AUTOMATIC INSTRUCTION-LEVEL SOFTWARE-ONLY RECOVERY

SOFTWARE-ONLY RELIABILITY TECHNIQUES PROTECT AGAINST TRANSIENT FAULTS WITHOUT THE OVERHEAD OF HARDWARE TECHNIQUES. ALTHOUGH EXISTING LOW-LEVEL SOFTWARE-ONLY FAULT-TOLERANCE TECHNIQUES DETECT FAULTS, THEY OFFER NO RECOVERY ASSISTANCE. THIS ARTICLE DESCRIBES THREE AUTOMATIC, INSTRUCTION-LEVEL, SOFTWARE-ONLY RECOVERY TECHNIQUES REPRESENTING DIFFERENT TRADE-OFFS BETWEEN RELIABILITY AND PERFORMANCE.

George A. Reis
Jonathan Chang
David I. August
Princeton University

•••••• Microprocessor performance has been increasing exponentially in recent decades, due in large part to smaller and faster transistors enabled by improved fabrication technology. Although such transistors enhance performance, their lower threshold voltages and tighter noise margins make them less reliable,[1–3] rendering processors that use them more susceptible to transient faults. Although they cause no permanent damage, transient faults can result in incorrect program execution by altering signal transfers or stored values. Such faults have already caused significant failures in commodity processors. In 2000, Sun Microsystems acknowledged that cosmic rays had interfered with cache memories and caused crashes in server systems at dozens of major customer sites, including America Online and eBay.[4] More recently, Hewlett Packard acknowledged that a large installed base of a 1024-CPU server system in Los Alamos National Laboratory has been crashing frequently because of cosmic ray strikes causing transient faults.[5]

Computer architects have typically addressed reliability issues by adding redundant hardware, but these techniques are often too expensive to be widely used. Software-only reliability techniques have shown promise in their ability to protect against soft errors without any hardware overhead (see the sidebar, "Hardware vs. software solutions for fault detection and recovery"). In software solutions, the software can direct the reliability level, reducing costs by enabling fault detection only when necessary. For example, a transient fault in a movie player affecting a single frame of a movie during playback will likely go unnoticed. In this situation, a fault-tolerance technique that negatively impacts the frame rate is much more onerous than a single-frame fault would warrant. For this scenario, a lightweight technique might be most appropriate. However, if instead of a rendering a frame in a movie, the user were rendering a single high-quality image for use in print, a fault in the image would be unacceptable. This situation might warrant a more reliable option. Regardless of the specific reliability implementation, the reconfigurability created by increasing software reliability is extremely valuable to system designers.

To truly be reliable, a system must be able not only to detect faults, but also to recover. Until now, all proposed low-level software-only techniques that we're aware of have addressed only fault detection, not fault recovery. Although this prevents faults from corrupting data, it doesn't let the application correctly run to completion in the presence of a fault.

We present three software-only recovery techniques at the compiler level that offer varying levels of protection with different costs:

- *Swift-R* intertwines three copies of a program and adds majority voting before critical instructions, offering near-perfect reliability for applications that require it.
- *Triple Redundancy Using Multiplication Protection* (Trump) intertwines the original program with an *AN*-encoded version of the program.
- *Mask*, a more lightweight technique than Swift-R or Trump, dynamically enforces invariants that can be proved true statically.

We implemented these techniques in a compiler and evaluated them in isolation, as well as in hybrid combinations. Our results show that these techniques offer a wide spectrum of viable options for fault tolerance that designers can use to increase reliability with reasonable performance costs and without having to design or deploy new hardware.

Fault model

Throughout this work, we assume the commonly used single-event upset fault model. In the SEU model, exactly one bit-flip in one state element will occur throughout a particular execution of the program. Our techniques also tolerate a wide variety of multibit errors, although we don't quantify this effect.

To evaluate a system's reliability, we classify faults according to their effect on the program's final output in the fault's presence. If the fault causes the execution to be abnormally terminated because of a segmentation violation, we categorize the fault as SEGV. If the program completes execution, but doesn't produce correct output, we categorize the fault as a silent data corruption (SDC). Finally, if the program completes execution and the output is correct, we categorize the fault as unnecessary for architecturally correct execution (unACE).[6]

Hardware vs. software solutions for fault detection and recovery

Designers frequently introduce redundant hardware[1–3] to detect or recover from transient faults. For example, storage structures, such as caches and memory, typically include extra information in the form of parity or error-correcting codes (ECC), which let these hardware structures detect and recover from such faults. However, protecting all transistors, particularly those used in combinational logic, is difficult without significant area, power, and performance penalties. Researchers have also proposed higher-level techniques, such as lock-stepping or redundant multithreading,[4] for full-processor fault detection; however, these techniques still require moderate to significant changes to the hardware design, and possibly even to the operating system, significantly increasing validation time and often incurring performance penalties.

Software-only approaches to fault detection[5–7] and recovery can significantly improve reliability without requiring hardware modifications, and are therefore cheaper and easier to deploy. Because hardware-only techniques are too expensive to use in the field, researchers compensate with software-only approaches. Deployment of redundancy techniques in the field is important because designers might incorrectly estimate the soft-error rate or the machine's usage condition might change. Changes to the hardware's operating environment can noticeably affect reliability and require the deployment of software redundancy techniques. For example, the soft-error rate from atmospheric neutrons is 4 to 5 times higher in Denver than in New York City because of Denver's higher altitude.[8]

References

1. R. W. Horst, R.L. Harris, and R.L. Jardine, ''Multiple Instruction Issue in the NonStop Cyclone Processor,'' *Proc. 17th Int'l Symp. Computer Architecture*, ACM Press, 1990, pp. 216–226.
2. T.J. Slegel et al., ''IBM's S/390 G5 Microprocessor Design,'' *IEEE Micro*, vol. 19, no. 2, Mar. 1999, pp. 12–23.
3. Y. Yeh, ''Triple-Triple Redundant 777 Primary Flight Computer,'' *Proc. 1996 IEEE Aerospace Applications Conf.*, vol. 1, IEEE Press, 1996, pp. 293–307.
4. S.K. Reinhardt and S.S. Mukherjee, ''Transient Fault Detection via Simultaneous Multithreading,'' *Proc. 27th Ann. Int'l Symp. Computer Architecture*, ACM Press, 2000, pp. 25–36.
5. N. Oh, P.P. Shirvani, and E.J. McCluskey, ''Error Detection by Duplicated Instructions in Super-Scalar Processors,'' *IEEE Trans. Reliability*, vol. 51, no. 1, Mar. 2002, pp. 63–75.
6. G.A. Reis et al., ''Swift: Software Implemented Fault Tolerance,'' *Proc. 3rd Int'l Symp. Code Generation and Optimization*, IEEE Press, 2005, 242–254.
7. R. Venkatasubramanian, J.P. Hayes, and B.T. Murray, ''Low-Cost On-line Fault Detection Using Control Flow Assertions,'' *Proc. 9th IEEE Int'l Online Testing Symp.*, IEEE CS Press, 2003, pp. 137–143.
8. J.F. Ziegler and H. Puchner, *SER–History, Trends, and Challenges: A Guide for Designing with Memory ICs*, Cypress Semiconductor Corp., 2004.

```
ld r3 = [r4]

add r1 = r2, r3



st[r1] = r2
```
(a)

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

Figure 1. Swift duplication and validation: original code (a) and Swift code (b).

We refer to a system's reliability as the percentage of faults that are unACE, because SEGV and SDC faults are both deleterious.

Swift

The Swift-enabled compiler duplicates a program's instructions and schedules them along with the original instructions in the same execution thread.[7] The original and duplicate versions of the instructions are register-allocated so they don't interfere with each other. At certain synchronization points in the combined program, the compiler inserts validation code to ensure that the data produced by the original and redundant instructions are equal.

Because program correctness is defined by the program's output, we insert validation checks before any instruction that might potentially generate output. There are two principal methods for user-level code to produce output: memory-mapped I/O and system calls.

If a program produces all output via memory-mapped I/O, it has executed correctly if all of its loads and stores 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 take place. The lack of software redundancy in memory doesn't often significantly impact reliability, because error-correcting code (ECC) typically protects memory and caches against transient faults.

Figure 1 shows a sample code sequence before and after the Swift fault-detection transformation. The Swift-enabled compiler duplicates the add instruction and inserts it 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′.

The compiler inserts instructions 1 and 2 to validate and replicate the load instruction data. Instruction 1 ensures that the subsequent load's address matches its duplicate address. Furthermore, the compiler doesn't insert a redundant load instruction because the load might be uncacheable, so, to set r3′, the technique must find an alternative to redundantly executing the load. In this case, instruction 2 simply copies the load instruction's result into its duplicate register.

We use r1 and r2 values at the store instruction at the end of the example. Because we must avoid storing incorrect values into memory and storing values to incorrect addresses, the technique checks that both the address and value match their redundant copy. If it detects a difference, a fault has occurred, and it notifies the system via instruction 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) can schedule the instructions to use additional available instruction-level parallelism (ILP), thus minimizing the transformation's performance penalty.

Checking at loads and stores will protect against many faults. However, programs can also generate output via system calls, or more generally, via external libraries. Because external code might not have any protection, the best Swift can do is verify that all of the inputs to the function or system call are correct. 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—won't protect against faults that affect branch outcomes. If a fault occurs on a data slice that only feeds a branch, the application might take an incorrect execution path and execute incorrect loads and stores, although no fault will be detected. To protect against this, Swift also verifies the input registers to any branch predicate.

Figure 2 demonstrates this protection. 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 similar to a load instruction not only in that the inputs must be checked, but also that they can't be safely duplicated. So, to produce a redundant copy of the return value, here given as R0, the compiler must insert instruction 2.

Swift, like all software-only reliability techniques, has some vulnerabilities. Previous work has described these vulnerabilities in detail and has shown that software-only techniques effectively detect most faults in many parts of the system.[7,8]

Swift-R

The Swift transformation can be seen as a double-modular redundancy implemented in software. Double redundancy provides detection but not recovery. Swift-R achieves reliability with recovery by using triple-modular redundancy.

Instead of creating one redundant copy, as in Swift, the Swift-R transformation 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, the system can correct any single-bit fault.

Figure 3 shows the Swift code from Figure 1 in Swift-R. 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 that 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).

We also replaced Swift's fault-detection code with recovery code at instructions 1, 6,

```
                                   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)                                (b)
```

Figure 2. Swift branch and function-call validation: original code (a) and Swift code (b).

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″, doesn't, we set r1″ to the value in r1 and r1′, correcting the corrupted value of r1″.

Trump

AN-codes form the theoretical backdrop for the Trump technique. They allow a more compact representation of redundancy, ultimately letting Trump contain Swift-R's redundant data in two registers instead of three. Although Trump's *AN*-encoding is less general than Swift-R's triple-modular redundancy, rendering it unable to protect certain portions of programs, Trump's redundant computation is much less onerous. Thus, it provides an alternative for applications that can't afford Swift-R's performance penalty, but could benefit from moderate protection.

AN-codes

AN-codes are a class of arithmetic codes (codes that are preserved across arithmetic expressions) in which the code word is

```
                                   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)                                (b)
```

Figure 3. Swift-R triplication and validation: original code (a) and Swift-R code (b).

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

if ($3x \neq y$)
     if ($y \equiv 0 \pmod 3$)
          $x = y/3$
     else
          $y = 3x$
```

Figure 4. Trump recovery pseudocode.

simply the original data multiplied by a constant, *A*. The fact that *AN*-codes are arithmetic codes is evident using standard algebra:

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

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

You can use *AN*-codes to detect errors by verifying that the code word is divisible by *A*. Precisely, *C* is a valid code word only if $C \equiv 0 \pmod A$. The choice of *A* significantly impacts the implementation cost as well as the resulting code's reliability. $A = 2^n - 1$ is a particularly good choice with respect to both of these.

First, consider the reliability ramifications of this choice. You can consider any single-bit fault to a code word as either an addition or subtraction of 2^k for some *k*. Observe that $2^k \neq 2^n - 1$ for any $n > 0$. So,

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

This proof guarantees that the faulty *AN* code word won't be divisible by *A*, so *A* will be able to detect any single-bit faults to the code word. Although we don't prove it here, this choice of code word can also protect against numerous multibit faults.

$A = 2^n - 1$ is also a convenient implementation for performance because you can compute multiplication by *A* as simply a shift left by *n* and a subtraction, specifically, $Ax = (x \ll n) - x$.

The choice of *A* also determines how many bits you'll need to represent the code word. For $A = 2^n - 1$, you'll need *n* extra bits to represent the code word. 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.

Trump transformation

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

Trump essentially has two copies of every value, similar 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* doesn't match the *AN*-encoded copy. If they don't match, Trump can recover the code by inferring which copy is correct. If the *AN*-encoded copy is divisible by *A*, we can surmise that the fault struck the original copy. If it isn't, the *AN*-encoded copy was struck and the original is correct. Figure 4 shows pseudocode for this recovery sequence.

Although this might be costly because of the division and modulo operations, Trump executes these instructions only during fault recovery, which is relatively rare.

Figure 5 illustrates the Trump transformation. We denote the redundant Trump registers by appending a t to the register's name. Although we show code with multiplications here for brevity, we implement multiplications with the faster combination of shifts and adds.

```
                                1:   call recovery, 3*r4 != r4t
ld r3 = [r4]                         ld r3 = [r4]
                                2:   mul r3t = 3, r3
add r1 = r2, r3                      add r1 = r2, r3
                                3:   add r1t = r2t, r3t
                                4:   call recovery, 3*r1 != r1t
                                5:   call recovery, 3*r2 != r2t
st[r1] = r2                          st [r1] = r2
(a)                             (b)
```

Figure 5. Trump transformation: original code (a) and Trump code (b).

As implemented in previous software-only reliability techniques, the Trump-enabled compiler must validate the load address before load instructions. Instruction 1 performs this check by ensuring that three times the original value is equal to the redundant value. If a mismatch occurs, the system will execute the recovery code shown in Figure 4. Similarly, before the store instruction, instructions 4 and 5 check the store operands.

Also similarly to previous software-only reliability techniques, the compiler must copy the load instruction result into the redundant register, as in instruction 2. In Trump, instead of a simple move, we perform a multiplication 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 code words are preserved through arithmetic operations. Therefore, we don't need to alter this instruction in any way from the Swift version.

Thus, Trump offers recovery similar to Swift-R, but only requires two independent versions.

Applicability

In addition to the vulnerabilities of all software-only recovery schemes, Trump has two primary limitations.

First, *AN*-codes don't propagate through many logical operations, such as and and or,[9] and therefore can't be applied to certain dependence chains.

Second, a register can never assume a value greater than $2^M/A$, where M is the number of bits in that register. If it does, the *AN*-encoded version of the register will overflow. To avoid this situation, we only apply Trump on dependence chains whose values never exceed $2^M/A$. If the compiler can't statically prove that a certain dependence chain has this property, it must 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, typically don't use many bits when executed on 64-bit architectures. These two phenomena make Trump applicable on most applications.

```
mov r3 = 0

Loop:
…

call otherFunc, r3 != 0
xor r3 = r3, 1
br Loop
```
(a)

```
mov r3 = 0

Loop:
…
and r3 = r3, 1
call otherFunc, r3 != 0
xor r3 = r3, 1
br Loop
```
(b)

Figure 6. Mask transformation: original code (a) and Mask code (b).

Mask

Mask is a very low-cost reliability technique. It enforces statically known invariants 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 an example. Consider the code in Figure 6, loosely culled from adpcmdec, an adaptive pulse-code modulation (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 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, 63/64 of the faults will be of this latter type while only 1/64 of the faults will be of the first type. Mask attempts to resolve the latter and more common type of fault.

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. Faults occurring to any of the bits that should be zero will be masked out and won't affect any subsequent computation.

This will increase register r3's reliability by a factor of 64.

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 can't cause the system to produce incorrect output. Although we only evaluate masking with and instructions to enforce known-zero bits, we could easily extend the technique to use or instructions to enforce known-one bits, or sign-extensions to enforce known-sign bits. We could also eventually extend the technique to account for higher-level semantic information and programmer annotations.

Hybrid techniques

In our evaluation, we also consider four hybrid combinations of Swift-R, Trump, and Mask. The Trump/Swift-R hybrid technique uses Trump protection when applicable and Swift-R for those instructions left uncovered by Trump, such as bit-manipulation operations. Similarly, the Trump/Mask hybrid technique protects with the Trump technique when appropriate and applies Mask on the remaining unprotected instructions. These are often exclusive, because it's typically difficult to prove that any of the bits in the instructions that Trump can protect—namely arithmetic operations—are zero, whereas it's usually much easier to prove that bits are zero in instructions that Trump can't protect, such as logical and and or.

We don't evaluate the Swift-R and Mask combination because it would simply consist of full Swift-R protection with additional Mask instructions inserted. However, because Mask provides only a strict subset of Swift-R's protection, the Mask instructions would provide no reliability benefits. For the same reason, we also don't evaluate the Trump/Swift-R/Mask hybrid.

Evaluation

To evaluate the techniques, we implemented each of them as a pass in the gcc compiler, version 3.4.1, targeted for the PowerPC 970. Our additional compilation phase occurs in the compiler's back end immediately before register allocation and scheduling. We evaluated the techniques on a variety of benchmarks taken from SPEC CPU2000, MediaBench, 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.

Reliability

We performed fault-injection experiments to evaluate the techniques' reliability. In accordance with the SEU model, we inserted exactly one fault per execution. We inserted the fault into a uniformly randomly selected bit in a uniformly randomly selected integer register at a uniformly random dynamic instruction in the program's execution. We performed 250 such runs for each benchmark for each technique and recorded each run's outcome. We injected faults into the register file because, as researchers have shown, it's a leading contributor of soft errors.[11] The proposed techniques also protect against most errors to other structures, such as the arithmetic logic unit, which are expensive to protect with ECC, because errors to these structures will often manifest themselves similarly to register file faults.

Figure 7 shows the reliability evaluation's results. The percentage of unACE bits in the baseline code with no added fault tolerance (NOFT) is quite high at 74.18 percent, demonstrating that the unprotected code already contains numerous dynamically dead registers and masked bits. The SEGV percentage for NOFT is 18.00 percent, much higher than the SDC percentage, at 7.82 percent. This indicates that faults in registers are much more likely to cause segmentation violations than to corrupt data, suggesting that a great deal of computation for most benchmarks feeds the addresses of memory accesses rather than the data itself.

As expected, Swift-R, with its triple-modular redundancy, greatly reduces the SEGV and SDC—to 1.93 percent and 0.81 percent, respectively. Furthermore, it is consistently low across all benchmarks, indicating universal applicability. The amount of SEGV and SDC is still nonzero,

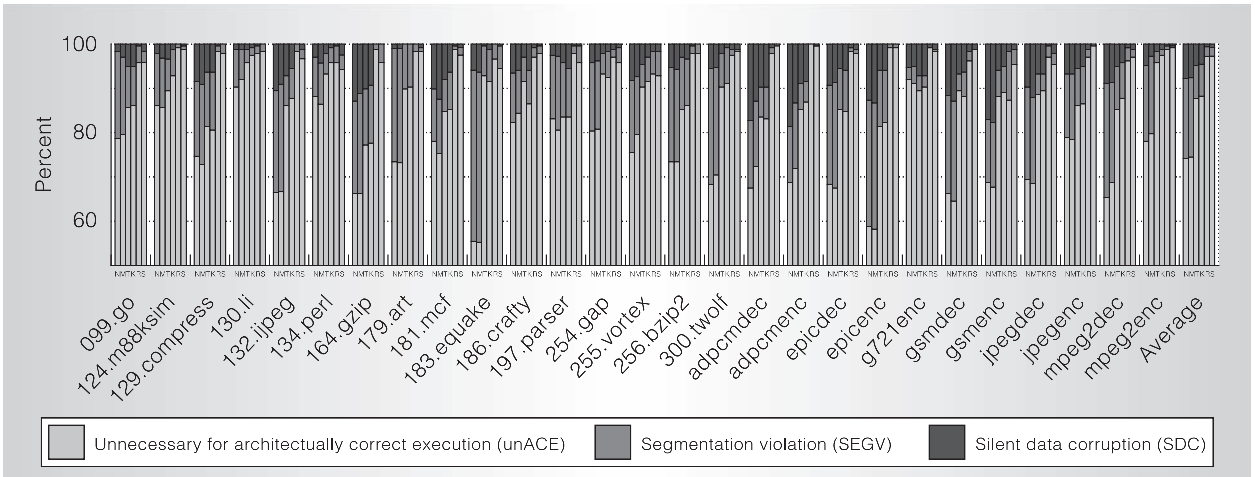

Figure 7. Reliability percentage for NOFT (N), Mask (M), Trump (T), Trump/Mask (K), Trump/Swift-R (R), and Swift-R (S). The average is across all benchmarks.

however, due to the windows of vulnerability.[7,8] Because we didn't specifically direct the compiler to schedule for reliability, we could further improve the reliability by forcing the compiler to move the checks as close as possible to the uses, possibly with some performance cost.

Trump also significantly improves reliability over NOFT, albeit not as much as Swift-R. Trump reduces the SEGV to 7.39 percent and the SDC to 4.88 percent while increasing the unACE percentage to 87.73 percent. Trump improves SEGV much more dramatically than SDC because Trump can protect most pointer dependence chains but few other types of dependence chains. There are two principal reasons for this:

- Pointer ranges are limited to valid memory addresses, so it's easier to verify that the *AN*-encoded values won't overflow.
- Pointer computations tend to be restricted to simple arithmetic operations such as addition, which Trump can protect.

As we'll demonstrate in the next section, the performance penalty incurred by Trump is significantly less than that of Swift-R. This characteristic, coupled with its reliability, makes Trump a promising middle ground for designers who can't afford Swift-R's performance penalty but who still need significant reliability enhancement. However, designers must keep in mind that Trump doesn't increase reliability uniformly across all benchmarks. For benchmarks that are dominated by arithmetic instructions that Trump can protect, 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 can't protect, such as logical operations, Trump's reliability is significantly lower than Swift-R's.

The Mask technique doesn't significantly reduce SDC (7.61 percent versus NOFT's 7.82 percent) or SEGV (17.89 percent versus NOFT's 17.89 percent) across all benchmarks. In fact, in some benchmarks, Mask's reliability can be slightly worse than NOFT's, because of poorer schedules in terms of reliability. However, in other benchmarks, such as adpcmdec or mpeg2dec, the Mask technique makes a significant difference. In adpcmdec, it lowers the SDC from 17.30 percent to 12.87 percent, and in mpeg2dec, it lowers the SEGV from 25.74 percent to 22.57 percent. This suggests that by exploiting additional program invariants, the Mask technique could enhance reliability with practically no cost.

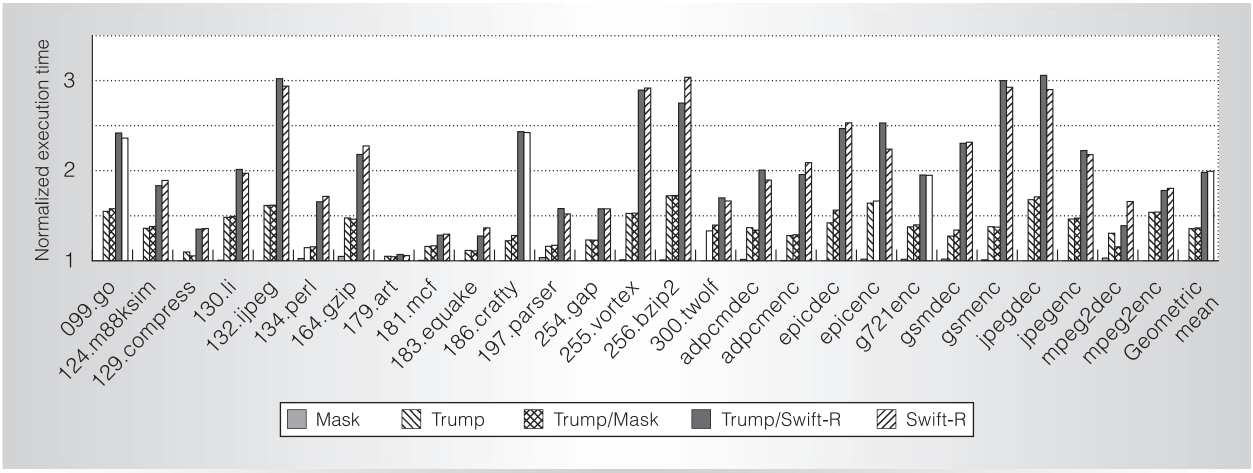

Figure 8. Execution time of Mask, Trump, Trump/Mask, Trump/Swift-R, and Swift-R normalized to NOFT. The geometric mean is across all benchmarks.

As you'd expect, 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, the Trump/Mask combination fares significantly better than either Trump or Mask individually. In fact, Trump/Mask reduces adpcmdec's SDC to 4.55 percent compared with 4.88 percent for Trump and 7.61 percent for Mask. We attribute this additive effect to the fact that Mask and Trump protect different types of instructions. Trump protects arithmetic instructions whereas Mask protects instructions in which bits can be proved to be zero, which are almost always logical instructions.

Finally, the Trump/Swift-R technique performs similarly to Swift-R, with a SEGV of 2.14 percent and a SDC of 0.62 percent. This implies that the Swift-R portions of the code are successfully filling Trump's protection gaps, leaving windows of vulnerability on par with those in Swift-R. However, Trump/Swift-R's reliability is slightly worse than Swift-R's for some benchmarks because adding the Trump instructions can increase the total dynamic number of instructions. This is because transitions between Swift-R and Trump require extra instructions, and Trump's verification sequence is longer than Swift-R's. This can ultimately increase register live ranges and vulnerability window size. Thus, 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 require additional investigation, which we're pursuing as future work.

Performance

We collected performance results for each technique using Linux's OProfile when no faults were injected. Figure 8 shows the execution times for each of our techniques normalized to a baseline build with NOFT. Note that the bars are clipped at one. In most cases, Mask's performance is only nominally above one, and in some cases, Mask's performance bests NOFT's because the inserted instructions cause slight changes to the scheduling and register allocation heuristics. Consequently, Mask bars appear to be missing for many benchmarks.

Our techniques exhibit a wide range of performance behaviors. The low-cost techniques, Trump and Mask, have normalized execution times of only 1.36 and 1.00, respectively. The Trump/Mask technique 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 Swift-R's than Trump's. This implies that Trump/Swift-R's protection choices track more closely with Swift-R than with Trump—that is, Swift-R protects many more instructions than Trump does. This concurs with the reliability evaluation, which showed that Trump/Swift-R's reliability is much closer to Swift-R's than to Trump's.

Much like its reliability, Trump/Swift-R's performance highly depends on the trade-offs between Swift-R protection and Trump protection. The Swift-R technique is more expensive than Trump in terms of redundancy because it requires two additional versions of the computation instead of one. Trump, on the other hand, is more expensive in terms of verification because it must convert the *AN*-encoded and original data to the same form for comparison. Depending on the ratio of redundant computation to comparison, a Trump dependence chain might actually be more costly than a Swift-R dependence chain, which accounts for Swift-R occasionally outperforming Trump/Swift-R.

Trump/Mask typically performs much better than either Trump/Swift-R or Swift-R, but significantly worse than Mask and on par with Trump. This is to be expected because Mask's performance impact is nearly negligible. The performance is slightly worse than the simple sum of Mask and Trump, because each technique alone can use some of the previously unused resources, but the processor doesn't have enough unused resources to support both the Mask and Trump protections, thus creating a superadditive performance penalty. In some cases, most notably mpeg2dec, Trump/Mask outperforms Trump, just as Mask occasionally outperforms NOFT. Once again, this is because of changes in the scheduler and register allocator that result from inserting extra instructions.

The normalized execution time of all of our techniques, even Swift-R, averages far less than 3, the time one might naively expect after triplicating the code. In benchmarks dominated by floating-point instructions that we don't protect, such as 179.art, we'd 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 3 for most integer benchmarks. All of our techniques exploit the well-documented existence of unused ILP resources in most modern processors. Because most of the instructions added in Swift-R and Trump are independent of the original instructions, the reliable code can typically use previously unused ILP resources. This effect is especially visible in benchmarks that already exhibit poor ILP in NOFT, such as 181.mcf. The 181.mcf benchmark spends a large fraction of its time in memory stalls; consequently, our transformations have a 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 ILP effect, 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 checks are needed, the reliable compiler inserts another more complex sequence of instructions. Although the verification code differs for each technique, in benchmarks with many checks, such as 255.vortex (due to a preponderance of loads), the performance impact is typically much higher than benchmarks, such as 300.twolf, with fewer checks and more time dedicated to pure computation.

In summary, Swift-R, which has a normalized execution time of 1.99, can significantly improve reliability, increasing unACE to 97.27 percent. You should use this technique when high reliability requirements warrant this level of performance degradation. Moving to the hybrid technique Trump/Swift-R slightly improves the performance of Swift-R.

When the system's reliability requirements aren't stringent enough to warrant Swift-R or Trump/Swift-R, you can use Trump or Trump/Mask. Trump has the much lower normalized runtime of 1.36, but still manages to increase the unACE to 87.73 percent versus 74.18 percent for NOFT. Trump/Mask improves on this slightly, increasing reliability further while having a negligible impact on performance.

Finally, for systems that can tolerate almost no performance degradation, you can use the Mask technique. Although Mask only sometimes improves reliability, it's essentially free in terms of performance cost, so applying it is almost certainly worthwhile. MICRO

Acknowledgments

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

References

1. R.C. Baumann, ''Soft Errors in Advanced Semiconductor Devices–Part I: The Three Radiation Sources,'' *IEEE Trans. Device and Materials Reliability*, vol. 1, no. 1, Mar. 2001, pp. 17-22.
2. T.J. O'Gorman et al., ''Field Testing for Cosmic Ray Soft Errors in Semiconductor Memories,'' *IBM J. Research and Development,* Jan. 1996, pp. 41-49.
3. P. Shivakumar et al., ''Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic,'' *Proc. 2002 Int'l Conf. Dependable Systems and Networks*, IEEE CS Press, 2002, pp. 389-399.
4. R.C. Baumann, ''Soft Errors in Commercial Semiconductor Technology: Overview and Scaling Trends,'' *IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals,* 2002, pp. 121_01.1-121_01.14.
5. S.E. Michalak et al., ''Predicting the Number of Fatal Soft Errors in Los Alamos National Laboratory's ASC Q Computer,'' *IEEE Trans. Device and Materials Reliability*, vol. 5, no. 3, 2005, pp. 329-335.
6. S.S. Mukherjee et al., ''A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor,'' *Proc. 36th Ann. IEEE/ACM Int'l Symp. Microarchitecture*, IEEE CS Press, 2003, pp. 29-42.
7. G.A. Reis et al., ''Swift: Software Implemented Fault Tolerance,'' *Proc. 3rd Int'l Symp. Code Generation and Optimization*, IEEE Press, 2005, pp. 242-254.
8. G.A. Reis et al., ''Software-Controlled Fault Tolerance,'' *ACM Trans. Architecture and Code Optimization (TACO)*, vol. 2, no. 4, Dec 2005, pp. 366-396.
9. W.W. Peterson and M.O. Rabin, ''On Codes for Checking Logical Operations,'' *IBM J. Research and Development*, vol. 3, no. 2, 1959, p. 163.
10. C. Lee, M. Potkonjak, and W. Mangione-Smith, ''MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems,'' *Proc. 30th Ann. Int'l Symp. Microarchitecture*, IEEE CS Press, 1997, pp. 330-335.
11. N.J. Wang et al., ''Characterizing the Effects of Transient Faults on a High-performance Processor Pipeline,'' *Proc. 2004 Int'l Conf. Dependable Systems and Networks*, IEEE CS Press, 2004, pp. 61-72.

George A. Reis is a fourth-year PhD student in the Department of Electrical Engineering at Princeton University. His research interests include computer reliability and configurable techniques to mitigate processor faults using combinations of software and hardware implementations. Reis has a BSE and an MA in electrical engineering from Princeton University. He is a student member of the ACM.

Jonathan Chang is a fourth-year PhD student in the Department of Electrical Engineering at Princeton University. His research interests include natural language processing and machine learning, as well as computer architecture and fault tolerance. Chang has a BS in electrical and computer engineering from the California Institute of Technology and an MA in electrical engineering from Princeton University. He is a student member of the ACM.

David I. August is an associate professor in the Department of Computer Science at Princeton University, where he directs the Liberty Research Group. The Liberty Research Group is studying next-generation architectures, code analyses, and code transformations to enhance performance, reliability, and security. August has a PhD in electrical engineering from the University of Illinois at Urbana-Champaign. He is a member of the IEEE and ACM.

Direct questions about this article to David August, Department of Computer Science, Princeton University, 35 Olden St., Princeton, NJ 08540; august@princeton.edu.