Automatic Instruction-Level Software-Only Recovery

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

Abstract

As chip densities and clock rates increase, processors are becoming more susceptible to transient faults that can affect program correctness. Computer architects have typically addressed reliability issues by adding redundant hardware, but these techniques are often too expensive to be used widely. Software-only reliability techniques have shown promise in their ability to protect against soft-errors without any hardware overhead. However, existing low-level software-only fault tolerance techniques have only addressed the problem of detecting faults, leaving recovery largely unaddressed. In this paper, we present the concept, implementation, and evaluation of automatic, instruction-level, software-only recovery techniques, as well as various specific techniques representing different trade-offs between reliability and performance. Our evaluation shows that these techniques fulfill the promises of instruction-level, software-only fault tolerance by offering a wide range of flexible recovery 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 [3, 14, 26], 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, that do not cause permanent damage, but may result in incorrect program execution by altering signal transfers or stored values.

When cost is not an issue, system designers typically address transient faults by relying on large amounts of redundant hardware [8, 27, 32, 33]. While effective, this redundancy is prohibitively expensive for arenas outside of the high-end, high-availability market, rendering these techniques impractical for the desktop and embedded computing markets. For example, protecting the register file with ECC has shown to be extremely costly in terms of both performance [28] and power [19].

To provide protection when hardware costs are prohibitive, software-only approaches have been proposed as alternatives [15, 17, 25, 29]. In particular, techniques such as SWIFT [23] have demonstrated that high reliability can be achieved through a software-only fault-detection solution which degrades performance modestly. These software-only reliability techniques are valuable because they do not require any hardware support. They can be applied to future designs without any hardware changes or even to currently deployed systems. Software-only approaches also allow for software-control; the user, the application, or the system may dynamically reconfigure the trade-off between reliability and performance after the system has been deployed to best suit varying conditions.

However, detecting faults is only part of the path to full fault tolerance. In order to truly be reliable, a system must also be able to recover from faults. Until now, all proposed low-level software-only techniques of which we are aware have addressed only fault detection, not fault recovery. Although this prevents faults from corrupting data, it does not allow the application to correctly run to completion in the presence of a fault.

In this paper, we present three novel, software-only recovery techniques at the compiler level which offer varying levels of protection. The first is SWIFT-R, which is based on SWIFT [23], an existing software-only detection scheme. The SWIFT-R technique intertwines three copies of a program and adds majority voting before critical instructions offering near-perfect reliability for those applications that require it.

The second technique we present is TRUMP (Triple Redundancy Using Multiplication Protection), which intertwines the original program with an AN-encoded version of the program. Section 4.1 will give an overview of AN-encoding, a more efficient representation of redundant information than simple triplication. At certain points in the program, the original and AN-encoded versions are compared and recovery code is triggered if a mismatch is detected. The AN-encoding of TRUMP allows recovery although only two versions of the program are computed. Although TRUMP's AN-encoding is not as general as SWIFT-R's triple-modular redundancy, rendering it unable to protect certain portions of programs, TRUMP's redundant computation is much less onerous, providing an alternative for applications that cannot afford the performance penalty of SWIFT-R, but could benefit from moderate protection.

The last technique, MASK, dynamically enforces invariants that can be proved true statically. By merely asserting statically known facts at various points in the program, MASK is able to improve the reliability of the system without adding redundancy. The MASK technique is more lightweight than the other two techniques but can still substantially increase reliability in some cases.

We implemented SWIFT-R, TRUMP, and MASK in a

compiler and evaluated them in isolation, as well as in interesting hybrid combinations. The evaluation shows that these techniques offer a wide spectrum of viable options for fault-tolerance that can be deployed on existing systems without any hardware additions. These techniques range from the lightweight MASK which does not incur any significant performance penalty to SWIFT-R which incurs a 99% increase in execution time. Similarly, these techniques reduce the number of incorrect program executions in the presence of faults from 1.24% for the lightweight technique to 89.39% for the heavyweight technique.

This paper contributes the concept, implementation, and evaluation of instruction-level, software-only recovery techniques. These techniques will allow designers to increase reliability with reasonable performance costs without having to design or deploy new hardware.

The rest of the paper is organized as follows. Section 2 introduces the fault model and terminology that will be used in the remainder of this paper, and also gives a description of the SWIFT technique. Sections 3, 4, 5 describe the SWIFT-R, TRUMP, and MASK techniques respectively. Section 6 discusses various hybrid combinations of these techniques. Section 7 provides an experimental evaluation of the performance and reliability of each of the techniques. Section 8 describes related work. The paper concludes with Section 9.

2 Background

This section first describes the fault model assumed throughout this paper. It then briefly describes the SWIFT fault-detection technique (Section 2.2), the basis of several recovery techniques presented in this paper. SWIFT has been shown to protect against most faults in various structures including the register files and the instruction buffer [24]. Note that for the remainder of the paper, we will not concern ourselves with SWIFT's control-flow protection technique which is used to detect faults that corrupt the program counter. It is largely orthogonal to SWIFT's instruction-duplication technique and can be implemented on top of any of the techniques that follow. For the purposes of this work, this paper assumes that no faults occur to the program counter register.

2.1 Fault Model

Throughout this paper, the commonly used single-event upset (SEU) fault model will be assumed. In the SEU model, exactly one bit-flip in one state element will occur throughout a particular execution of the program. The techniques presented also tolerate a wide variety of multi-bit errors, although we do not quantify this effect.

In order to evaluate the reliability of a system, faults are classified according to their effect on the final output of the program in the presence of the fault. If the fault causes the execution to be abnormally terminated due to a segmentation fault, the fault is categorized as *SEGV*. If the program completes execution, but does not produce correct output, then the fault is categorized as an *SDC* (silent data corruption). Finally, if the program completes execution and the output is correct, the fault is categorized as *unACE* (unnecessary for architecturally correct execution) [13]. In this paper, the "reliability" of a system refers to the percentage of faults that are unACE, since SEGV and SDC faults are both deleterious.

```
                          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

Figure 1. SWIFT Duplication and Validation.

2.2 SWIFT

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

Since program correctness is defined by the output of a program, the validation checks must be inserted before any instruction which may potentially generate output. There are two principal methods for user-level code to produce output: memory-mapped I/O and system calls. We first address the issue of memory-mapped I/O.

If all output is produced via memory-mapped I/O, then a program has executed correctly if all correct loads and stores in the program have executed correctly. Under this conservative assumption, data must be validated before all loads and stores. By the same token, the redundancy must also avoid adding any extra stores and loads, lest any unwanted I/O be performed. The lack of redundancy in memory accesses typically does not significantly impact reliability, since memory and caches are often protected against transient faults by ECC.

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. As mentioned earlier, program correctness can only be ensured if all loads and stores execute correctly. In the case of load instructions, that amounts to verifying the address of the load. Instruction **1** is a comparison inserted to ensure that the address of the subsequent load matches its duplicate address. Furthermore, since a redundant load instruction cannot be inserted as the load may be uncacheable [22, 24], to set `r3'`, the technique must find an alternative to redundantly executing the load. In this case, instruction **2** accomplishes this by simply copying the result of the load instruction into its 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 and to avoid storing values to incorrect addresses, the technique must check that both the

```
                          1:  br faultDet, P0 != P0'
call otherFunc                call otherFunc
                          2:  mov R0'= R0
                          3:  br faultDet, r1 != r1'
                          4:  br faultDet, r2 != r2'
br r1 == r2, label            br r1 == r2, label
```

(a) Original Code (b) SWIFT Code

Figure 2. SWIFT Branch and Function Call Validation.

address and value match their redundant copy. If a difference is detected, then a fault has occurred and the system is notified via instructions **4** or **5**. Otherwise, the store proceeds normally.

Although in this 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.

Checking at loads and stores is sufficient to protect against a large number of faults. However, as mentioned earlier, programs may also generate output via system calls, or more generally, via external libraries. Since external code may not have any protection, the best SWIFT can do is verify that all of the inputs to the function or system call are correct. There are two classes of inputs arguments: register and memory.

Parameters passed through memory do not need to be rechecked since they will have already been checked via the store that placed them in memory. Thus, the only parameters needing special handling are the parameters passed via registers. Just as the compiler inserts checks to compare the input to each store instruction against its redundant copy, the compiler inserts check instructions for register arguments before function calls.

Unfortunately, checking at these points alone, namely before loads, stores, and function calls, is not enough to protect against faults which affect branch outcomes. If a fault occurs on a data slice which only feeds a branch, then an incorrect execution path may be taken and incorrect loads and stores may be executed, although no fault will be detected. In order to protect against this, the technique also verifies the input registers to any branch predicate. The technique can also verify the input to any register indirect jump, although we do not implement it here since register indirect jumps are relatively infrequent.

An example of this protection is given in Figure 2. Instructions **3** and **4** check that the source registers to the conditional branch are correct. Instruction **1** checks that the input parameter register `P0` is equal to its redundant version before making the external function call to `otherFunc`. The function call is akin to a load instruction not only in the fact that the inputs must be checked, but also that they cannot be safely duplicated. Therefore, in order to produce a redundant copy of the return value, here given in `R0`, instruction **2** must be executed.

While SWIFT has some vulnerabilities which will be described in Section 3.2, previous work [24] has shown it to be effective at detecting most faults in many parts of the system.

```
                          1:  majority(r4,r4',r4'')
ld r3 = [r4]                  ld r3 = [r4]
                          2:  mov r3' = r3
                          3:  mov r3''= r3
add r1 = r2, r3               add r1 = r2, r3
                          4:  add r1'= r2',r3'
                          5:  add r1''= r2'',r3''
                          6:  majority(r1,r1',r1'')
                          7:  majority(r2,r2',r2'')
st [r1] = r2                  st [r1] = r2
```

(a) Original Code (b) SWIFT-R Code

Figure 3. SWIFT-R Triplication and Validation.

3 SWIFT-R

This section introduces the novel recovery extensions to SWIFT which comprise SWIFT-R and also discuss SWIFT-R's vulnerabilities and limitations.

3.1 The SWIFT-R Transformation

The SWIFT transformation can be seen as a double-modular redundancy implemented in software. Double redundancy provides detection but not recovery. In order to achieve recovery, it is natural to move to triple-modular redundancy.

The SWIFT-R transformation, instead of creating one redundant copy as in SWIFT, creates two redundant copies. Having three copies means that should a fault corrupt any one version's computation, two other versions will still have the correct computation. By using a simple majority voting scheme, any single-bit fault can be corrected.

The code in Figure 1 is shown again in Figure 3, except that SWIFT-R code is given instead of SWIFT code. Instruction **4** duplicates the previous `add` instruction, just as in SWIFT. However, the SWIFT-R transformation also inserts instruction **5**, a third version of the add instruction which uses a third set of registers, here denoted by `r1''`, `r2''`, and `r3''`. Similarly, after the load instruction, instead of a single move instruction (instruction **2**), SWIFT-R also inserts a second move instruction (instruction **3**).

Furthermore, the fault detection code used by SWIFT has been replaced, in SWIFT-R, by recovery code at instructions **1**, **6**, and **7**. The recovery code is simply a majority voting procedure — if two versions of a register, `r1` and `r1'` for example, have the same value, but the third version, `r1''`, does not, then `r1''` is set to the value in `r1`/`r1'`, correcting the corrupted value of `r1''`.

3.2 Windows of Vulnerability

SWIFT-R, like all software-only reliability techniques, including SWIFT and the techniques which will follow, has small windows of vulnerability. There are four principal classes of vulnerabilities:

1. **Between validation and use**. Because the validation occurs in software, the verification of a value must invariably happen some number of instructions before the use of that value. Consider the region in Figure 3 between instruction **7** and its subsequent store (`st`). If a fault occurs to `r2` during this time, a faulty value will be transmitted to memory. Similarly, if a fault occurs to `r1` between instruction **6** and the `st` instruction then the store will go to an incorrect address. While it is im-

possible to remove this vulnerability altogether, it can be partially mitigated by scheduling the verification instructions as closely as possible to their corresponding uses.

2. **Before a value is copied**. At various points, such as after load instructions and after function call returns, values are triplicated. If a fault occurs to the register to be copied before all copies have been made, then multiple copies of the register will be corrupted. Consider the region in Figure 3 between the load (`ld`) and the move in instruction **2**. If a fault occurs to `r3` at this point, then `r3'` and `r3''` will also be incorrect, compromising all future checks on those values. Once again, scheduling can minimize the window of vulnerability but it cannot eliminate it altogether.
3. **Certain faults to opcode bits**. Faults corrupting the opcode bits of an instruction may transform it into a store or a branch. This will cause memory to be corrupted or incorrect control-flow to be taken. Since no checks are inserted before most instructions, any non-load/store/branch instructions transformed into a load, store, or branch may go uncaught by our techniques. Note that other faults to the opcode bits will be caught by our techniques.
4. **Deadlock**. Since the micro-architectural state of the processor is not exposed to the software, software-only techniques are unable to protect against faults on some portions of the micro-architectural state such as parts of the control logic. If faults to such state cause deadlock, any software-only scheme will be unable to make forward progress.

4 TRUMP

This section describes the second novel software-only recovery technique, TRUMP. Section 4.1 gives a brief overview of AN-codes which form the theoretical backdrop for the TRUMP technique. AN-codes allow redundancy to be represented more compactly, ultimately enabling the triple redundancy of SWIFT-R to be contained in two registers instead of three. Section 4.2 presents the TRUMP transformation itself.

4.1 AN-codes

AN-codes are a class of arithmetic codes where the codeword is simply the original data multiplied by a constant, A. The fact that AN-codes are arithmetic codes, codes which are preserved across arithmetic expressions, is evident by standard algebra:

$$(Ax) + (Ay) = A(x + y) \quad (1)$$

$$(Ax) \cdot k = A(x \cdot k) \quad (2)$$

AN-codes can be used to detect errors by verifying that the codeword is divisible by A. Precisely, C is a valid codeword only if $C \equiv 0 \pmod{A}$. The choice of A has a large impact on the implementation cost as well as the reliability of the resulting code. $A = 2^n - 1$ is a particularly good choice with respect to both of these.

Let us first consider the reliability ramifications of this choice. Any single-bit fault to a codeword may be considered either an addition or subtraction of 2^k for some k. Observe that $2^k \neq 2^n - 1$ for any $n > 0$. Therefore,

$$C \pm 2^k \equiv \pm 2^k \pmod{A} \not\equiv 0 \pmod{A}$$

By the above proof, the faulty AN codeword is guaranteed not to be divisible by A. Thus, this choice of A will be able to detect any single-bit faults to the codeword. Although we will not prove it here, this choice of codeword can also protect against a large number of multi-bit faults.

$A = 2^n - 1$ is also a convenient implementation for performance because a multiplication by an A of this type can simply be computed as a shift left by n and a subtraction, specifically, $Ax = (x \ll n) - x$.

The choice of A also determines how many bits will be required to represent the codeword. For $A = 2^n - 1$, n extra bits will be needed to represent the codeword. Therefore, in our implementation, we choose the smallest nontrivial n, namely $n = 2$ and $A = 2^2 - 1 = 3$, to minimize the additional bits necessary for storage.

```
let x = original copy
let y = AN-encoded copy

if (3x ≠ y)
  if (y ≡ 0 (mod 3))
    x = y/3
  else
    y = 3x
```

Figure 4. TRUMP recovery pseudo-code.

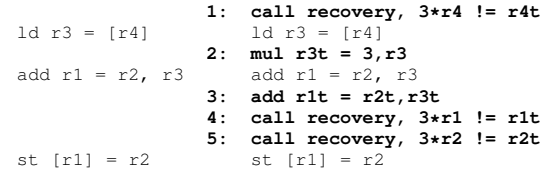
4.2 The TRUMP Transformation

In TRUMP, we exploit AN-codes to implement software-only recovery more efficiently than in SWIFT-R. As noted in the prior section, an AN-code with $A = 3$ is sufficient to detect any single-bit error. We can extend its detection capability to recovery by adding one extra, non-AN-encoded version.

TRUMP essentially has two copies of every value, akin to SWIFT. However, unlike SWIFT, one copy of the data is AN-encoded. Under this scheme, the program detects a fault whenever the original copy multiplied by A does not match the AN-encoded copy. If they do not match, then the code can recover by inferring which copy is correct. This can be done by making use of the result in Section 4.1. If the AN-encoded copy is divisible by A, then we can surmise that the fault struck the original copy. If it is not, then the AN-encoded copy was struck and the original is correct. Pseudo-code for this recovery sequence is shown in Figure 4. While it may be costly due to the division and modulo operations, these instructions are only executed during fault recovery, which is relatively rare.

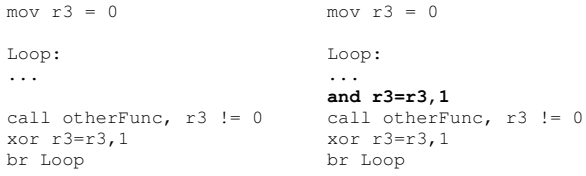
An example of the TRUMP transformation is shown in Figure 5. Note that the redundant trump registers are denoted by appending a '`t`' to the register's name. Although we have shown code with multiplications here for brevity, note that we implement multiplications with the faster combination of shifts and adds.

As implemented in previous software-only reliability techniques, the load address must be checked before load instruc-



(a) Original Code (b) TRUMP Code

Figure 5. TRUMP example.



(a) Original Code (b) MASK Code

Figure 6. MASK example.

tions. Instruction **1** performs this check by ensuring that three times the original value is equal to the redundant value. If there is a mismatch, the recovery code shown in Figure 4 is called. Similarly, before the store instruction, the operands of the store are checked in instructions **4** and **5**.

Also similarly to previous software-only reliability techniques, the result of the load instruction must be copied into the redundant register, as in instruction **2**. In TRUMP, instead of a simple move, a multiplication is performed in order to ensure that the redundant copy is properly AN-encoded. Finally, instruction **3** performs a redundant `add` instruction. Recall that AN-codes are arithmetic codes, which means that codewords are preserved through arithmetic operations. Therefore this instruction does not have to be altered in any way from the SWIFT version.

Thus, TRUMP offers recovery similar to SWIFT-R, but only requires two independent versions.

4.3 Applicability

In addition to the vulnerabilities which all software-only recovery schemes have, described in Section 3.2, TRUMP has two primary limitations that must be kept in mind. First of all, AN-codes do not propagate through many logical operations, such as `and` and `or` [18], and therefore cannot be applied to certain dependence chains. Secondly, a register can never assume a value greater than $\frac{2^M}{A}$, where M is the number of bits in that register. Otherwise, the AN-encoded version of the register will overflow. In order to avoid this situation, TRUMP can only be applied on dependence chains whose values never exceed $\frac{2^M}{A}$. If the compiler cannot statically prove that a certain dependence chain has this property, it has no choice but to leave it at least partially unprotected. Fortunately, restrictions on valid memory addresses on most architectures provide ample spare bits for the TRUMP transformation to be applied to pointers. Also, code written in languages with primarily 32-bit data types, such as C, will typically not utilize many bits when executed on 64-bit architectures. These two phenomena allow TRUMP to be applied widely on most applications.

5 MASK

This section introduces the MASK technique, which enforces statically known invariants in order to eliminate faults that can be reasoned away. Using these invariants, MASK can remove faults that would otherwise be deleterious, thus increasing reliability without redundant execution.

The MASK technique is best illustrated through example. Consider the code given in Figure 6, loosely culled from `adpcmdec`, an adaptive PCM decoder benchmark from MediaBench [10]. In this snippet, the `otherFunc` function is called every other iteration of the loop, via the guarding register `r3`. Any faults on the lowest bit of this register will certainly be detrimental to the program, causing it to execute or not execute `otherFunc` erroneously for every subsequent iteration. Furthermore, any fault on any of the other bits of register `r3` will cause `otherFunc` to be erroneously executed every iteration, instead of every other iteration as originally intended. On a 64-bit system, $\frac{63}{64}$ of the faults will be of this latter type while only $\frac{1}{64}$ of the faults will be of the first type. It is this latter and more common type of fault that MASK attempts to resolve.

By statically analyzing the code, the compiler can know that all but the lowest-order bit of `r3` must necessarily be zero. The MASK technique enforces this invariant by adding the boldfaced instruction shown in Figure 6. If a fault occurs to any of the bits which should be zero, then it will be masked out and will not affect any subsequent computation. Therefore, the reliability of register `r3` will be increased by a factor of 64 in our example.

The MASK technique reduces the total number of live bits in the system, thereby increasing the system's resilience against faults; any fault to a dead bit cannot cause the system to produce incorrect output. Although we only evaluate masking with `and` instructions to enforce known-zero bits, the technique could easily be extended to use `or` instructions to enforce known-one bits, or sign-extensions to enforce known-sign bits. The technique could also eventually be extended to take into account higher-level semantic information and programmer annotations.

6 Hybrid Techniques

In this section, we describe four hybrid combinations of SWIFT-R, TRUMP, and MASK.

6.1 TRUMP/SWIFT-R

Although TRUMP can be applied in many situations, it cannot be applied to all. In order to ensure the highest level of reliability, the TRUMP technique can be applied to protect as much of the program as possible, and then the remaining parts of the program can be protected with the universally applicable SWIFT-R technique.

This hybrid technique is called TRUMP/SWIFT-R. In TRUMP/SWIFT-R, every dependence chain is broken up into exactly two continuous segments: the part of the chain where TRUMP can be applied and the part where TRUMP cannot be applied. The technique further places the restriction that the region where TRUMP can be applied must include the uses of this dependence chain, i.e. the end of the dependence

```
                        1:  majority(r4,r4',r4'')
ld r3 = [r4]                ld r3 = [r4]
                        2:  mov r3' = r3
                        3:  mov r3'' = r3
and r3 = r3, 0xFF           and r3 = r3, 0xFF
                        4:  and r3' = r3', 0xFF
                        5:  and r3'' = r3'', 0xFF
                        6:  shl r3t = r3',1
                        7:  add r3t = r3t,r3''
add r1 = r2, r3             add r1 = r2, r3
                        8:  add r1t= r2t,r3t
                        9:  call recovery, 3*r1 != r1t
                        10: majority(r4,r4',r4'')
st [r4] = r1                st [r4] = r1
```

(a) Original Code (b) TRUMP/SWIFT-R Code

Figure 7. TRUMP/SWIFT-R example.

chain. This means that within a dependence chain, there will only ever be one transition from SWIFT-R to TRUMP and not vice versa. This restriction is required because, as we will see, converting SWIFT-R redundancy into TRUMP redundancy is far more affordable than converting TRUMP redundancy into SWIFT-R redundancy, which requires expensive checks and division.

At the point where the SWIFT-R chain ends and the TRUMP chain begins, the SWIFT-R redundancy is converted into TRUMP redundancy. To do this, two of the SWIFT-R values are combined to create a single AN-encoded TRUMP value. This ensures that any faults to either one of the two SWIFT-R values will be properly reflected in the TRUMP value. An example is shown in Figure 7.

In this code sequence, all instructions through the `and` comprise the SWIFT-R portion of the chain, and the remainder comprises the TRUMP portion of the chain. Therefore, the `ld` and the `and` instructions have typical SWIFT-R protection in instructions **1** through **5**. Instruction **6** multiplies `r3'` by two and instruction **7** adds `r3''` to this value. If there were no faults, then $2 \cdot \texttt{r3'} + \texttt{r3''} = 2 \cdot \texttt{r3} + \texttt{r3} = 3 \cdot \texttt{r3}$, the proper AN-encoded value. If either `r3'` or `r3''` has a single-bit fault, then the resulting value, `r3t` will also have the fault and not be divisible by 3. The TRUMP redundant addition is inserted as instruction **8**. The store instruction at the end of the example has its sources protected by both TRUMP and SWIFT-R, and the appropriate recovery code is inserted at instructions **9** and **10**.

6.2 TRUMP/MASK

TRUMP/MASK is the TRUMP and MASK techniques combined. In TRUMP/MASK, it is only desirable to apply the MASK technique to the original version of the code and not to TRUMP's redundant instructions. This is because instructions that are protected by TRUMP are already tolerant of faults and therefore need no additional protection from MASK. However, MASK can be applied on those dependence chains which TRUMP cannot protect. These are often exclusive, since it is typically difficult to prove that any of the bits in the instructions that TRUMP can protect, namely arithmetic operations, are zero, while it is usually much easier to prove that bits are zero in instructions that TRUMP cannot protect, such as logical `and` and `or`. Because of this exclusivity, the TRUMP and MASK techniques are able to complement each other in TRUMP/MASK by protecting different portions of the program.

6.3 Other Hybrids

In this paper, we do not evaluate the combination of SWIFT-R and MASK. Since the MASK technique does not add any redundancy into the program, unlike TRUMP, all of SWIFT-R's redundancy must remain in the program. Therefore, the SWIFT-R/MASK combination would simply consist of SWIFT-R with additional MASK instructions inserted. However, since MASK does not close or reduce any of SWIFT-R's windows of vulnerability, the MASK instructions would be useless. Therefore, we do not evaluate this hybrid combination. For the same reason, we also do not evaluate a TRUMP/SWIFT-R/MASK hybrid.

7 Evaluation

This section evaluates the SWIFT-R, TRUMP, and MASK techniques, as well as the TRUMP/MASK and TRUMP/SWIFT-R hybrid techniques. Each technique was implemented as a pass in the `gcc` compiler, version 3.4.1, targeted for the PowerPC 970. Our additional compilation phase occurs in the backend of the compiler immediately before register allocation and scheduling. We evaluated these techniques on a variety of benchmarks taken from SPEC CPU2000, MediaBench [10], and other benchmark suites. All binaries were compiled with the `-O2` level of optimization and run on an Apple Xserve G5 with a dual-core PPC970FX.

7.1 Reliability

We performed fault injection experiments for each of our techniques in order to evaluate their reliability. In accordance with the SEU model, we inserted exactly one fault per execution. The fault was inserted into a uniformly randomly selected bit in a uniformly randomly selected integer register at a uniformly random dynamic instruction in the program's execution. 250 such runs were performed for each benchmark for each technique and the outcome of each run was recorded. We injected faults into the register file since it has been shown to be one of the leading contributors of soft-errors [20, 30]. We believe the proposed techniques are also able to protect against most errors to other structures such as the ALU, which are nearly impossible to protect with ECC, since errors to these structures will often manifest themselves similar to faults to the register file.

The fault injection infrastructure instrumented the binaries with code which would alter the appropriate bit at the appropriate dynamic instruction. However, the fault injection infrastructure did not permit fault injections into the TOC pointer, a PPC specific register which is a pointer to global data locations. Additionally, since we implemented our transformations before register allocation, we were unable to protect all uses of the stack pointer. Therefore, we also did not inject faults into the stack pointer.

For future work, we plan to extend our infrastructure to allow for injections into the TOC, and to investigate ways of protecting the stack pointer while not being prohibitively costly. Since the PPC64 has 32 registers, an upper bound on the additional SDC for not protecting both of these is

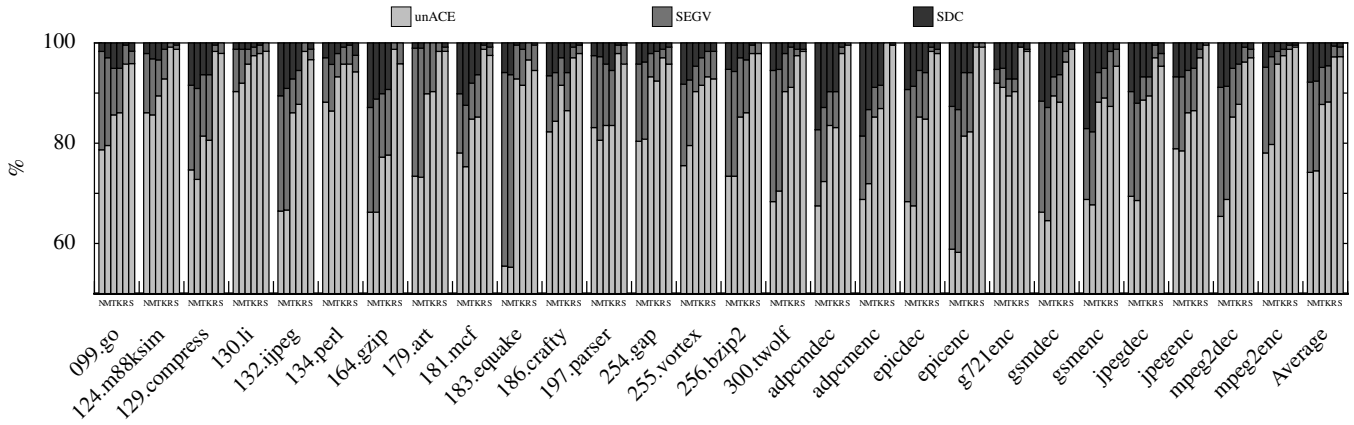


Figure 8. Reliability percentage for NOFT(N), MASK(M), TRUMP(T), TRUMP/MASK(K), TRUMP/SWIFT-R(R), and SWIFT-R(S). The Average is across all benchmarks.

$\frac{2}{32} \approx 6\%$, the probability that a fault will occur in either the stack pointer or the TOC pointer. For these experiments, we also neither inserted faults into, nor duplicated, floating-point registers.

Figure 8 shows the results of the reliability evaluation. The amount of unACE in NOFT is quite high at 74.18%, demonstrating that there is already a large number of dynamically dead registers and masked bits in unprotected code. The SEGV percentage for NOFT is 18.00%, much higher than the SDC percentage of 7.82%. This indicates that faults in registers are much more likely to cause segmentation faults than to corrupt data, suggesting that a great deal of computation for most benchmarks only feeds the addresses of memory accesses rather than the data itself.

As expected, SWIFT-R and its triple-modular redundancy is able to greatly reduce the SEGV and SDC, down to 1.93% and 0.81% respectively. Furthermore, it is consistently low across all benchmarks, indicating universal applicability. The amount of SEGV and SDC is still nonzero, however, due to the windows of vulnerability described in Section 3.2. Since the compiler was not specifically directed to schedule for reliability, the reliability could be further improved, possibly at the cost of some performance, if the compiler were forced to move the checks as close as possible to the uses.

TRUMP also significantly improves reliability over NOFT, albeit not as much as SWIFT-R. TRUMP reduces the SEGV down to 7.39% and the SDC down to 4.88% while increasing the unACE percentage to 87.73%. TRUMP improves SEGV much more dramatically than SDC; this is due to the fact that most pointer dependence chains can be protected by TRUMP while many other dependence chains cannot be. There are two principal reasons for this. First, as mentioned earlier, pointer ranges are limited to valid memory addresses, making it easier to verify that the AN-encoded values will not overflow. Secondly, pointer computations tend to be restricted to simple arithmetic operations such as addition, which can be protected by TRUMP.

As will be shown in the next section, the performance penalty incurred by TRUMP is significantly less than that of SWIFT-R. Coupled with TRUMP's reliability, this technique represents a promising middle-ground for designers who cannot afford to incur the performance penalty of SWIFT-R but who still need significant reliability enhancement. However, designers must keep in mind that TRUMP does not increase reliability uniformly across all benchmarks. For benchmarks that are dominated by arithmetic instructions that can be protected by TRUMP, such as `183.equake` and `mpeg2enc`, TRUMP performs on par with SWIFT-R. For benchmarks, such as `197.parser`, that are dominated by instructions TRUMP cannot protect, such as logical operations, TRUMP's reliability is significantly lower SWIFT-R's.

The MASK technique does not significantly reduce SDC (7.61% versus 7.82% for NOFT) or SEGV (17.89% versus 17.89% for NOFT) across all benchmarks. In fact, in some benchmarks, MASK's reliability can be slightly worse than NOFT's, due to poorer schedules in terms of reliability. However, in other benchmarks, such as `adpcmdec` or `mpeg2dec`, the MASK technique can make a significant difference. In `adpcmdec`, it is able to lower the SDC from 17.30% down to 12.87%, and in `mpeg2dec`, the SEGV is lowered from 25.74% down to 22.57%. This is encouraging and suggests that by looking for further opportunities to enforce program invariants, the MASK technique offers the potential to enhance reliability with practically no cost.

As would be expected, combining the TRUMP and MASK techniques yields reliability similar to that of TRUMP. However, for benchmarks where MASK makes a significant difference, such as `adpcmdec`, TRUMP/MASK fares significantly better than either TRUMP or MASK. For `adpcmdec`, in fact, the SDC of the benchmark is reduced to 4.55% compared with 4.88% for TRUMP and 7.61% for MASK. This additive effect can be attributed to the fact that MASK and TRUMP protect very different types of instructions. TRUMP protects arithmetic instructions while MASK protects instructions where bits can be proved to be zero, which are almost always logical instructions. Their protections complement each other yielding a technique which is more reliable than either TRUMP or MASK alone.

Finally, the TRUMP/SWIFT-R technique performs similarly to SWIFT-R, with a SEGV of 2.14% and a SDC of 0.62%. This implies that the SWIFT-R portions of the code are successfully filling in the gaps in protection left by TRUMP, leaving windows of vulnerability on par with those in SWIFT-R. However, the reliability of TRUMP/SWIFT-R is slightly worse than that of SWIFT-R for some benchmarks because the addition of the TRUMP instructions can sometimes increase the total dynamic number of instructions. This is due to the fact that transitions between SWIFT-R and TRUMP require extra instructions, and that TRUMP's verification sequence is longer than SWIFT-R's. This can ultimately increase register live ranges and the size of windows of vulnerabilities. This suggests that the heuristics of when to apply SWIFT-R and when to apply TRUMP and how to transition from one to the other within a single dependence chain requires additional investigation, which we are pursuing as future work.

7.2 Performance

We collected performance results for each technique using `oprofile` [11] when no faults were injected. Figure 9 shows the execution times for each of our techniques normalized to a baseline build with no additional fault tolerance (NOFT). Note that the bars are clipped at one. In most cases, the performance of MASK is only nominally above one, and in some cases, the performance of MASK bests that of NOFT because the inserted instructions cause slight changes to the scheduling and register allocation heuristics. Consequently, MASK bars appear "missing" for many benchmarks.

Our techniques exhibit a wide range of performance behavior. The low-cost techniques, TRUMP and MASK, have normalized execution times of only 1.36 and 1.00 respectively. Combining these two is the TRUMP/MASK technique, which correspondingly has the larger normalized execution time of 1.37. The higher coverage techniques, SWIFT-R and TRUMP/SWIFT-R, have normalized execution times of 1.99 and 1.98 respectively.

TRUMP/SWIFT-R's execution time is closer to that of SWIFT-R than TRUMP. This implies that TRUMP/SWIFT-R's protection choices track more closely with SWIFT-R than TRUMP, i.e. there are many more instructions protected by SWIFT-R than instructions protected by TRUMP. This is in agreement with the reliability evaluation, showing that the reliability of TRUMP/SWIFT-R was much closer to that of SWIFT-R than that of TRUMP. The performance of TRUMP/SWIFT-R is highly dependent, much like its reliability, on the particular tradeoffs between SWIFT-R protection and TRUMP protection that TRUMP/SWIFT-R makes. The SWIFT-R technique is more expensive than TRUMP in terms of redundancy, because two additional versions of the computation are required instead of one. TRUMP, on the other hand, is more expensive in terms of verification because the AN-encoded and original data must be converted to the same form for comparison. Depending on the ratio of redundant computation to comparison, a TRUMP dependence chain may actually be more costly than a SWIFT-R dependence chain, which accounts for SWIFT-R occasionally outperforming TRUMP/SWIFT-R.

TRUMP/MASK's performance is typically much higher than that of either TRUMP/SWIFT-R or SWIFT-R, but also significantly lower than MASK and on par with TRUMP. This is to be expected as the performance impact of MASK is nearly negligible. The performance is slightly worse than the simple sum of MASK and TRUMP, because each technique alone is able to use some of the previously unused resources, but there are not enough unused resources to support both the MASK and TRUMP protections, thus creating a super-additive performance penalty. Note that in some cases, most notably `mpeg2dec`, TRUMP/MASK outperforms TRUMP, just as MASK occasionally outperforms NOFT. Once again, this is due to unpredictable changes in the scheduler and register allocator as a result of inserting extra instructions.

One interesting observation is that the normalized execution time of all of our techniques, even SWIFT-R, averages far less than three, what one might naïvely expect after triplicating the code. In benchmarks dominated by floating-point instructions which we do not protect, such as `179.art`, we would expect little difference in performance between the various versions of the code, and this is exactly the case. However, the normalized execution time is also far less than three for most integer benchmarks. All of our techniques take advantage of the well-documented existence of unused ILP resources in most modern processors. Since most of the instructions added in SWIFT-R and TRUMP are independent of the original instructions, the reliable code is typically able to make use of previously unused ILP resources. This effect is especially visible in benchmarks which already exhibit poor ILP in NOFT, such as `181.mcf`. `181.mcf` spends a large fraction of its time in memory stalls, consequently, our transformations have a very small impact on the performance. The variety in available ILP leads to wide variations in the performance cost for each benchmark.

In addition to the effect of ILP, the instruction mix of the various benchmarks affects the performance cost of added reliability. Recall that for both TRUMP and SWIFT-R, the protection for most instructions is simply replication. However, whenever there are checks, another more complex sequence of instructions is executed. Although the verification code differs for each technique, in benchmarks where there are many checks, such as `255.vortex` due to a preponderance of loads, the performance impact is typically much higher than benchmarks with smaller numbers of checks and more of their time dedicated to pure computation, such as `300.twolf`.

In summary, SWIFT-R, which has a normalized execution time of 1.99, can significantly improve reliability, increasing unACE to 97.27%. SWIFT-R should be used in situations where high reliability requirements warrant this level of performance degradation. The performance of SWIFT-R can be improved slightly by moving to the hybrid technique TRUMP/SWIFT-R. When the reliability requirements of the system are not stringent enough to warrant SWIFT-R or TRUMP/SWIFT-R, TRUMP or TRUMP/MASK can be used. TRUMP has the much lower normalized runtime of 1.36, while still managing to increase the unACE to 87.73%

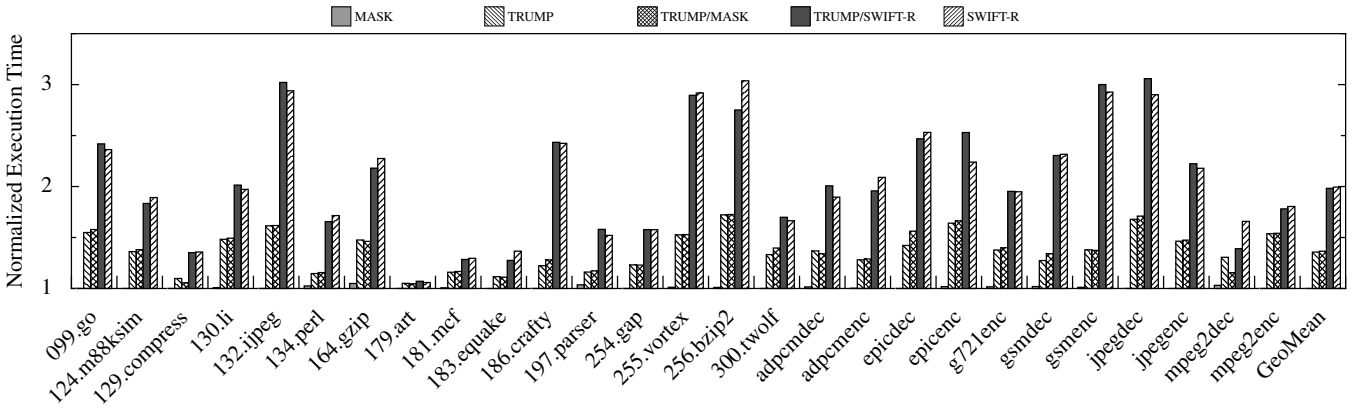


Figure 9. Execution time of MASK, TRUMP, TRUMP/MASK, TRUMP/SWIFT-R, and SWIFT-R normalized to NOFT. The GeoMean is across all benchmarks.

versus 74.18% for NOFT. TRUMP/MASK improves on this slightly, increasing reliability further while having a negligible impact on performance. Finally, when almost no performance degradation can be tolerated, the MASK technique can be used. While MASK does not improve reliability in many cases, it does on some occasions, and since it is essentially free in terms of performance cost, it is almost certainly worthwhile to apply it.

8 Relation to Prior Work

The techniques presented in this paper are the first low-level software-only recovery methods. Previous works have proposed single-threaded fault mitigation techniques both at the source code level [21] and the instruction level [17, 23, 31], but these techniques only address fault detection, not fault recovery. Techniques have also been previously devised which use arithmetic codes, however, they either only address fault detection [16] or require some hardware [5, 6].

There has also been previous work on the notion of multiple execution and majority reconciliation to enable fault recovery. N-version programming (NVP), the process of using N independent modules to do the same task, was originally created to reduce faults in the system design process, by using different teams and compilation tools to develop a software system [1, 2, 4]. While our techniques create multiple versions of the computation, it is notably different from NVP in that NVP attempts to address software programmer errors, while our technique addresses transient faults. Consequently, NVP requires programs to be independently developed multiple times, whereas our techniques are fully-automated, compiler-driven approaches.

Techniques using software-only N-way redundancy have also been applied to parallel systems [7, 9, 12]. In those systems, an application was split into independent tasks and each task was assigned to multiple computation nodes. The results of the computations were compared from the multiple nodes, and a final output was determined, usually by majority voting. These techniques used software recovery, but at a much higher level. By targeting thread-level parallelism rather than instruction-level parallelism, these high-level techniques under-utilize available ILP resources, resulting in a lower transistor efficiency than our techniques. Furthermore, their techniques are only applicable in multi-processor environments, whereas our techniques are all single-threaded.

9 Conclusion

As faults become more commonplace, it will be critical for designers, especially at the embedded and commodity level, to maintain the reliability of their systems without adding hardware and increasing the design complexity of already baroque hardware.

This paper demonstrates that software-only recovery in the face of transient faults is a reliable option. Three novel techniques are introduced – SWIFT-R, an augmentation of the software-only detection scheme, SWIFT; TRUMP, a new recovery system which makes use of AN-codes for implementing redundancy more cost-effectively; and MASK, which enforces program invariants with minimal intrusiveness. The experimental evaluation shows that the techniques represent a wide spectrum of performance and reliability tradeoffs. SWIFT-R offers nearly total protection against faults by reducing the amount of SDC and SEGV events by 89.39%, while MASK offers nearly negligable performance degradation. TRUMP offers an attractive middle ground, reducing SDC and SEGV by 52.48% while only costing 36% in execution time overhead. Combining these techniques into hybrid techniques offers even more options to designers.

Our implementation of these software-only recovery techniques in a production compiler demonstrates that fault recovery can indeed be added into today's designs and today's processors.

Acknowledgments

We thank the entire Liberty Research Group, Shubhendu Mukherjee, and the anonymous reviewers for their support during this work. This work has been supported by the Intel Corporation. Opinions, findings, conclusions, and recommendations expressed throughout this work are not necessarily the views of Intel Corporation.

References

[1] A. Avizienis. The methodology of N-version programming. Chapter 2 of Software Fault Tolerance, M. R. Lyu (ed.), Wiley, 23-46, 1995., 1995.

[2] A. Avizienis and L. Chen. On the implementation of N-version programming for software fault tolerance during execution. In *COMPSAC*, pages 149–155, 1997.

[3] 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.

[4] S. S. Brilliant, J. C. Knight, and N. G. Leveson. Analysis of faults in an n-version software experiment. *IEEE Trans. Softw. Eng.*, 16(2):238–247, 1990.

[5] J.-L. Dufour. Safety computations in integrated circuits. In *VTS '96: Proceedings of the 14th IEEE VLSI Test Symposium (VTS '96)*, page 169, Washington, DC, USA, 1996. IEEE Computer Society.

[6] P. Forin. Vital coded microprocessor principles and application for various transit systems. In *IFAC/IFIP/IFORS Symposium*, pages 79–84, September 1989.

[7] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30:68–74, April 1997.

[8] 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.

[9] Z. T. Kalbarczyk, R. K. Iyer, S. Bagchi, and K. Whisnant. Chameleon: A software infrastructure for adaptive fault tolerance. *IEEE Transactions on Parallel and Distributed Systems*, 10:560–579, June 1999.

[10] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 330–335, December 1997.

[11] J. Levon. Oprofile - a system profiler for linux. Web site: http://oprofile.soruceforge.net/, 2005.

[12] M. Li, D. Goldberg, W. Tao, and Y. Tamir. Fault-tolerant cluster management for reliable high-performance computing. In *13th International Conference on Parallel and Distributed Computing Systems*, 2001.

[13] 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.

[14] 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.

[15] 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.

[16] N. Oh, P. P. Shirvani, and E. J. McCluskey. ED^{4}I: Error detection by diverse data and duplicated instructions. In *IEEE Transactions on Computers*, volume 51, pages 180 – 199, February 2002.

[17] 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.

[18] W. W. Peterson and M. O. Rabin. On codes for checking logical operations. *IBM Journal of Research and Development*, 3(2):163, 1959.

[19] R. Phelan. Addressing soft errors in ARM core-based SoC. ARM White Paper, December 2003.

[20] M. Rebaudengo, M. S. Reorda, and M. Violante. An accurate analysis of the effects of soft errors in the instruction and data caches of a pipelined microprocessor. In *DATE '03: Proceedings of the Conference on Design, Automation and Test in Europe*, page 10602, Washington, DC, USA, 2003. IEEE Computer Society.

[21] M. Rebaudengo, M. S. Reorda, M. Violante, and M. Torchiano. A source-to-source compiler for generating dependable software. In *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 33–42, 2001.

[22] 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.

[23] 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.

[24] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee. Design and evaluation of hybrid fault-detection systems. In *Proceedings of the 32th Annual International Symposium on Computer Architecture*, pages 148–159, June 2005.

[25] 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.

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

[27] 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.

[28] M. Tremblay and Y. Tamir. Support for fault tolerance in VLSI processors. volume 1, pages 388–392, May 1989.

[29] 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.

[30] 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 Dependable Systems and Networks*, pages 61–72, June 2004.

[31] J. Yan and W. Zhang. Compiler-guided register reliability improvement against soft errors. pages 203–209, 2005.

[32] 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.

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