the program is for each system. Although B has double the MTTF of A, it must run for twice as long, and

so the probability of failure *for the program* is equal for A and B. In this sense, they are equally reliable. However, the MTTF metric would counterintuitively select the slower of the two systems, B, as the most reliable. In order to address this, Weaver et al. introduced the alternative *Mean Instructions To Failure* (MITF) metric [Weaver et al. 2004].

While this metric does capture the trade-off between performance and reliability for hardware fault-tolerance techniques (i.e. those which do not change the programs being executed, but which may affect IPC), it is still inadequate for the general case where the program binary, and hence the number of instructions committed, can vary.

To adequately describe the reliability for hardware *and* software fault-tolerance techniques, we introduce a generalization of MITF called *Mean Work To Failure* (MWTF).

$$\text{MWTF} = \frac{\text{amount of work completed}}{\text{number of errors encountered}} = \left(\text{raw error rate} \times \text{AVF} \times \text{execution time}\right)^{-1}$$

The execution time corresponds to the time to complete one unit of work. A unit of work is a general concept whose specific definition depends on the application. The unit of work should be chosen so that it is consistent across evaluations of the systems under consideration. For example, if one chooses a unit of work to be a single instruction, then the equation reduces to MITF. This is appropriate for hardware fault-detection evaluation because the program binaries are fixed and an instruction represents a constant unit of work. In a server application it may be best to define a unit of work as a transaction. In other cases, work may be better defined as the execution of a program or a suite of benchmarks. With this latter definition of work, it is obvious that halving the AVF while doubling execution time will not increase the metric. Regardless of the method (hardware or software) by which AVF or execution time is affected, the metric accurately captures the reliability or the system. Such a metric is crucial when comparing hardware, software, and hybrid systems.

7.2 Measuring MWTF

To compute the MWTF, one must have an estimate for the number of errors encountered while running a program a fixed number of times or, alternatively, the AVF of the system. We present a framework that improves upon the speed and accuracy of existing AVF measurement techniques, especially for software-only and hybrid techniques. Currently, researchers generally use one of two methods for estimating AVF.

The first method involves labeling bits as unACE, SDC, or DUE and running a detailed simulation to measure the frequency of each type of bit for a given structure. This methodology is problematic when used to analyze software-only or hybrid systems. In these cases, since it is often difficult to categorize bits as either unACE, SDC or DUE, conservative assumptions are made. In SWIFT, for example, many opcode bits are DUE bits because changes to instructions will be caught on comparison, but certain corner cases, like those that change an instruction into a store instruction, are SDC bits. An opcode bit flip that changes an add instruction into a subtract may cause an SDC outcome, but if it changes a signed add to an unsigned add, it may not. Identifying all these cases is non-trivial, but conservatively assuming that all opcode bits are SDC may cause many DUE and unACE bits to be incorrectly reported as SDC. The resulting SDC-AVF from this technique is then a very loose upper bound. AVF categorization has been successfully used for evaluating hardware fault-detection systems that cover all single-bit errors and have few corner
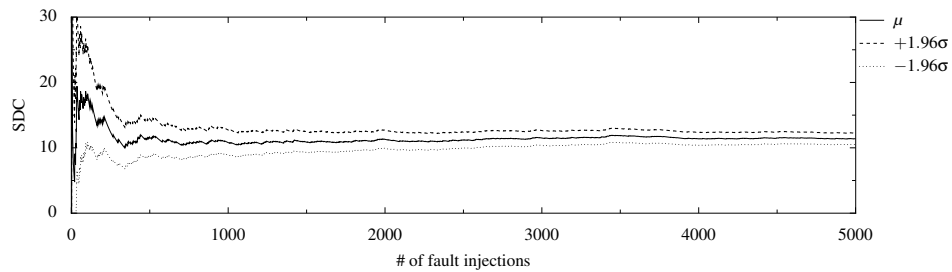
Fig. 6. SDC vs. number of fault injections for 124.m88ksim. The mean SDC (middle line) rapidly converges to the final value. The upper and lower lines represent the bounds of the 95% confidence interval.

cases. When used to compare software or hybrid techniques, the complexity of correctly categorizing faults becomes a burden.

Another method, fault injection, provides an alternative. Fault injection is performed by inserting faults into a model of a microarchitecture and then observing their effects on the system [Czeck and Siewiorek 1990; Kim and Somani 2002; Wang et al. 2004]. Since the microarchitectural models required for this are very detailed, the simulation speeds are often too slow to execute entire benchmarks, let alone benchmark suites. It is common to simulate for a fixed number of cycles and then to compare the architectural state to the known correct state. Such a comparison is useful for determining if microarchitectural faults affect architectural state. However, this comparison does not precisely indicate whether a flipped bit is unACE, SDC, or DUE.

To avoid these shortcomings, we inject faults *and* run all benchmarks to completion. In conventional fault injection systems, this would lead to unmanageable simulation times. However, a key insight facilitates tremendous simulation speedups. We recognize that all microarchitectural faults will have no effect unless they ultimately manifest themselves in architectural state. Consequently, when a particular fault only affects architectural state, detailed cycle-accurate simulation is no longer necessary and a much faster functional (ISA) model can be used.

Our technique consists of two phases. The first involves a detailed microarchitectural simulation. During the microarchitectural simulation, bits in a particular structure in which we desire to inject faults are randomly chosen. Each chosen bit is then tracked until all effects of the transient fault on this bit manifest themselves as architectural state. The list of architectural state that would be modified is recorded for the next phase. The second phase uses architectural fault simulation for each bit chosen during the first phase by altering the affected architectural state as determined by the microarchitectural simulation. The program is run to completion and the final result is verified to determine the fault category of the bit: unACE, DUE, or SDC.

To show that randomly chosen fault injections do indeed converge to an accurate value, we performed 5000 fault-injection runs for the benchmark 124.m88ksim. More details on our fault injections will be given in Section 8. For now, simply note that Figure 6 shows the SDC, and its 95% confidence interval, as a function of the number of fault injections. The confidence interval is $\pm$ 2.00% after 946 fault injections, $\pm$ 1.50% after 1650 fault injections, and $\pm$ 1.00% after 3875 fault injections. The mean stabilizes very quickly; we have found that on the order of 1000 fault injections are required to obtain a reasonably

Name	Description	PROFIT	SWPROFIT	CRPROFIT
NOFT	No fault detection	X	X	X
SWIFT:ST	Software-only store checking	X	X	
SWIFT:CF	Software-only control-flow checking	X	X	
SWIFT	SWIFT:ST + SWIFT:CF	X	X	
CRAFT:CSB	SWIFT + Checking Store Buffer	X		X
CRAFT:LVQ	SWIFT + Load Value Queue	X		X
CRAFT:CSB+LVQ	CRAFT:CSB + CRAFT:LVQ	X		X

Table I.   Summary of techniques evaluated.

accurate estimate of SDC, although more injections could certainly be run if an experiment were to require a higher degree of accuracy.

8. EVALUATION

This section evaluates the SWIFT and CRAFT techniques, both with and without the PROFiT algorithm applied to them, comparing the performance, reliability and hardware cost of these options. Since SWIFT consists of two orthogonal techniques, namely store checking and control-flow checking, we generated binaries having only store checking code (SWIFT:ST), binaries having only control-flow checking code (SWIFT:CF), as well as binaries having both store checking and control-flow checking (SWIFT). We also evaluated all three hybrid techniques: CRAFT with the Checking Store Buffer (CRAFT:CSB), CRAFT with the Load Value Queue (CRAFT:LVQ), and CRAFT with both structures (CRAFT:CSB+LVQ).

We applied the PROFiT algorithm three times, each time allowing it to choose from a different set of reliability options. We allowed PROFiT to select from the entire space of reliability options (PROFIT), only software-only techniques (SWPROFIT), or only hybrid techniques (CRPROFIT). This is summarized in Table I.

8.1 PROFiT Utility Function

Although PROFiT can be applied to any arbitrary utility function, we will now present a particularly salient utility function which was used for evaluation purposes. We begin with the goal of minimizing the product of the execution time and the inverse of the MWTF:

$$U(X) = -t_X^2 \cdot \text{AVF}_X \propto \frac{t_X}{\text{MWTF}_X}$$

where t_X is the execution time of a program on a particular permutation of functions, X. This function captures both the reliability of the system in terms of MWTF_X and the performance of the system in terms of t_X. Most designers, however, will make a distinction between AVF_{DUE} and AVF_{SDC}. Typically, server architects will accept a DUE rate of once per 10 years while only accepting an SDC rate of once per 1000 years [Bossen 2002]. Therefore, in our utility function, we make SDC 100 times more important than DUE. Finally, we take the negative in order to transform the minimization problem into a maximization problem (i.e. the typical sense of utility).

With the available reliability options, the utility function we choose to maximize is then:

$$U(X) = -t_X^2 \cdot (\text{AVF}_{\text{SDC}_X} + \frac{\text{AVF}_{\text{DUE}_X}}{100})$$
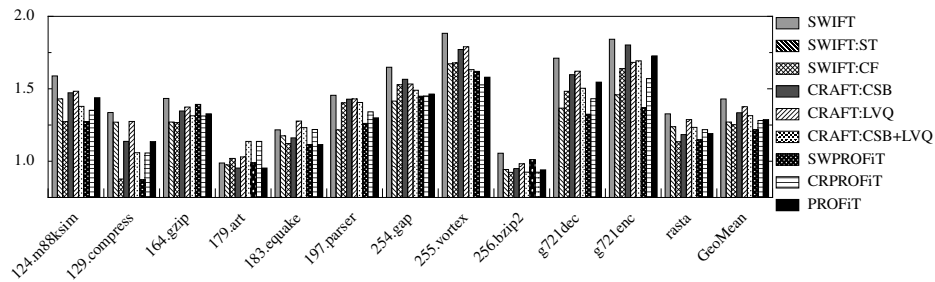
Fig. 7. Normalized execution times by system and benchmark.

8.2 Performance

In order to evaluate the performance of the all of the techniques, we generated redundant codes by modifying a pre-release version of the OpenIMPACT compiler for the IA-64 architecture targeting the Intel® Itanium® 2 processor. The baseline binaries are aggressively optimized OpenIMPACT compilations without any additional redundancy.

We evaluated the performance for a set of benchmarks drawn from SPEC CPUINT2000, SPEC CPUFP2000, SPEC CPUINT95, and MediaBench [Lee et al. 1997] suites. Performance numbers were obtained by running the resulting binaries 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 was used to measure the CPU cycles for each benchmark executed.

The results in Figure 7 show the execution times for SWIFT, CRAFT, and PROFiT binaries normalized to the baseline execution runs. The normalized execution times for SWIFT have a geometric mean of 1.429, while the performance for each of the CRAFT techniques were 1.334, 1.376, and 1.314 for the addition of the CSB, LVQ, and both CSB and LVQ respectively. The hybrid CRAFT techniques perform better than the software-only SWIFT technique because the use of hardware structures in CRAFT eliminates certain scheduling constraints and also removes the need for some of the comparison instructions.

The variations of the SWIFT technique, SWIFT:ST and SWIFT:CF have normalized execution times of 1.270 and 1.251. These techniques have greater performance than full SWIFT and all CRAFT versions because they protect only a subset of the program calculations and thus need only to execute a subset of duplicated instructions. The reduced protection leads directly to increased performance.

The PROFiT algorithms are able to provide better performance than any of the full reliability techniques (which exclude SWIFT:CF and SWIFT:ST). PROFiT maximized the utility function by trading off performance for reliability in a sophisticated way. SWPROFiT outperforms SWIFT (1.217 vs. 1.429), CRPROFiT outperforms all CRAFT versions (1.281 vs. 1.314) and full PROFiT performs just as well as CRPROFiT.

It is notable that in some instances, such as 179.art and 256.bzip, the performance remained the same or even improved after a reliability transformation. We discovered that the baseline version of these benchmarks suffered heavily from load-use stalls. Therefore, inserting instructions between the load and the use will not have any negative impact on performance. In fact, these perturbations in the schedule may even slightly improve performance.

Hardware-only redundancy techniques, such as lockstepping and RMT, typically have a smaller performance penalty than software-only approaches. The RMT hardware technique has been shown to suffer a 32% slowdown when compared to a non-redundant single-threaded core and a 40% slowdown when compared to a non-redundant multi-threaded core. These numbers correspond to normalized execution times of 1.47x and 1.67x respectively. Although the evaluation of the CRAFT techniques were done on an Intel® Itanium® 2 processor and the RMT implementation on a Compaq EV8 processor, we believe a comparison of the results from across the two evaluations is still meaningful and instructive. The normalized execution times for CRAFT are similar to those for RMT, and since we believe that the CRAFT techniques will perform similarly on aggressive out-of-order processor, CRAFT can be expected to yield performance comparable to RMT techniques. More importantly, these hardware-only approaches cannot be easily altered to enable software controllability and thus are unable to fully reap the benefits of tuning reliability to work in concert with the inherent redundancy and sensitivity of the code.

8.3 Reliability

The AVF and MWTF reliability metrics were also measured for the all of the SWIFT, CRAFT, and PROFiT variations. The reliability was evaluated using the fault injection methodology presented in Section 7.2. We first evaluate the AVF of the different systems, and then the MWTF (of which AVF is one component).

8.3.1 *Architectural Vulnerability Factor.* We evaluated the AVF of the integer register file for each of our techniques and benchmarks. We chose to fully investigate the integer register file because its AVF is an order of magnitude greater than that of most other structures. An evaluation of the AVF of other structures and a comparison to the AVF of the integer register has been previously reported by Reis et al. [2005b].

We conducted 5,000 separate injection experiments for each benchmark for each system we evaluated (except SWIFT:CF and SWIFTBR which had 1,000 injections), for a total of 504,000 individual injection runs. We randomly selected a point in time uniformly distributed across the run time of the program. We executed the program, and at that point in time, we selected one random bit from one random physical register and inverted that bit to simulate a transient fault. We then continued executing the program to completion and noted the output and exit status.

We evaluated the reliability of the systems using an Intel® Itanium® 2 as the baseline microarchitecture. The Liberty Simulation Environment's (LSE) simulator builder [Vachharajani et al. 2002; Vachharajani et al. 2004] was used to construct our cycle-accurate performance models. Our baseline machine models the real hardware with extremely high fidelity; its cycles per instruction (CPI) matches that of a real Intel® Itanium® 2 to within 5.4% absolute error [Penry et al. 2005]. For details of the IA64 architecture and the Intel® Itanium® 2 processor implementation, please refer to the Architecture Software Developer's Manual [Intel Corporation 2002].

The Intel® Itanium® 2 integer register file is composed of 128 65-bit registers, 64 bits for the integer data and 1 bit for the NaT (Not a Thing) bit. In the microarchitectural simulation, any of the $127 \times 65 = 8255$ bits of the integer register file could be flipped in any cycle. One register, $r0$, cannot be affected by a transient fault since it always contains the constant value zero, but the other 127 registers are vulnerable.

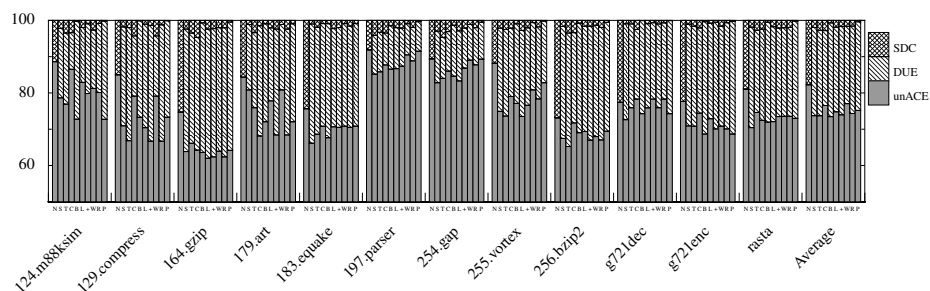After a fault was injected, the microarchitectural simulator monitored the bit for the

Fig. 8. AVF of the Integer Register File. Bars from left to right: No fault tolerance (N), SWIFT (S), SWIFT:ST (T), SWIFT:CF (B), CRAFT:CSB (C), CRAFT:LVQ (L), CRAFT:CSB+LVQ (+), PROFiT:SWIFT (W), PROFiT:CRAFT (R), PROFiT:SWIFT+CRAFT (P)

first use of the corrupted register. Instructions in-flight past the REG stage (register read) either already have their input values or will obtain their input values from the bypass logic. In either case, the register file will not be accessed and so the transient fault will not affect these instructions. The microarchitectural simulation recorded the first instance of an instruction consuming the corrupted register, if one existed. The fault may never propagate to an instruction if the register is written to after the fault but before it is read. If the fault was never propagated, then it had no effect on architecturally correct execution (i.e. was unACE).

If the corrupted register was determined to be consumed, then native Intel$^{®}$ Itanium$^{®}$ 2 execution was continued until program completion. By allowing the program to run to completion, we determined if the corrupted and consumed bit caused a fault in the final output. As previously mentioned in Section 7.2, a fault affecting architectural state does not necessarily affect program correctness. If the native run resulted in correct output, the corrupted bit was considered an unACE bit. If a fault was detected, then the bit was considered DUE. Otherwise, the bit was considered SDC.

If the program ran to completion, but produced the wrong output, the corrupted bit was considered a dSDC bit. Certain corrupted bits caused the program to terminate with a noticeable event. Those bits were considered pSDC because the error can potentially be detected by the noticiable event. We consider segmentation faults, illegal bus exceptions, floating point exceptions, NaT consumption faults, and self-termination due to application-level checking as pSDC events. Programs whose execution time exceeded 100 times the normal execution time were deemed to be in an infinite loop and considered pSDC. Although pSDC could potentially be considered DUE, for brevity's sake we shall bucket pSDC and dSDC together as simply SDC for the remainder of this paper.

Figure 8 shows the AVF for each of the benchmarks and systems evaluated. The stacked bar graph represents percentage of bits in the register file that are the SDC, DUE, and unACE. For each benchmark and each of the ten systems, the distribution of bits is reported. In each of the benchmarks, the reliability techniques greatly reduce the amount of SDC bits (top crosshatch bar) compared with the unprotected baseline. SDC bits represent on average 17.15% of the unprotected case, whereas they represent 2.80% of the bits for SWIFT:CF, and 0.71% for the CRAFT:CSB and PROFiT techniques. Even the reliability technique with the worst SDC percentage reduces that SDC by a factor of 6.
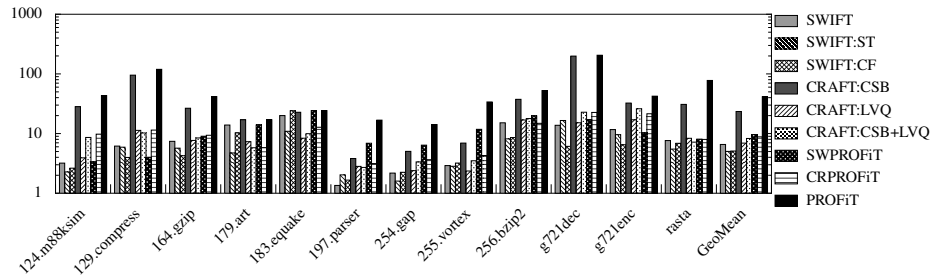
Fig. 9.   Normalized Mean Work To SDC Failure for the Integer Register File

Despite the high reliability of CRAFT:CSB+LVQ, the SDC is still nonzero. This is due to two reasons. The first is that in order to maintain the calling convention, argument passing cannot be duplicated. Before entering a function, argument registers are checked against their duplicate counterparts to ensure their values match. Then, after entering the function, the parameters are copied into duplicate registers for use by the subsequent redundant code. A transient fault on one of the parameters to the function after the check, but before the copy, will lead to that fault going undetected.

The second source of SDC errors arises from the exceptions that terminate the program. For example, if a fault occurs to data that feeds the denominator of a divide instruction, that fault could create a divide-by-zero exception and terminate the program. The techniques do not validate the source operands to divide instructions, only data being written to memory or affecting control flow. Another type of exception that will terminate the program is a segmentation fault. Segmentation faults are particularly onerous in CRAFT:LVQ and CRAFT:LVQ+CSB. Recall from Section 5.2 that the code which uses the load value queue does not check the address before issuing the load instruction, leaving validation to the hardware structure. Since the first load is sent to memory directly, while the second is shunted from the LVQ, a fault on the address of the first load may cause a segmentation fault. This increases the AVF_{SDC} of techniques which incorporate the load value queue.

The reliability of the PROFiT builds are also given in Figure 8. Clearly, both PROFiT and SWIFT builds are much more reliable than NOFT builds, which have an AVF_{SDC} of approximately 15%. The AVF_{SDC} of PROFiT and SWIFT are statistically identical at approximately 1.7%. But observe that the AVF_{DUE} of PROFiT, 25.2%, is significantly smaller than the AVF_{DUE} of SWIFT, 26.6%. Therefore, the number of false DUE detections is reduced and the amount of work that can be accomplished before the raising of a fault-detection exception is increased. Note that both SWIFT:ST and SWIFT:CF have an even lower AVF_{DUE} at the expense of a higher AVF_{SDC}.

8.3.2 *Mean Work To Failure.* By combining the performance and the AVF of our techniques, we can compute the Mean Work To Failure as described in Section 7. Figure 9 shows the normalized Mean Work To Failure for each of the techniques we have analyzed. All of the techniques increase the MWTF by at least 5x, with CRAFT:CSB and PROFiT increasing MWTF by 23.4x and 41.4x respectively.

The increase in MWTF is due to the large decrease in the average number of SDC bits for the integer register file. However, this improvement in MWTF is tempered by the longer execution times that the reliability techniques incur. For example, while the SWIFT:CF
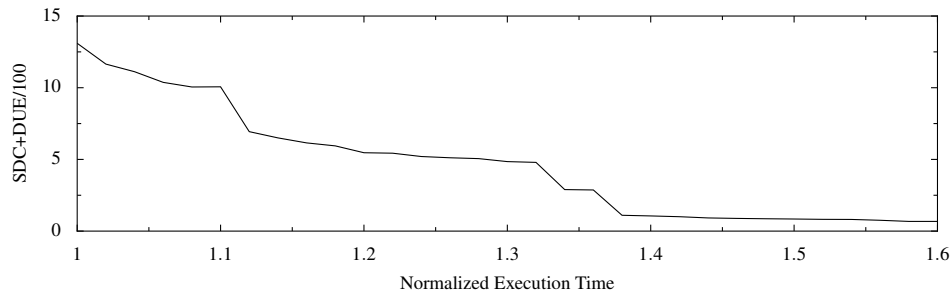
Fig. 10.   Reliability as a function of constrained performance.

technique reduces AVF_{SDC} by a factor of 6x, the MWTF only increases by a factor of 5x.

The CRAFT:CSB technique realizes a 23.4x increase in MWTF because of the large decrease in SDC and only modest increase in execution time. The PROFiT technique has approximately the same AVF_{SDC} as the CRAFT:CSB technique, but the selective application of redundancy is able to decrease the execution time relative to the CRAFT technique alone. This results in a higher MWTF for PROFiT.

8.4   Hardware

Hardware cost is an important consideration for processor architects trying to meet soft-error goals. Software-controlled reliability and performance optimization algorithms like PROFiT incur zero hardware cost. The SWIFT technique, which is a software-only technique, incurs no hardware cost. The CRAFT techniques, which build on SWIFT, require the addition of simple low-cost hardware structures, namely the checking store buffer (CSB) and the load value queue (LVQ), which were inspired by RMT implementations. Overall, these techniques incur very low hardware costs compared to expensive implementations like RMT, while achieving comparable levels of reliability and performance.

8.5   PROFiT with Strict Performance Budget

Often, designers will be faced with a fixed performance budget. In such cases, no matter what the reliability cost, the performance degradation may not exceed a fixed amount. PROFiT is applicable in such cases, by simply assigning a utility of $-\infty$ to configurations which exceed the performance budget.

Figure 10 shows the average weighted AVF across all benchmarks for various levels of performance constraint. It is notable PROFiT is able to greatly reduce the SDC with less than 15% performance degradation. Also note that there are inflection points around 1.10, 1.30, and 1.40 where the slope of the curvature changes. This implies that the set of functions which are both more reliable and within the performance budget does not increase significantly between 1.40 and 1.60. This is as expected, since all of our protection schemes can be implemented in such a budget, thus making all permutations available for selection. Between 1.10 and 1.30, the slope of the line is also less steep than at other points in the graph, indicating a paucity of functions in this region of the reliability/performance space.

This phenomenon is further elucidated upon in Figure 11, which shows the cumulative
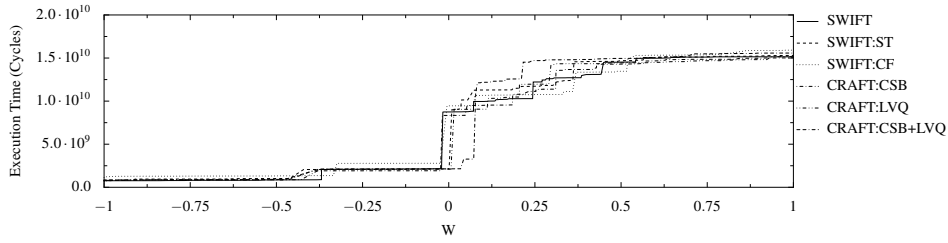
Fig. 11.   CDF of functions vs. W weighted by execution profile.

distribution, weighted by execution profile, of functions which satisfy

$$\frac{t_{\text{protected}}}{(\mathbf{MWTF}_{\text{protected}})^W} < \frac{t_{\text{original}}}{(\mathbf{MWTF}_{\text{original}})^W}$$

for various values of W. This represents the weight given to reliability versus performance for each function when regarded in isolation.

The most significant behavior occurs at $W = 0$. In this neighborhood, MWTF is weighted so as to be relatively insignificant, putting more emphasis on performance. After this initial rise, the curve begins to level off, becoming nearly flat by the time $W = 1$. At this point, MWTF and execution time are inversely proportional: a 10% increase in execution time is met with a 10% increase in MWTF.

This graph implies that nearly all functions of any significance cross their *tipping point*, the point at which a transition to a fault-tolerant version becomes desirable, in the positive neighborhood around zero. That is to say that MWTF can be vastly improved with little impact on performance. This can be deduced from the MWTF and performance numbers presented earlier; execution time increases of 0.5x lead to orders of magnitude improvement in MWTF. One should only very rarely need to pay a linear performance penalty for reliability.

8.6   Case study: 124.m88ksim

We examine some of these phenomena within the context of specific example, namely, 124.m88ksim. 124.m88ksim is especially instructive because it clearly demonstrates some of the phenomena which the PROFiT technique attempts to exploit. Figure 12 shows the top three functions which contribute to execution time and AVF for this benchmark. Each of these three functions has distinctly different behavior.

The lower-right function, `loadmem`, is by far the least reliable without any fault protection. However, when fault protection is added, the reliability increases drastically, while the performance is nearly unaffected. On the graph, this type of change is represented by the line with a steep slope. This corresponds to a function in the $W = 0$ region of Figure 11. This function performs many irregular loads, and so much of the time is dominated by load-use stalls. When a protection technique is added, it can perform the redundant computation in the slots that would otherwise be empty and thus not significantly degrade the overall execution time.

The center function, `Data_path`, follows a very different trend than `loadmem`. Whereas `loadmem`'s versions are positioned vertically above the baseline, `Data_path`'s versions are positioned along a diagonal axis above and to the right, implying a larger decrease in performance for added reliability. This smaller slope shows that the reliability improve-
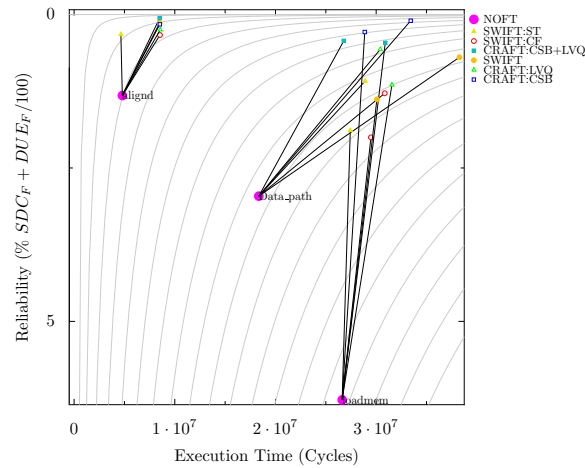
Fig. 12. A scatter plot of SDC vs. execution time in cycles for selected functions of the benchmark 124.m88ksim. The large, labeled circles represent versions of the function without any fault-detection transformation applied to them. The different versions of that function are joined to it by lines. The light-grey contour lines delineate the level curves of our utility function.

ment per unit of performance degradation is much smaller than with loadmem. In fact, the slope of the rays indicate that this function can be more closely binned in the $W = 1$ point in Figure 11. Data_path has a mix of irregular data accesses, irregular control flow, function calls, and logical computation.

The last function, alignd, mostly resembles Data_path, except for one aberrant point directly above the unprotected function which correspondes to SWIFT without control-flow checking. Observe that since all other protected versions of the function have control-flow checking, the performance impact of control-flow is the dominating factor. On the other hand, reliability is not dominated by control-flow, since all versions have approximately the same reliability. alignd consists mainly of a short, counted, hot loop containing a dependent chain of logical operations. Since operations within the loop are dependent, the code cannot execute faster than 1 IPC. Thus, in the six-wide Intel® Itanium® 2, there are many empty slots, which the redundant computation can exploit. Therefore, the performance penalty of duplicating the backwards slice of the store instructions is much smaller than that of checking control-flow, which is frequent and regular.

9. CONCLUSION

In this paper, we demonstrated that SWIFT, a software-only fault-detection technique, and CRAFT, a set of hybrid fault-detection systems, are able to vastly reduce the number of undetected faults with very little or no hardware overhead. In order to properly measure the reliability of SWIFT and CRAFT, we introduced the notion of Mean Work To Failure, a metric which is able to faithfully portray the reliability for software-only and hybrid techniques. We also presented a fault injection methodology which produces highly accurate results with fast simulation time.

Using these techniques, we determined that SWIFT reduced the number of output corrupting faults by 89.0% with a 42.9% performance degradation. Meanwhile, CRAFT was

able to further reduce execution time by 11.5% and decrease the number of output corrupting faults by 63.6% over SWIFT alone.

More importantly, SWIFT and CRAFT enabled the software-controlled fault tolerance introduced in this paper. We proposed an implementation of software-controlled fault tolerance, called PROFiT, a profile-guided compiler technique for determining which portions of code are most vulnerable and adjusting the protection accordingly. PROFiT was able to reduce the execution time by 21.2% and 3.3% for the software and hybrid techniques while having no deleterious impact on reliability. PROFiT demonstrates that software-controlled fault tolerance offers designers more flexibility in their application of fault-detection techniques and encourages further forays into software-controlled fault tolerance.

Acknowledgments

REFERENCES

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

BAUMANN, R. C. 2001. Soft errors in advanced semiconductor devices-part I: the three radiation sources. *IEEE Transactions on Device and Materials Reliability 1,* 1 (March), 17–22.

BOLCHINI, C. AND SALICE, F. 2001. A software methodology for detecting hardware faults in vliw data paths. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*.

BOSSEN, D. C. 2002. CMOS soft errors and server design. In *IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals*. 121_07.1 – 121_07.6.

CZECK, E. W. AND SIEWIOREK, D. 1990. Effects of transient gate-level faults on program behavior. In *Proceedings of the 1990 International Symposium on Fault-Tolerant Computing*. 236–243.

DEAN, A. G. AND SHEN, J. P. 1998. Techniques for software thread integration in real-time embedded systems. In *Proceedings of the IEEE Real-Time Systems Symposium*. IEEE Computer Society, Washington, DC, USA, 322.

GOMAA, M., SCARBROUGH, C., VIJAYKUMAR, T. N., AND POMERANZ, I. 2003. Transient-fault recovery for chip multiprocessors. In *Proceedings of the 30th annual international symposium on Computer architecture*. ACM Press, 98–109.

HOLM, J. G. AND BANERJEE, P. 1992. Low cost concurrent error detection in a VLIW architecture using replicated instructions. In *Proceedings of the 1992 International Conference on Parallel Processing*. Vol. 1. 192–195.

HORST, R. W., HARRIS, R. L., AND JARDINE, R. L. 1990. Multiple instruction issue in the NonStop Cyclone processor. In *Proceedings of the 17th International Symposium on Computer Architecture*. 216–226.

INTEL CORPORATION. 2002. *Intel Itanium Architecture Software Developer's Manual, Volumes 1-3*. Santa Clara, CA.

KIM, S. AND SOMANI, A. K. 2002. Soft error sensitivity characterization for microprocessor dependability enhancement strategy. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*. 416–425.

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. 1997. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*. 330–335.

MAHMOOD, A. AND MCCLUSKEY, E. J. 1988. Concurrent error detection using watchdog processors-a survey. *IEEE Transactions on Computers 37,* 2, 160–174.

MUKHERJEE, S. S., KONTZ, M., AND REINHARDT, S. K. 2002. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the 29th annual international symposium on Computer architecture*. IEEE Computer Society, 99–110.

MUKHERJEE, S. S., WEAVER, C., EMER, J., REINHARDT, S. K., AND AUSTIN, T. 2003. 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*. IEEE Computer Society, 29.

O'GORMAN, T. J., ROSS, J. M., TABER, A. H., ZIEGLER, J. F., MUHLFELD, H. P., MONTROSE, I. C. J., CURTIS, H. W., AND WALSH, J. L. 1996. Field testing for cosmic ray soft errors in semiconductor memories. In *IBM Journal of Research and Development*. 41–49.

OH, N. AND MCCLUSKEY, E. J. 2001. Low energy error detection technique using procedure call duplication. In *Proceedings of the 2001 International Symposium on Dependable Systems and Networks*.

OH, N., SHIRVANI, P. P., AND MCCLUSKEY, E. J. 2002a. Control-flow checking by software signatures. In *IEEE Transactions on Reliability*. Vol. 51. 111–122.

OH, N., SHIRVANI, P. P., AND MCCLUSKEY, E. J. 2002b. ED^{4}I: Error detection by diverse data and duplicated instructions. In *IEEE Transactions on Computers*. Vol. 51. 180 – 199.

OH, N., SHIRVANI, P. P., AND MCCLUSKEY, E. J. 2002c. Error detection by duplicated instructions in superscalar processors. In *IEEE Transactions on Reliability*. Vol. 51. 63–75.

OHLSSON, J. AND RIMEN, M. 1995. Implicit signature checking. In *International Conference on Fault-Tolerant Computing*.

PATEL, J. H. AND FUNG, L. Y. 1982. Concurrent error detection in alu's by recomputing with shifted operands. *IEEE Transactions on Computers 31,* 7, 589–595.

PENRY, D. A., VACHHARAJANI, M., AND AUGUST, D. I. 2005. Rapid development of a flexible validated processor model. In *Proceedings of the 2005 Workshop on Modeling, Benchmarking, and Simulation (MOBS)*.

RAY, J., HOE, J. C., AND FALSAFI, B. 2001. Dual use of superscalar datapath for transient-fault detection and recovery. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 214–224.

REBAUDENGO, M., REORDA, M. S., VIOLANTE, M., AND TORCHIANO, M. 2001. A source-to-source compiler for generating dependable software. In *IEEE International Workshop on Source Code Analysis and Manipulation*. 33–42.

REINHARDT, S. K. AND MUKHERJEE, S. S. 2000. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th annual international symposium on Computer architecture*. ACM Press, 25–36.

REIS, G. A., CHANG, J., VACHHARAJANI, N., RANGAN, R., AND AUGUST, D. I. 2005a. SWIFT: Software implemented fault tolerance. In *Proceedings of the 3rd International Symposium on Code Generation and Optimization*.

REIS, G. A., CHANG, J., VACHHARAJANI, N., RANGAN, R., AUGUST, D. I., AND MUKHERJEE, S. S. 2005b. Design and evaluation of hybrid fault-detection systems. In *Proceedings of the 32th Annual International Symposium on Computer Architecture*. 148–159.

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

SAXENA, N. AND MCCLUSKEY, E. J. 1998. Dependable adaptive computing systems – the ROAR project. In *International Conference on Systems, Man, and Cybernetics*. 2172–2177.

SCHUETTE, M. A. AND SHEN, J. P. 1994. Exploiting instruction-level parallelism for integrated control-flow monitoring. In *IEEE Transactions on Computers*. Vol. 43. 129–133.

SHIRVANI, P. P., SAXENA, N., AND MCCLUSKEY, E. J. 2000. Software-implemented EDAC protection against SEUs. In *IEEE Transactions on Reliability*. Vol. 49. 273–284.

SHIVAKUMAR, P., KISTLER, M., KECKLER, S. W., BURGER, D., AND ALVISI, L. 2002. 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*. 389–399.

SLEGEL, T. J., AVERILL III, R. M., CHECK, M. A., GIAMEI, B. C., KRUMM, B. W., KRYGOWSKI, C. A., LI, W. H., LIPTAY, J. S., MACDOUGALL, J. D., MCPHERSON, T. J., NAVARRO, J. A., SCHWARZ, E. M., SHUM, K., AND WEBB, C. F. 1999. IBM's S/390 G5 Microprocessor design. In *IEEE Micro*. Vol. 19. 12–23.

VACHHARAJANI, M., VACHHARAJANI, N., AND AUGUST, D. I. 2004. The Liberty Structural Specification Language: A high-level modeling language for component reuse. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI)*. 195–206.

VACHHARAJANI, M., VACHHARAJANI, N., PENRY, D. A., BLOME, J. A., AND AUGUST, D. I. 2002. Microarchitectural exploration with Liberty. In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO)*. 271–282.

VENKATASUBRAMANIAN, R., HAYES, J. P., AND MURRAY, B. T. 2003. Low-cost on-line fault detection using control flow assertions. In *Proceedings of the 9th IEEE International On-Line Testing Symposium*. 137–143.

VIJAYKUMAR, T. N., POMERANZ, I., AND CHENG, K. 2002. Transient-fault recovery using simultaneous multithreading. In *Proceedings of the 29th annual international symposium on Computer architecture*. IEEE Computer Society, 87–98.

WANG, N., FERTIG, M., AND PATEL, S. J. 2003. 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*. 56–67.

WANG, N. J., QUEK, J., RAFACZ, T. M., AND PATEL, S. J. 2004. Characterizing the effects of transient faults on a high-performance processor pipeline. In *Proceedings of the 2004 International Conference on Dependendable Systems and Networks*. 61–72.

WEAVER, C., EMER, J., MUKHERJEE, S. S., AND REINHARDT, S. K. 2004. Techniques to reduce the soft error rate of a high-performance microprocessor. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*.

YEH, Y. 1996. Triple-triple redundant 777 primary flight computer. In *Proceedings of the 1996 IEEE Aerospace Applications Conference*. Vol. 1. 293–307.

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