Design and Evaluation of Hybrid Fault-Detection Systems

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

Departments of Electrical Engineering and Computer Science
Princeton University
Princeton, NJ 08540
{gareis,jcone,nvachhar,ram,august}@princeton.edu

FACT Group
Intel Massachusetts
Hudson, MA 01749
shubu.mukherjee@intel.com

Abstract

As chip densities and clock rates increase, processors are becoming more susceptible to transient faults that can affect program correctness. Up to now, system designers have primarily considered hardware-only and software-only fault-detection mechanisms to identify and mitigate the deleterious effects of transient faults. These two fault-detection systems, however, are extremes in the design space, representing sharp trade-offs between hardware cost, reliability, and performance.

In this paper, we identify hybrid hardware/software fault-detection mechanisms as promising alternatives to hardware-only and software-only systems. These hybrid systems offer designers more options to fit their reliability needs within their hardware and performance budgets. We propose and evaluate CRAFT, a suite of three such hybrid techniques, to illustrate the potential of the hybrid approach. For fair, quantitative comparisons among hardware, software, and hybrid systems, we introduce a new metric, Mean Work To Failure*, which is able to compare systems for which machine instructions do not represent a constant unit of work. Additionally, we present a new simulation framework which rapidly assesses reliability and does not depend on manual identification of failure modes. Our evaluation illustrates that CRAFT, and hybrid techniques in general, offer attractive options in the fault-detection design space.*

1 Introduction

In recent decades, microprocessor performance has been increasing exponentially. A large fraction of this performance gain is directly due 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 [1, 11], rendering processors that use them more susceptible to *transient faults*. Transient faults, also known as *soft errors*, are intermittent faults caused by external events, such as energetic particles striking the chip, that do not cause permanent damage, but may result in incorrect program execution by altering signal transfers or stored values.

To detect or recover from these faults, designers typically introduce redundant hardware. For example, storage structures such as caches and memory often include error correcting codes (ECC) or parity bits. However, adding such fine-grained protection to all hardware structures can be prohibitively expensive. Other techniques, which we classify *macro-reliability techniques*, duplicate coarse-grained structures such as functional units, processor cores [6, 21, 29], or hardware contexts [5, 9, 18, 25] in order to provide transient fault tolerance more cost-effectively. The IBM z900 [21] and the Boeing 777 [29] all implement some form of macro-reliability.

To provide protection when even macro-reliability hardware costs are prohibitive, software-only approaches have been proposed as alternatives [12, 13, 19, 20, 24]. While software-only systems are cheaper to deploy, they cannot achieve the same performance or reliability as hardware-based techniques, since they have to execute additional instructions and are unable to examine microarchitectural state. Despite these limitations, software-only techniques have shown promise, in the sense that they can significantly improve reliability with reasonable performance overhead [12, 13, 19].

Despite the promise of existing methods, the fault-tolerance design space remains sparse. System designers must choose between two extremes when trading off between reliability, performance, and hardware cost. For systems where software-only reliability is inadequate but hardware-only techniques are too costly, designers are left with no solution. *Hybrid techniques*, which combine software *and* hardware aspects, may best be able to satisfy the performance, reliability, and hardware cost requirements of a design. Recognizing this, we present a suite of three hybrid fault-detection implementations called CRAFT, for CompileR Assisted Fault Tolerance. The CRAFT suite is based on the software-only SWIFT [19] technique augmented with structures borrowed from the hardware-only Redundant Multithreading (RMT) technique [9, 18]. These three additional points in the design space provide a smoother transition from software-only to hardware-only techniques; they provide better reliability and performance than software-only techniques, and have lower hardware costs than hardware-only techniques.

Proceedings of the 32nd International Symposium on Computer Architecture (ISCA'05)
1063-6897/05 $20.00 © 2005 IEEE

For fair, quantitative comparisons between the CRAFT suite and other points in the design space, we introduce a new methodology for evaluating fault-tolerance systems. The methodology adds support for software-only and hybrid schemes to existing support for hardware techniques. The methodology includes a new reliability metric called *Mean Work To Failure* (MWTF) that captures the trade-off between reliability and performance. Additionally, the methodology includes a new reliability simulation framework that provides accurate reliability estimates for software-only and hybrid systems, as well as hardware-only systems, by using fault injection and programs run to completion. Unlike ACE analysis [10], our methodology can produce meaningful results without having to identify all corner cases of a software-only or hybrid system. At the same time, through clever combination of microarchitectural and architectural fault simulations, our methodology avoids excessive runtimes associated with fault injection.

We use this new methodology to characterize the reliability of various points in the fault-tolerance design space through analysis of three different hardware structures of an Intel® Itanium® 2 processor: the integer register file, the predicate register file, and the instruction fetch buffer.

The CRAFT hybrid techniques reduce overall execution time by 5% and output-corrupting faults by 75% over software-only fault-detection techniques. Undetected output-corrupting faults are reduced by 98% compared to the no-fault-detection case, causing errors on only 0.35% of bit flips for the structures measured. This reliability is comparable to that of hardware-only systems, but is achieved using significantly less hardware.

In identifying hybrid fault-detection systems as interesting design points, this paper makes four contributions:

1. We propose CRAFT, a suite of three new hybrid fault-detection systems.
2. We propose MWTF, a new metric for evaluating the trade-off between reliability and performance across systems where instructions do not represent a constant unit of work.
3. We describe a new reliability evaluation framework that quickly and accurately estimates reliability based on simulation at the microarchitectural level and complete program execution at the architectural level.
4. Using the proposed evaluation framework, we demonstrate that hybrid hardware/software systems offer additional design options for achieving the desirable reliability, performance, and hardware cost trade-off.

The rest of the paper is organized as follows. Section 2 provides background information for transient fault detection. Sections 3 and 4 describe existing hardware and software fault-detection systems. Section 5 motivates the hybrid approach to fault detection through a detailed description of the three implemented CRAFT techniques. Section 6 describes the new framework for evaluating reliability and introduces the MWTF metric. Section 7 evaluates the hybrid fault-detection design space. The paper concludes in Section 8.

2 Preliminaries

Throughout this paper, we will assume a *Single Event Upset* fault model. That is, we will assume that at most one bit can be flipped at most 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 [10]:

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 output. For example, unACE bits occur in state elements that hold dynamically dead information, logically masked values, or control flows that are Y-branches [26]

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 results in 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 *definite SDC* (dSDC), faults that silently corrupt program output, and *potential SDC* (pSDC), faults that cause abnormal program termination. 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.

In this paper, we will sometimes refer to a bit as 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*, the sum of SDC and DUE, than the original system. There are two principal reasons for this. First, most practical fault-detection schemes may detect an actual fault, although it would not have affected the output. These faults, known as *false DUE*, 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. A more detailed discussion can be found in [10, 28]. 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 soft error rate, they reduce SDC faults, which are more deleterious than DUE. Consequently, system designers typically tolerate higher incidents of DUE in order to reduce SDC. A typical target SDC rate is one fault per 1000 years, while a typical target DUE rate is 100 times greater, one fault per 10 years [3].

To measure a microarchitectural structure's susceptibility to transient faults, the notion of an *architectural vulnerability factor* (AVF) is often used. It 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 SDC or DUE 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 recovery, as shown by the recovery techniques [5, 25] that have been derived from detection-only techniques [9, 18].

3 Hardware-Only Redundancy Techniques

Error correcting codes (ECC) and parity bits have been widely used to protect various hardware structures at a very fine granularity. However, it is tedious and impractical to protect all pieces of hardware logic with ECC or parity bits. In order to protect the entire processor core, researchers have come up with various *macro-redundancy* mechanisms, which introduce coarse-grain redundancy across the entire system.

Over the years, techniques have been proposed to provide redundant execution in hardware that converts almost all SDC events to DUE events. Lockstepping, used in the Compaq Non-Stop Himalaya processor [6], performs the same computation on two procesosrs and compares the results on every cycle. While this provides complete fault coverage under the SEU model, it incurs a substantial hardware cost due to the duplicated cores and performance overhead due to the routing of signals to a central checker unit. Other commercial fault-tolerant systems, such as the IBM S/390 [21] and Boeing 777 airplanes [29] also replicate the processor in part or in whole and use hardware checkers to validate the redundant computations.

```
                             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 L1', r1' != r5'
                             5:  xor RTS = sig0,sig1
                             6:  L1':
br L1, r1 == r5                  br L1, r1 == r5
...                              ...
L1:                              L1:
                             7:  xor pc' = pc',RTS
                             8:  br faultDet, pc' != sig1
                             9:  br faultDet, r1 != r1'
                             10: br faultDet, r2 != r2'
st m[r1] = r2                    st m[r1] = r2
```

(a) Original Code (b) SWIFT Code

Figure 1. Software-only fault-detection

Reinhardt and Mukherjee [18] proposed simultaneous Redundant MultiThreading (RMT), which takes advantage of the multiple hardware contexts of SMT, but makes better use of system resources through loosely synchronized redundant threads, and reduces the overhead of validation by eliminating cache misses and misspeculations in the trailing thread. Ray et al. [16] proposed modifying an OoO superscalar processor's microarchitectural components such as the reorder buffer and register renaming hardware to implement redundancy. They also added an *Instruction Replicator* to duplicate instructions and validate their results before committing. Mukherjee et al. [9] proposed Chip-level Redundant Threading (CRT) that applied RMT-like techniques to a chip-multiprocessor setup.

Table 1 gives a summary of the protection level and hardware cost of the aforementioned schemes. These redundancy mechanisms are characterized by almost-perfect fault coverage, low performance degradation, and high hardware costs, thus occupying the high-end extreme in the spectrum of fault-detection mechanisms.

4 Software-Only Redundancy Techniques

Software techniques are attractive from both the chip designers' and the consumers' perspectives because they can provide very high levels of fault coverage with small performance cost and *zero* hardware cost. Additionally, software-only techniques provide users and programmers with the ability to turn redundancy on and off in the generated code, thus allowing fine-grained trade-offs between performance and fault coverage. This makes software-only techniques a vital point in the design space.

Several software-only fault-detection methods have been proposed, such as CFCSS [12] and ACFC [24], that protect execution control flow. At the high-level, Rebaudengo et al. [17] proposed a source-to-source pre-pass compiler that generates fault-detection code in the source

				Instructions				
Technique	Category	Opcode	Control	Loads	Stores	Other	Memory	Hardware Requirements
Lockstepping [6]	HW	all	all	all	all	all	none	Dual core, checking logic
RMT [18]	HW	all	all	all	all	all	none	SMT base machine, Branch Outcome Queue, Checking Store Buffer, Load Value Queue
CRT [9]	HW	all	all	all	all	all	none	CMP base machine, Branch Outcome Queue, Checking Store Buffer, Load Value Queue, intercore communication datapath
Superscalar [16]	HW	most[a]	most[a]	most[a]	most[a]	most[a]	none	Superscalar machine, Instruction replicator, checking logic
CFCSS [12]	SW	some[b]	most[c]	none	none	none	none	None
ACFC [24]	SW	some[b]	most[c]	none	none	none	none	None
EDDI [13]	SW	most[d]	most[c,e]	all	all	all	all	None
SWIFT [19]	SW	most[f]	most[c]	most[e]	most[e]	all	none	None
CRAFT:CSB	Hybrid	all	most[c]	most[e]	all	all	none	Checking Store Buffer
CRAFT:LVQ	Hybrid	most[f]	most[c]	all	most[e]	all	none	Load Value Queue
CRAFT:CSB+LVQ	Hybrid	all	most[c]	all	all	all	none	Checking Store Buffer, Load Value Queue

[a] faults to the instruction replicator and register file 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 fault coverage and hardware requirements for fault-detection approaches

language. Oh et al. [13] proposed EDDI, a low-level detection technique, which duplicates all instructions except branches and inserts validation code before all stores and control transfers thus ensuring the correctness of values to be written to memory. Reis et al. [19] proposed SWIFT, which improves on EDDI performance and reliability through better control-flow validation and other optimizations. By taking advantage of modern memory systems' built-in protection, such as ECC, SWIFT avoids duplicating store instructions, thus resulting in superior fault coverage and performance.

SWIFT provides fault detection by inserting redundancy directly into compiled code, without making any assumptions about the underlying hardware. SWIFT duplicates the original program's instructions, schedules the original and duplicated instruction sequences together in the same thread of control and inserts explicit checking instructions at strategic points to compare values flowing through the original and duplicated code sequences. Because of the slack in the original program's static code schedule, it is usually possible to interleave redundant and original operations so that the performance degradation is limited.

Figure 1 shows a sample code sequence before and after the SWIFT fault-detection transformation. In the SWIFT code shown in Figure 1(b), the additional instructions added to provide redundancy have been annotated with instruction numbers. 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. Instruction 3 is the redundant version of the `add` instruction in the original code. Instructions 9 and 10 verify that the source operands of the store instruction match their duplicate versions.

The control flow checking is handled by instructions 4 to 8. Instruction 4 skips the computation of the *RunTime Signature* (RTS) if the duplicate registers indicate that the branch in the original code will not be taken. The RTS is computed in instruction 5 by XORing the signature of the current block (sig0) and the signature of the destination block (sig1). Prior to `L1`, `pc'` should contain sig0, the signature of that block. After entering `L1`, `pc'` is updated to reflect the new current block (sig1) by instruction 7. Instruction 8 verifies that `pc'` contains the signature of the current block.

Although they provide high reliability benefit for zero hardware cost, their inability to directly examine microarchitectural components prevents software-only techniques from offering protection against a variety of faults. Also, since redundancy is introduced solely via instructions, there can be a delay between validation and the use of validated values and any faults during this gap may lead to SDC. The probability of such events can be reduced by scheduling the validation code as close to branches and stores as possible, but it cannot be eliminated. Moreover, faults in an instruction's opcode bits can convert a non-store instruction to a store instruction and vice versa, resulting in corrupted memory state.

Performance degradation is another unavoidable consequence of software-only fault-detection techniques. While most duplicated instructions can be scheduled compactly, resource limitations and validation instructions will invariably increase static code schedule height.

Overall, software redundancy mechanisms are characterized by high fault coverage, modest performance degradation, and zero hardware cost. Table 1 gives a summary of the protection level of software reliability systems in comparison with hardware systems.

5 Hybrid Redundancy Techniques

Hybrid software/hardware designs provide an alternative to software-only or hardware-only fault detection tech-

Inst. type	NOFT	SWIFT	CRAFT:CSB	CRAFT:LVQ	CRAFT:CSB+LVQ
STORE	st [r1] = r2	br faultDet, r1 != r1' br faultDet, r2 != r2' st [r1] = r2	st$_1$ [r1] = r2 st$_2$ [r1'] = r2'	*Same as SWIFT*	*Same as CRAFT:CSB*
LOAD	ld r1 = [r2]	br faultDet, r2 != r2' ld r1 = [r2] mov r1' = r1	*Same as SWIFT*	ld$_1$ r1 = [r2] ld$_2$ r1' = [r2']	*Same as CRAFT:LVQ*

Table 2. Comparison of instruction stream for CRAFT techniques

niques. Hybrid techniques for protecting main memory [8] and validating control flow [2, 8, 14] have been proposed, but hybrid designs for whole-processor reliability have not yet been evaluated by the community. A thorough exploration of the design space of redundancy mechanisms is necessary to determine the set of techniques, hardware-only, software-only, or hybrid, that best meet each set of design goals.

In this section, we present three hybrid redundancy techniques. We call these techniques *CompileR-Assisted Fault Tolerance* (CRAFT). CRAFT combines the best known software-only technique, SWIFT, with minimal hardware adaptations from RMT to create systems with nearly perfect reliability, low performance degradation, and low hardware cost. Table 1 shows how these systems provide fine-grain trade-offs between the desired level of reliability, performance, and hardware.

5.1 CRAFT: Checking Store Buffer (CSB)

As noted in Section 4, SWIFT is vulnerable to faults in the time interval between the validation and the use of a register value. If the register value is used in a store instruction, faults during these vulnerable periods can lead to silent data corruption. Similarly, faults in instruction opcode bits can transform non-stores to stores, also resulting in SDC.

In order to protect data going to memory, the CRAFT:CSB compiler duplicates store instructions in the same way that it duplicates all other instructions (refer to Table 2), except that store instructions are also tagged with a single-bit version name, indicating whether a store is an original or a duplicate. Also, the compiler schedules stores so that the duplicate stores happen in the same dynamic order as the original stores. Code thus modified is then run on hardware with an augmented store buffer called the *Checking Store Buffer* (CSB). The CSB functions as a normal store buffer, except that it does not commit entries to memory until they are validated. An entry becomes *validated* once both 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 are equal.

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 signalling, 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.

These modest hardware additions allow the system to detect faults in store addresses and data, as well as dangerous opcode bit-flips. Note that, though this technique duplicates all stores, no extra memory traffic is created, since only one store of each pair leaves the CSB. Unlike the original SWIFT code, CRAFT:CSB code no longer needs validation codes before store instructions (Table 2). This reduces dependence height, improving performance. Further, CRAFT:CSB code also exhibits greater scheduling flexibility. This is because the stores from the two different versions and the instructions they depend on can now be scheduled independently (subject to the constraints mentioned earlier), 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.

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. As shown in Table 2, the load address is validated, data is loaded from memory, and the loaded value is copied into a duplicate register. This code sequence opens two windows of vulnerability. First, there is a window of vulnerability between the address validation and the address consumption by the load instruction. Second, there is a window of vulnerability between the load instruction and the value duplication. A fault which affects the load address in the first window or the loaded value in the second window can lead to data corruption. While the code in the table shows these instructions scheduled in succession, in general, an arbitrary number of instructions can be between each pair of instructions potentially making the vulnerability windows large.

Additionally, as in the case of store instructions, load instructions force the compiler to schedule dependence chains from the original and duplicate versions before the validation code for the load instruction. Just like CRAFT:CSB, CRAFT:LVQ handles these issues for loads by duplicating load instructions (refer to Table 2).

Unfortunately, merely duplicating load instructions will not provide us with correct redundant execution in practice. In multi-programmed environments, intervening writes by another process to the same 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 hardware structure called the *Load Value Queue* (LVQ) to achieve 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 and stop program execution. Loads from different versions may be scheduled independently, but they must maintain the same relative ordering across the two versions. For out-of-order architectures, the hardware must ensure that loads and their duplicates both access the same entry in the load value queue. Techniques presented in the literature can be adapted to accomplish this [9, 18].

The duplicated loads and the LVQ provide completely redundant load instruction execution. The LVQ allows the compiler more freedom in scheduling instructions from the original and duplicate versions around loads, just like the store buffer relaxed the constraint of the original and duplicate code slices around stores. Since the duplicate load instruction will always be served from the LVQ, it will never cause a cache miss or additional memory bus traffic.

5.3 CRAFT: CSB + LVQ

The third and final CRAFT technique duplicates both store and load instructions (refer to Table 2) and adds both the checking store buffer and the load value queue enhancements simultaneously to a software-only fault detection system such as SWIFT.

6 A Methodology for Measuring Reliability

Mean Time To Failure (MTTF) and AVF are two commonly used reliability metrics. However, MTTF and AVF are not appropriate in all cases. In this section, we introduce a new metric that generalizes MTTF to make it applicable in a wider class of fault-detection systems, and introduce a framework to accurately and rapidly measure it.

6.1 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, consider a 1-deep instruction fetch buffer on a single-issue machine. Suppose that we modify this buffer so that it automatically inserts a NOP after every instruction fetched. NOPs do not contain many ACE bits. For example, on an IA-64 architecture, 36 out of 41 bits in a `nop.i` instruction are unACE. Therefore, this insertion of NOPs has the effect of almost halving the buffer's AVF. Since MTTF $= \frac{1}{\text{raw error rate} \times \text{AVF}}$ [10], this NOP insertion will nearly double the buffer's MTTF. While it is certainly true that in the modified buffer failures will be about twice as far apart, few people would consider this increased fault tolerance.

To account for the trade-off between performance and reliability, Weaver et al. introduced the alternative *Mean Instructions To Failure* (MITF) metric [28]. 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 example presented earlier. If introducing the NOPs had no effect on IPC, then the reduction in AVF would correspond to a doubling of MITF.

To adequately capture the trade-off between performance and reliability for hardware *and* software fault-tolerance techniques, we introduce a generalized metric called *Mean Work To Failure* (MWTF).

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

The raw error rate and AVF are the same terms used in MTTF, and the execution time term is the time to execute the given unit of work. A unit of work is an abstract 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, but for software and hybrid techniques this is no longer the case. 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 trade-off between reliability and performance. Such a metric is crucial when comparing hardware, software, and hybrid systems.

6.2 A Framework For Measuring MWTF

To compute the MWTF, one must have an estimate for the 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 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 fault for a given structure [10]. 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 store may cause an SDC outcome, but if it changes a 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. The resulting AVF$_{SDC}$ 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 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 [4, 7, 27]. 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 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 because the program is not executed to completion.

To avoid these shortcomings, we propose using fault injection *and* running 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 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 in 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.

Note that for our structures we do not inject any faults during the simulation phase. We merely record what *would* happen at the architectural level should a fault be injected. Since we never change the simulator's state, we need only run the detailed simulation once. Not changing the simulator's state may yield approximations, however, since faults may change the timing of subsequent instructions. However, for many structures within a system, this kind of behavior is either minimal or nonexistent.

This methodology could be used in a manner that removed those approximations by checkpointing the detailed simulation before a fault injection, inserting the fault, then continuing the detailed simulation with the new value only until all effects are manifested in architectural state. The architectural deviations from the original program are recorded for the second phase, and the detailed simulation is reloaded from the checkpoint and continued with the uncorrupted value until the next fault injection. This captures the changes that may affect timing and only requires limited excess simulation time.

After one microarchitectural run, we can compile all the points we will need during the second phase, and the cost of detailed simulation can be amortized over the many archi-

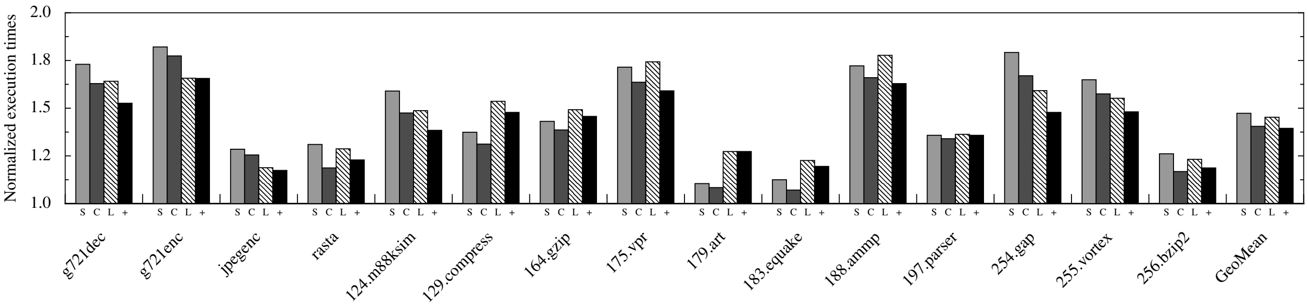


Figure 2. Normalized execution times by system and benchmark. Systems: SWIFT (S), CRAFT:CSB (C), CRAFT:LVQ (L), CRAFT:CSB+LVQ (+)

tectural fault injection simulations. The second phase can use a functional simulator, or an instrumented binary on native hardware, because only architectural state is affected.

We are able to capture the microarchitectural to architectural mapping for the three structures simultaneously. For each benchmark and system we collected approximately 5,000 points via detailed simulation. Out of those 5,000, on average only 57.84% needed to be executed in the architectural model, since faults that alter microarchitectural state that is guaranteed to never affect architectural state are unACE and can be omitted from the architectural fault simulations. The architectural fault simulations takes roughly 3.10x longer than executions on native hardware, and the reduction in necessary simulations creates an effective instrumented execution time of 1.95x.

7 Evaluation

This section evaluates the hybrid CRAFT techniques and does a quantitative and qualitative comparison of the performance, reliability, and hardware cost with existing software-only and hardware-only techniques.

7.1 Performance

In order to evaluate the performance of the CRAFT techniques, we generated redundant codes for SWIFT and all CRAFT techniques (in accordance with Table 2) by modifying a pre-release version of the OpenIMPACT compiler for the IA-64 architecture targeting the Intel® Itanium® 2 processor. We evaluated the performance of SWIFT and CRAFT codes for a set of benchmarks drawn from the SPEC CPUINT2000, SPEC CPUFP2000, SPEC CPUINT95 and MediaBench suites. The baseline binaries were aggressively optimized OpenIMPACT compilations without any built-in redundancy. 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 time for each benchmark.

The results in Figure 2 show the normalized execution times for SWIFT and CRAFT. 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.

For eight benchmarks, such as `254.gap` and `255.vortex`, the addition of the LVQ structure reduces the execution time when added to SWIFT or CRAFT:CSB. However, for seven of the benchmarks, such as `129.compress` and `164.gzip`, the addition of the LVQ increases the normalized execution time. In these benchmarks, the addition of a duplicate load for every load instruction results in over-subscription of memory functional units. As a result, the compiler generates a worse schedule than for SWIFT. For all of the benchmarks, the addition of the CSB increases performance.

7.2 Reliability

The second aspect of our evaluation is a reliability comparison. The AVF and MWTF reliability metrics were measured for the SWIFT and CRAFT techniques, using the novel fault injection methodology presented in Section 6.2. This is the first known AVF estimation for software-only and hybrid fault-detection techniques.

7.2.1 Architectural Vulnerability Factor

Using the new framework, we evaluate the AVF of the baseline code without built-in redundancy (NOFT), the software-only approach (SWIFT), as well as, the three hybrid approaches (CRAFT:CSB, CRAFT:LVQ, CRAFT:CSB+LVQ). As explained in Section 6.2, our reliability evaluation consisted of microarchitectural and architectural simulations. 5,000 microarchitectural faults were injected for each benchmark for each system for each structure we were evaluating. We evaluated 3 hardware structures across each of the 5 systems using 15 different benchmarks for a total of 1.125 million transient fault injections.

We evaluated the reliability of the systems using an Intel® Itanium® 2 as the baseline microarchitecture. The Liberty Simulation Environment's simulator builder [22, 23] (LSE) was used to construct our cycle-accurate performance models. Our baseline machine models the real

	Integer Register File (GR)				Predicate Register File (PR)				Instruction Fetch Buffer (IFB)			
System	AVF	AVF_{dSDC}	AVF_{pSDC}	AVF_{DUE}	AVF	AVF_{dSDC}	AVF_{pSDC}	AVF_{DUE}	AVF	AVF_{dSDC}	AVF_{pSDC}	AVF_{DUE}
NO FT	18.65 %	7.89 %	10.76 %	0.00 %	1.58 %	0.55 %	1.03 %	0.00 %	8.64 %	4.48 %	4.16 %	0.00 %
SWIFT	26.78 %	0.13 %	1.69 %	24.96 %	3.95 %	0.03 %	0.01 %	3.91 %	19.17 %	0.65 %	0.53 %	17.99 %
CRAFT:CSB	23.80 %	0.09 %	0.41 %	23.30 %	3.49 %	0.01 %	0.01 %	3.47 %	19.82 %	0.02 %	0.23 %	19.57 %
CRAFT:LVQ	25.14 %	0.12 %	1.69 %	23.33 %	3.25 %	0.04 %	0.02 %	3.19 %	16.07 %	0.72 %	1.18 %	14.17 %
CRAFT:CSB+LVQ	22.95 %	0.04 %	1.39 %	21.52 %	2.68 %	0.01 %	0.02 %	2.65 %	14.97 %	0.01 %	1.05 %	13.91 %

Table 3. AVF Comparison for the different reliability systems

hardware with extremely high fidelity; its IPC matches that of a real Intel® Itanium® 2 to within 5.4% [15].

We evaluated the AVF of the Intel® Itanium® 2 integer register file (GR), predicate register file (PR), and instruction fetch buffer (IFB). The instruction fetch buffer can hold up to 8 bundles; each 128-bit bundle consists of three 41-bit instructions and a 5-bit template field that specifies decode and dispersal information. All 128 bits of a bundle are susceptible to faults. The IFB can have a maximum of 8 bundles in the buffer, but may have fewer and all bits in unoccupied entries are unACE.

If the corrupted instruction was consumed, then native Intel® Itanium® 2 execution was continued till program completion. By allowing the program to run to completion, we determined if the corrupted and consumed bit caused a error in the final output. As previously mentioned in Section 6.2, 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 the fault was detected, the corrupted bit was a DUE bit, and if the program had incorrect output, the corrupted bit was an SDC bit, either dSDC or pSDC.

If the program ran to completion, but produced the wrong output, the corrupted bit was considered an dSDC bit. Certain corrupted bits caused the program to terminate with a noticeable event, and those bits were considered pSDC. In this work, we consider segmentation faults, illegal bus exceptions, floating point exceptions, NaT consumption faults, and self-termination due to application-level checking as pSDC events.

We also evaluated the reliability of the integer and predicate register files. The GR 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, since a fault can never corrupt `r0`, which always contains zero. The predicate register file contains 64 1-bit registers, all of which are vulnerable to faults, except `p0` which always contains true.

After a fault occurred, the detailed simulator monitored the bit for the first use of the corrupted register. Instructions in-flight past the register read (REG) stage 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 read and the transient fault will not affect these instructions. The detailed simulation recorded the first instance of an instruction consuming the corrupted register, if one existed. If the fault was never propagated, then it had no effect on architecturally correct execution (i.e. was unACE).

Table 3 summarizes the GR, PR, and IFB AVF analysis for each of the systems. 18.65% of the bits in the integer register file are ACE bits for NOFT and consequently, the AVF of the structure is 18.65%. All of the ACE bits for the NOFT system are SDC bits, either pSDC or dSDC. For CRAFT:CSB+LVQ, the overall AVF is 22.95%, 4.30% larger than the NOFT system. Although the total AVF is larger, the distribution between SDC and DUE is much more desirable. The SDC of CRAFT:CSB+LVQ is 1.43% (0.04% dSDC and 1.39% pSDC) and the DUE is 21.52%. Only 1.43% of the bits in the structure cause an error in CRAFT:CSB+LVQ, as compared with 18.65% of the bits that cause an error in the unprotected system.

Table 3 illustrates the trade-offs made between SDC and DUE for all of the systems. As explained in Section 5, both the store buffer and load value queue compensate for points of failure in the software only approach as shown in the CRAFT:CSB dSDC rate of 0.09% and CRAFT:LVQ dSDC rate of 0.12% compared with the SWIFT dSDC rate of 0.13%. The hardware structures target different failure opportunities and can thus be combined for additive reliability benefit. The CRAFT:CSB+LVQ system reduces dSDC to almost zero for all structures; of the 225,000 injection experiments for this technique, only 47 created an dSDC situation.

The AVF of the predicate register file, also shown in Table 3. Similar to the GR, the hybrid techniques reduce the dSDC of the PR compared with the NOFT and SWIFT techniques. The AVF of the predicate register file is much lower than that of the integer register file, due to the very short lifespan of predicate registers. A corrupted integer register is more likely to be consumed before being overwritten than a corrupted predicate register.

The average instruction buffer AVF is listed in Table 3, and Figure 3 shows the instruction fetch buffer AVF for each of the benchmarks and systems evaluated. The IFB average and the GR average have similar trends with respect to the hybrid evaluation. When looking at the AVF for individual benchmarks, different benchmarks have different reliability gains. Benchmarks such as `g721dec` and `129.compress` have much larger dSDC reliability gains when comparing the hybrid techniques to NOFT executions because the NOFT executions have a large dSDC. Bench-

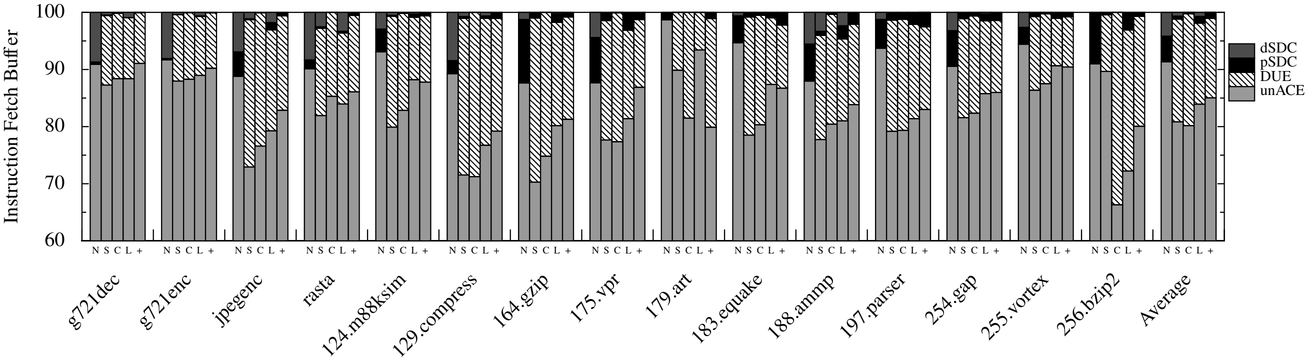


Figure 3. AVF by benchmark and system for the instruction fetch buffer

marks such as `164.gzip` and `197.parser`, on the other hand, do not have much dSDC in the NOFT case, so although the hybrid dSDC is reduced, the benefit is not as substantial.

Despite the high reliability of CRAFT:CSB+LVQ for all structures, the dSDC is still nonzero. This is due to two reasons. First, to maintain the calling convention, argument registers are checked against their duplicate counterparts and only one copy is passed to the function. Inside the function, the values are duplicated 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.

Another potential source of dSDC is a limitation of the control-flow checking. Register indirect branches were not protected in our implementation, but the control-flow checking technique can be extended to handle them. Furthermore, the signature control-flow checking suffers from the fundamental limitation that if a transient fault occurs on the address of the target such that the faulty target is the address of a store or a system call, then an erroneous value may be sent to memory before any fault is detected.

7.2.2 Mean Work To Failure

As described in Section 6, MWTF is a more appropriate reliability metric with which to compare a wide variety of fault-detection techniques because it encompases execution time as well as architectural vulnerability. Figure 4 shows the MWTF$_{SDC}$ and MWTF$_{dSDC}$ of the hybrid and software systems normalized to the NOFT baseline for the three hardware structures we have analyzed. Since all evaluations were done on the same machines in the same environment, those factors are canceled by normalizing to the NOFT baseline.

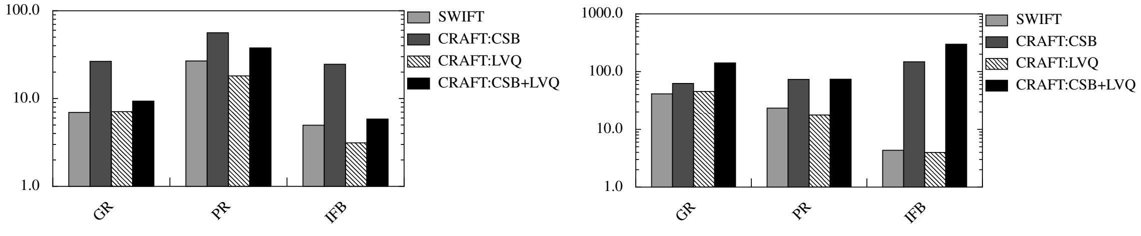
The normalized mean work to SDC failure of Figure 4(a) elucidates the reliability differences between the techniques. The SWIFT technique increases the MWTF$_{SDC}$ over the NOFT baseline by 7x, 27x, and 5x for the GR, PR, and IFB respectively. Although the software and hybrid techniques increase the execution time compared with unprotected baseline, all techniques for all structures have a larger MWTF than the baseline. The decrease in vulnerability has more impact than the decrease in performance, resulting in the ability to perform more work between failures.

The CRAFT:CSB technique has the largest MWTF$_{SDC}$ of all the reliability techniques, 26x, 56x, and 25x for the GR, PR, and IFB respectively. This is due to the decreased AVF$_{SDC}$, as shown in Table 3, and the decrease in execution time, as shown in Figure 2, compared to the other techniques. CRAFT:LVQ has a lower average MWTF$_{SDC}$ than SWIFT because it has roughly the same definite SDC but larger potential SDC.

The increase in pSDC is caused by segmentation faults that terminate the program. In the systems without the LVQ, the addresses of load instructions are validated in software. Faults that corrupt the initial load addresses that would cause segmentation faults are caught by the validation instruction. However, in the LVQ systems, the loades are executed *before* the validation, and so segmentation faults may occur if the addresses are corrupted. It is possible to extend our technique to delay the raising of exceptions until the corresponding load is checked in the LVQ.

The CRAFT:CSB+LVQ technique has better performance and SDC reliability than SWIFT and thus the MWTF$_{SDC}$ is larger. The combined CSB+LVQ technique has a smaller MWTF$_{SDC}$ than the CRAFT:CSB technique because it also suffer from higher pSDC.

The real benefit of the hybrid techniques is shown in Figure 4(b), the normalized mean work to dSDC failure, by excluding the SDC faults than can potentially be detected. The MWTF$_{dSDC}$ of the CRAFT:CSB+LVQ increases by 141x, 73x, and 298x for the GR, PR, and IFB over the NOFT baseline. SWIFT also increases MWTF$_{dSDC}$ compared with NOFT, but only by 41x, 23x, and 4x. The CRAFT:CSB+LVQ has a larger MWTF$_{dSDC}$ than the other hybrid systems because it has, on average, both better performance and better dSDC susceptibility. Although it has slightly better performance than the CRAFT:CSB system, it eliminates the window of vulnerability for load



(a) Mean Work to SDC failure

(b) Mean Work to dSDC failure

Figure 4. Normalized Mean Work To Failure by system and structure.

instructions and thus has better dSDC reliability. The CRAFT:LVQ has roughly the same performance and same dSDC reliability as SWIFT, and so the MWTF$_{dSDC}$ for CRAFT:LVQ is similar to SWIFT.

7.3 Hardware

Hardware cost is an important consideration for processor architects trying to meet soft-error goals. As seen from Table 1, 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. With these minimal hardware additions, CRAFT techniques are able to achieve almost-perfect reliability. Further, the single-program performance degradation of CRAFT techniques is comparable to that of heavyweight hardware-only schemes that rely on expensive hardware features like simultaneous multithreading or chip multiprocessing besides including all the low-cost hardware features employed by CRAFT.

CRAFT techniques are not strictly superior in terms of performance or reliability when compared to hardware-only techniques like RMT, nor are they less expensive than software-only techniques like SWIFT. But alternatively, RMT is more costly than CRAFT, while SWIFT does not perform as well and is not as reliable as CRAFT. The CRAFT hybrid techniques also offer designers the ability to add hardware in a piecemeal fashion. For example, if SWIFT is insufficient to meet a designer's reliability needs, but CRAFT:CSB is sufficient, then he or she only needs to add the CSB to the hardware design, rather than implementing all of the hardware demanded by RMT.

8 Conclusion

In this paper, we identified a significant unexplored portion of the fault-detection design space situated between hardware-only and software-only fault-detection mechanisms. To fill this void, we proposed using hybrid hardware/software fault detection systems which capture the virtues of existing systems while mitigating the costs. We developed CRAFT, a suite of three such techniques, to illustrate the potential of hybrid systems. CRAFT augments the best known software-only technique, SWIFT, with lightweight hardware structures borrowed from hardware-only techniques such as RMT. Our results indicate that adding or enhancing even a *single* hardware structure can, in many cases, vastly improve the reliability of a system. None of the evaluated techniques (hardware, software, or hybrid) were perfect in all respects, however, each had its own specific virtues making it the most appropriate for a particular set of design constraints.

In addition to exploring the hybrid design space, this paper introduced a new methodology for evaluating fault-detection systems. As part of this methodology we introduced a new evaluation metric, Mean Work To Failure, which allows objective, quantitative comparisons between hardware, software, and hybrid systems despite varying hardware *and* software. The methodology also includes a new evaluation framework that combines microarchitectural and architectural simulations to allow fast and highly accurate computation of MWTF and AVF. We believe that the CRAFT techniques and the evaluation framework presented in this paper form a solid foundation for continued research in hybrid fault-tolerance systems.

Acknowledgments

We thank the entire Liberty Research Group as well as John Sias 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] R. C. Baumann. Soft errors in advanced semiconductor devices-part I: the three radiation sources. *IEEE Transactions on Device and Materials Reliability*, 1(1):17–22, March 2001.

[2] A. Benso, S. D. Carlo, G. D. Natale, and P. Prinetto. A watchdog processor to detect data and control flow errors. In *Proceedings of the 9th IEEE International On-Line Testing Symposium*, 2003.

[3] D. C. Bossen. CMOS soft errors and server design. In *IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals*, pages 121_07.1 – 121_07.6, April 2002.

[4] E. W. Czeck and D. Siewiorek. Effects of transient gate-level faults on program behavior. In *Proceedings of the 1990 International Symposium on Fault-Tolerant Computing*, pages 236–243, June 1990.

[5] 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.

[6] 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.

[7] S. Kim and A. K. Somani. Soft error sensitivity characterization for microprocessor dependability enhancement strategy. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 416–425, September 2002.

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

[9] 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.

[10] 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.

[11] T. J. O'Gorman, J. M. Ross, A. H. Taber, J. F. Ziegler, H. P. Muhlfeld, I. C. J. Montrose, H. W. Curtis, and J. L. Walsh. Field testing for cosmic ray soft errors in semiconductor memories. In *IBM Journal of Research and Development*, pages 41–49, January 1996.

[12] N. Oh, P. P. Shirvani, and E. J. McCluskey. Control-flow checking by software signatures. In *IEEE Transactions on Reliability*, volume 51, pages 111–122, March 2002.

[13] N. Oh, P. P. Shirvani, and E. J. McCluskey. Error detection by duplicated instructions in super-scalar processors. In *IEEE Transactions on Reliability*, volume 51, pages 63–75, March 2002.

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

[15] D. A. Penry, M. Vachharajani, and D. I. August. Rapid development of flexible validated processor models. Technical Report Liberty-04-03, Liberty Research Group, Princeton University, November 2004.

[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] 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.

[19] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In *Proceedings of the 3rd International Symposium on Code Generation and Optimization*, March 2005.

[20] P. P. Shirvani, N. Saxena, and E. J. McCluskey. Software-implemented EDAC protection against SEUs. In *IEEE Transactions on Reliability*, volume 49, pages 273–284, 2000.

[21] 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.

[22] M. Vachharajani, N. Vachharajani, and D. I. August. 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)*, pages 195–206, June 2004.

[23] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. Microarchitectural exploration with Liberty. In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO)*, pages 271–282, November 2002.

[24] 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.

[25] 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.

[26] 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.

[27] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel. Characterizing the effects of transient faults on a high-performance processor pipeline. In *Proceedings of the 2004 International Conference on Dependendable Systems and Networks*, pages 61–72, June 2004.

[28] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt. Techniques to reduce the soft error rate of a high-performance microprocessor. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, 2004.

[29] 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.