A Characterization of Instruction-level Error Derating and its Implications for Error Detection

Jeffrey J. Cook[1] Craig Zilles[2]

[1]*Department of Electrical and Computer Engineering* [2]*Department of Computer Science*
University of Illinois at Urbana-Champaign
{*jjcook, zilles*}*@uiuc.edu*

Abstract

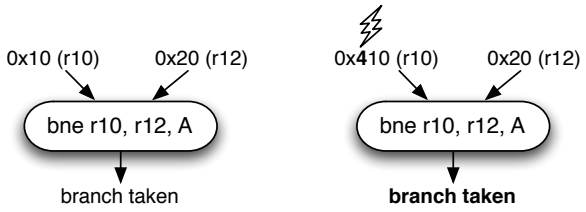
In this work, we characterize a significant source of software derating that we call instruction-level derating. Instruction-level derating encompasses the mechanisms by which computation on incorrect values can result in correct computation. We characterize the instruction-level derating that occurs in the SPEC CPU2000 INT benchmarks, classifying it (by source) into six categories: value comparison*,* sub-word operations*,* logical operations*,* overflow/precision*,* lucky loads*, and* dynamically-dead values*. We also characterize the temporal nature of this derating, demonstrating that the effects of a fault persist in architectural state long after the last time they are referenced. Finally, we demonstrate how this characterization can be used to avoid unnecessary error recoveries (when a fault will be masked by software anyway) in the context of a dual modular redundant (DMR) architecture.*

Keywords: Dual modular redundancy, error detection, fault injection, instruction-level derating, software derating.

1. Introduction

Transient faults are an important concern in modern microprocessor design. As we continue to scale transistors to smaller dimensions and pack wires closer together, they become increasingly susceptible to transient faults due to a number of factors, including radiation [21], crosstalk [10], parameter variation [5] and transistor wear-out [23]. Mechanisms to tolerate transient faults continue to be an active area of research.

Recently, in light of the power constraint and the end of the exponential growth in processor frequencies, there has been a trend to unify reliability and performance/power optimization [3, 9, 12, 17, 24]. With the goal of maximizing clock frequency or minimizing power consumption, these approaches eschew the timing and voltage safety margins traditionally used to make systems robust to faults; a robust error detection and correction mechanism is used to ensure correct execution in the presence of the relatively frequent occurance of transient faults (*e.g.*, 1 per 10^4-10^6 clock cycles) that occur when a design is just barely making timing.



(a) *error-free execution* (b) *execution with incorrect values*

Figure 1: **An example of instruction-level derating: value comparison.** Even with a bit flipped in `r10`, the value comparison in (b) has the same outcome as the error-free execution in (a).

One well documented aspect of transient faults is that only a fraction that occur manifest themselves as errors; the remaining faults are *derated* at various levels of the hardware-software hierarchy. At the circuit and micro-architectural level, previous work has both measured the fraction of circuit-level fault injections that manifest in architectural state [20, 28], as well as proposing an intuition for why *micro-architectural derating* occurs in the form of the Architectural Vulnerability Factor (AVF) metric [19], which is derived in part by the fraction of time that various state elements contain/contribute to producing architectural state.

Researchers have also explored *software derating* by injecting errors into architectural state and observing how many result in incorrect execution [4, 13–15, 25, 26, 28]. In this way, the derating effect of software has been estimated but there has been little work to understand these mechanisms, perhaps in part because the mechanisms were viewed to be application specific.

In particular, no work that we are aware of has characterized the sources of instruction-level derating of

architecturally-visible faults, that is the mechanisms by which an instruction can compute using incorrect data and still produce correct results. An example of this *instruction-level derating* is shown in Figure 1. The main goal of this paper is to provide a characterization of instruction-level derating, considering both its mechanisms and temporal behavior.

In our characterization in Section 4, we demonstrate the following:

1. we demonstrate that there are six major mechanisms leading to instruction-level derating,
2. we find in our experiments that, despite taking a rather conservative view of the opportunity for instruction-level derating, 36% of fault injections into architectural state are derated,
3. we show that roughly half of the derated faults propagate to other instructions before they are masked, and
4. we show that, even when a fault does not affect the program's outcome, incorrect temporary values can persist in architectural registers significantly past the last use of any incorrect value.

The practical relevance of these results is that they indicate *error detection schemes can be designed to avoid reporting self-correcting architecturally-exposed faults*, thereby avoiding the performance and power penalties of recovery for systems where recoveries are not uncommon. We first quantitatively demonstrate the behavior difference between error detection schemes (in Section 5): systems that check architectural state at every instruction boundary will always report architecturally-exposed faults as errors, eliminating any opportunity for instruction-level derating, whereas a periodic check will not. We continue by demonstrating an optimization, which exploits the temporal behavior of instruction-level derating, to further reduce the number of reported self-correcting errors by not checking registers known by the compiler to contain dead values.

Before the characterization of instruction-level derating, we first provide background/terminology on faults, errors, and their derating that provides the context for this paper (Section 2) and a description of the experimental method employed in these studies (Section 3). We conclude in Section 6 with a discussion of other potential opportunities for exploiting knowledge of these derating mechanisms.

2. Background

In this section, we discuss the context for our study on instruction-level derating. As illustrated in Figure 2, faults occur at the circuit level but must reach the application level by changing the program's output or causing an unrecoverable exception (*e.g.*, a memory protection fault) to actually

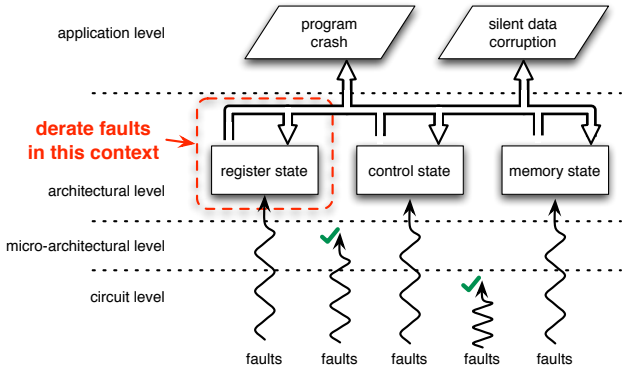


Figure 2: **Scope of instruction-level derating.** Error derating can occur at many levels; in this work, we focus on the software derating of faults that manifest at the architectural level, before they corrupt control flow or memory state or propagate to the application-level to cause a crash or silent data corruption.

be considered as an error. Along the way, there are many points at which faults can be *derated* (*i.e.*, used without producing faulty output) and *masked* (*i.e.*, eliminating faulty state) before they cause an error.

Faults originate at the circuit level, either altering the result of combinational logic or flipping latched state [21]. If left uncorrected by circuit- or logic-level techniques [18], faults propagate to the micro-architectural level which includes potential architectural state. At the micro-architectural level, it has been shown that many injected faults fail to reach architectural state. Two main mechanisms lead to this result: 1) many bits of micro-architectural state are *dead*, in that they will be written before being referenced, and 2) some micro-architectural state affects performance but not correctness, with predictor state being the obvious example. Previous work has characterized the relative error vulnerability of micro-architectural structures by observing what fraction of their bits are necessary for an *architecturally correct execution* (ACE, where architecturally-visible state never contains an incorrect value) [27] and by injecting micro-architectural faults and observing which fraction lead to program crashes and incorrect program outputs [20, 28].

There are discrepancies in the numbers computed by these two approaches because an architecturally correct execution is not explicitly required to compute the correct program output without program crashes; some derating can occur in the software itself. Previous work has measured the rates of software derating [4, 13–15, 25, 28], but little work has demonstrated the mechanisms involved in software derating. One notable exception, is the work of Wang *et al.* that characterized *y-branches*: branches whose outcome can be reversed without changing the program's

outcome [26]; the only permanent change is in terms of the program's execution time. Y-branches typically result from the structure of the program or its control-flow graph (*e.g.*, taking early exits from loops that end up not having a side effect, and if statements based on multiple predicates where one predicate determines the overall control flow even if another one is computed incorrectly). In addition, some architectural state (like microarchitectural state) is dead and, therefore, will not affect the program's final results because it is never read by future instructions.

These sources of derating, however, are difficult to exploit if our goal is providing high reliability. While previous work on software derating shows that a non-trivial number of architectural-level faults are masked, it provides little consolation for the faults that lead to damaging situations like silent data corruption. Generally, systems that provide fault tolerance check control flow and values before they are released to the memory system, because doing so simplifies the checking and reduces the recovery effort, respectively. For example, a commonly proposed approach for a bandwidth efficient implementation of checking in dual modular redundancy (DMR) is to compare three pieces of information between the processors: the stream of branch outcomes, a hash of the register values, and the store addresses and values [11].

For these reasons, we focus in this paper on the instruction-level mechanisms of software error derating, where faults potentially propagate through register state, but are derated/masked before they affect control flow or are exposed to the memory system. These sources of derating, which occur within the dashed region in Figure 2, do not rely on understanding the program's control flow structure or its memory access behavior to allow such faults to be naturally masked without risking a silent data corruption. In doing so, we consider an error model (described in Section 3.3) that abstracts but closely resembles those proposed for modern DMR systems.

3. Experimental Method

To study instruction-level derating, we performed a series of fault-injection experiments. Because we were concerned with software derating, these experiments were performed with a functional simulator that only models the architectural behavior of the machine. In this section, we describe our simulation infrastructure (Section 3.1), how we performed fault injection (Section 3.2), and the error model that we used to decide whether an injected fault was derated or not (Section 3.3).

3.1. Experimental Framework

We used a functional simulator derived from the SimpleScalar tool set [2] that models the user-level architectural state and ISA of the Alpha AXP architecture; system calls are emulated. In this model, no micro-architectural or timing modeling is performed because it is not necessary to observe software error derating.

Our experiments were performed using the SPEC CINT2000 benchmark suite, running each benchmark with its full reference inputs. To ensure that the behaviors that we observed were not merely due to the idiosyncrasies of a particular compiler, we performed our experiments using two compilers and three optimization levels: fully optimized OSF binaries, and binaries generated with gcc with no (-O0), standard (-O2), and aggressive (-O3) optimizations. We used the OSF binaries that are provided with SimpleScalar which were compiled using the DEC C compiler under OSF/1 V4.0 operating system for peak performance, using at least the -O4 optimization level. The gcc binaries were compiled on Linux using gcc 4.0.2. Only nine of the gcc-compiled benchmarks run to completion; `gap`, `perl`, and `vpr` fail to complete due to unsupported system calls invoked within GNU glibc. In Section 5, we use the LLVM [1] compiler v1.8 in experiments where we add an additional compiler pass to collect live register information.

3.2. Fault Injection

To achieve a representative set of fault injection experiments without having to simulate the whole benchmark, we selected 100 evenly distributed points in the program. At each of these points we perform a series of fault injection experiments in each of the first 100 instructions, giving us 10,000 dynamic instructions to study from each benchmark input. For each of these 10,000 instructions, we perform one fault injection trial for each bit of: the 32-bit instruction word, each of the 64-bit input register values (up to two), and the 64-bit output register value (if any), resulting in up to 224 trials per dynamic instruction. Each trial consists of flipping a single bit, as we assume architectural manifestation of faults to coincide with the results by Cha *et al.* [7]. Bit flips in the instruction change the opcode, register specifiers, and immediate values based on the instruction encoding. Bit flips to register inputs are as if they were errors in reading the value; the copy stored in the register file is unchanged. In contrast, bit flips to the instruction output directly affect the value stored in the register file.

To evaluate a fault injection trial, two parallel executions are run in the functional simulator. One of which, the "golden" execution, simulates execution without any fault present, and thus produces a correct execution to compare against. In the parallel execution, the "tainted" execution is simulated with the injected fault present. After executing each instruction in the trial, register state and control flow from the golden execution are used to compare against the tainted execution to determine if the fault (and possible transitive faults) are masked (a passing trial), fail (due to compulsory events such as memory protection) or due to

	gcc -O3	gcc -O2	gcc -O0	osf subset	avg	osf
pass	35.8%	37.5%	28.4%	42.1%	35.9%	40.0%
inconclusive	1.3%	1.3%	0.5%	1.1%	1.1%	0.9%
fail (error model)	41.0%	39.5%	44.4%	36.6%	40.4%	37.8%
fail (compulsory)	21.9%	21.7%	26.7%	20.2%	22.6%	21.3%

Table 1: **Results of fault injections as a function of optimization level.** Results averaged across benchmark suite.

	gcc -O3	gcc -O2	gcc -O0	osf subset	avg	osf
inst word	28.2%	29.2%	21.3%	32.7%	27.9%	31.5%
input1 val	49.0%	50.0%	40.5%	53.1%	48.1%	51.5%
input2 val	26.5%	28.5%	17.9%	29.6%	25.6%	27.5%
output val	39.6%	41.9%	34.7%	52.1%	42.1%	47.4%

Table 2: **Fraction of fault injections that PASS as a function of the fault injection site.**

the artificial error model (described next in Section 3.3), or is inconclusive (run for 10,000 dynamic instructions without resolution).

3.3. Error Model

We use the following error model, fashioned after a bandwidth efficient checker for a DMR system, to dictate when an trial succeeds/fails. If any of the following invariants are violated, we mark the trial as failing due to the error model.

1. control flow must match
2. store address and value must match
3. load address (alignment) must not fault
4. system call inputs must match

In the context of our hypothetical DMR system, a failure to match on one of these items would force a state rollback to the most recent checkpoint (backwards error recovery). As noted in Section 2, this error model is somewhat conservative, in that there are faults that violate this error model that may not affect the program's outcome. It is our assertion, however, that it would be quite expensive to exploit this additional error derating without exposing the execution to silent data corruption.

In addition to these error model constraints, we define as *compulsory failures* those exceptional behaviors that would represent a program crash in a non-fault tolerant execution (*e.g.*, invalid opcode exception or memory protection fault). We assume these would also invoke backward error recovery and hence terminate any opportunity for a fault to be derated. In our results, we distinguish these two class of failures.

4. Characterization of Instruction-level Error Derating

In this section, we present our observations about the error derating that occurs in software at the instruction level. We begin by presenting the overall rates at which derating occurs and explain how the degree of optimization impacts the derating by software (Section 4.1). We then describe in detail why derating occurs at the software level, demonstrating the instruction-level mechanisms of fault derating (Section 4.2). Finally, in Section 4.3 we characterize the derated errors in the time domain, exploring how quickly after the fault occurs are all traces of it eliminated from the system (*i.e.*, masked). In the following section, we explore an application of this characterization.

4.1. Derating Rates

Averaging across all of the benchmarks and optimization levels, we find that 35.9% of fault injection trials pass without violating the conservative error model defined in the previous section, as shown in Table 1. This promising result means that over one third of faults that are exposed architecturally are derated and masked by software before they affect control flow, propagate to the memory system, or cause an architectural exception. Of the 63.0% of architectural fault injections that fail, roughly one-third are compulsory failures (22.6%) and two-thirds trigger our error model (40.4%). A breakdown of fault injection trial outcomes is described in Table 3 and can be seen in Figure 3.

Interestingly, the unoptimized code experiences about 8% fewer passing fault injections than the optimized versions. This is because register allocation of variables has not been performed, significantly increasing (from 31.8% to 44.5%) the fraction of loads and stores relative to the optimized code. These additional memory instructions have two effects: first, a higher fraction of loads and stores means that an error injection is more likely to hit an address calculation, leading to a compulsory failure (*i.e.*, segmentation fault) or an error model-induced failure (*i.e.*, unaligned access). Second, the rate of error model-induced failures further increases because intermediate values are stored to memory preventing opportunity for those faults

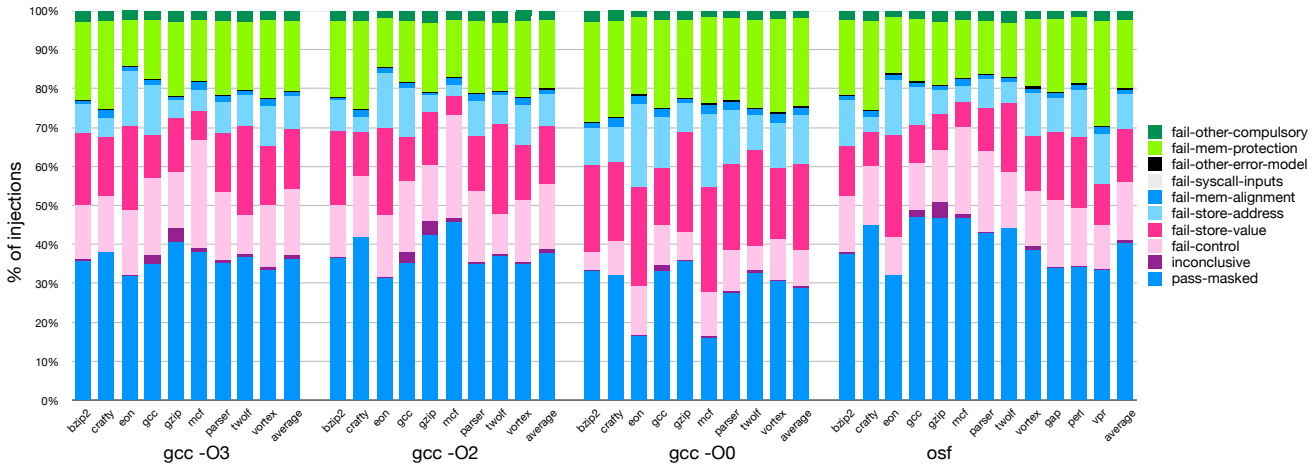


Figure 3: **Detailed fault injection failure-mode classification.** Data presented on a per-benchmark basis. The outcome categories are described in Table 3. On average, 35.9% of fault injections pass; 22.6% fail due compulsory reasons.

RESULT	EXPLANATION	
pass-masked	faulty state masked without violating error model	
inconclusive	trial ran for 10,000 instructions without violating error model or masking faulty state	
fail-control	program counter deviates from golden execution	*error model*
fail-store-value	incorrect store value	*error model*
fail-store-address	incorrect store address (although a valid address)	*error model*
fail-mem-alignment	misaligned load/store address (would cause a PALcode trap fix up)	*error model*
fail-syscall-inputs	incorrect register inputs to syscall	*error model*
fail-other-em	store unexpected/missing, etc. due to instruction word injection	*error model*
fail-mem-protection	invalid load/store address causes memory protection exception	*compulsory*
fail-other-comp	bad opcode, instruction word bit should be zero, etc.	*compulsory*

Table 3: **Description of fault injection trial outcomes.** Failure modes are grouped by category: *error model* or *compulsory*.

to be masked by later instructions. While this effect is visible in most of the benchmarks, it is particularly pronounced in `eon` and `mcf`, as shown in Figure 3; for `eon`, a C++ benchmark, this discrepancy is also likely due to the inherent mechanisms of virtual function calls, as well as the use of helper functions to access object state, as both significantly increase the amount of control flow instructions in unoptimized code.

4.2. Instruction-level Derating

In this subsection, we describe the main mechanisms that lead to instruction-level derating. As shown in Table 2, the instruction-level derating rates on output values and the first input values significantly exceed those of injections into instruction words and second input values. For the instruction word, the lower derating rate is easily attributed to intolerance in changing the opcode or input/output register indices; whereas the second input value is more sensitive than the first input value because it is primarily used for the base address in loads and stores.

We can categorize the sources of instruction-level derating into the six categories shown in Figure 4. The first class, *value comparison*, results from the fact that the information in the values being compared is being reduced down to a single bit, meaning that there is a significant amount of information that is being discarded by the comparison. This category, which accounts for 35% of correct values generated by incorrect inputs, includes stand-alone comparisons (*e.g.*, `cmpeq`) as well as those belonging to branches.

The second class, *sub-word operations*, results from operations that only utilize a fraction of the bits in the incoming values. The example shown in the figure is a byte store, which considers only the bottom eight bits of the register holding the data to store. In addition to sub-word stores, this category (31%) includes derated errors on the upper bits of the shift amount/selection operand for shifts/extracts/inserts, of inserted sub-words, and of 32-bit arithmetic (*e.g.*, `addl`) for this 64-bit machine, as well as

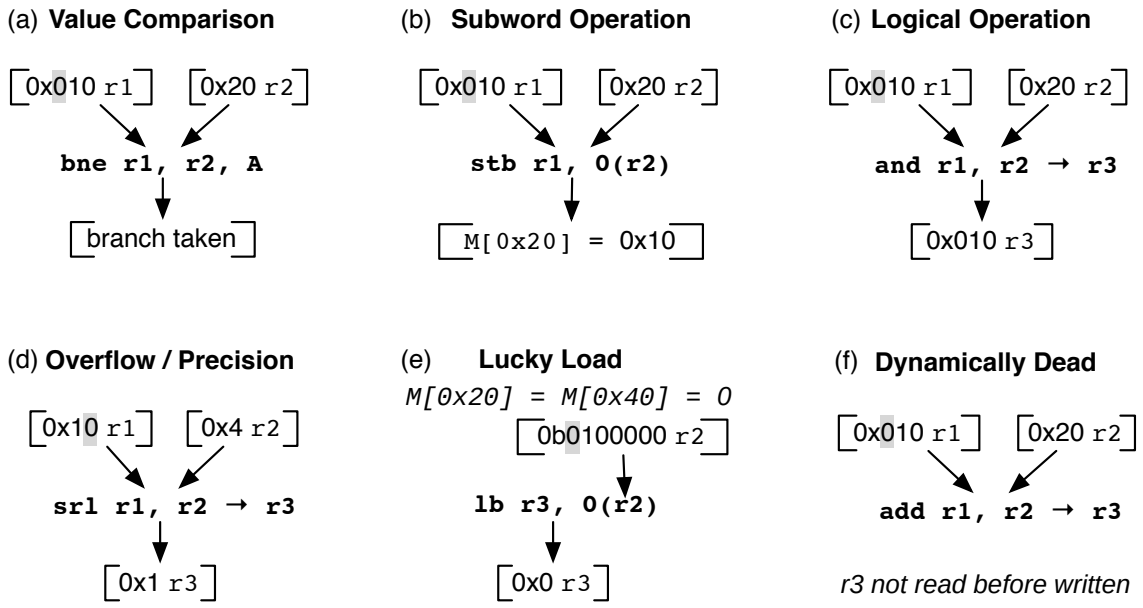


Figure 4: **Examples of the six classes of mechanisms leading to instruction-level error derating.** In each of these examples, the shaded bits of the inputs may be flipped without changing the instruction's outcome.

copy sign, `cpys`, operations which only inspect the sign bit of one of their floating point register inputs.

The third class, *logical operations*, derates errors that occur in AND operations when the corresponding bit in the other operand is 0, as well as OR operations when the corresponding bit in the other operand is 1. In addition to AND and OR operations (*e.g.*, `and` and `bis`), the mask (*e.g.*, `mskqh`) and extract (*e.g.*, `extbl`) operations fall into this category, which represents 24% of the correct values produced by incorrect inputs.

The fourth class, *overflow/precision*, is the source of roughly 5% of correct operations on incorrect values. As shown in Figure 4(d), this class accounts for operations where the faulty bit is shifted off one end of the word, such that it is not part of the output; in the Alpha architecture such shifts occur both in isolation (*e.g.*, `sll` and `srl`) as well as those that are part of scaled adds and subtracts (*e.g.*, `s8addl` and `s4subq`). In addition, we include in this category the small number of cases when erroneous inputs do not affect the output because they either overflow the output register (*e.g.*, if you flip the top bit of one operand in a multiplication, `mull`, and the other operand's LSB is not set) or, for floating point operations, there is no impact due to a lack of precision (*e.g.*, if the LSB of the mantissa of a FP register is flipped and it is added to a much larger number).

The fifth class, representing less than 4% of instructions that take erroneous values and produce correct values, is *lucky loads*. These instructions take an erroneous base address register (for example) and result in a load of the correct value in spite of this incorrect address. As is to be expected, this largely occurs when common values (*e.g.*, zero) are loaded.

In addition to these instructions that generate correct values, a major source of software error tolerance is *dynamically dead* values [6]. These are values that are computed but then not used before another instruction overwrites them. Such values are generally computed for use by a program path that doesn't end up getting taken; they result both from the way that the program is written and from compiler scheduling — especially by the OSF compiler — when an instruction is hoisted above a branch. In the GCC optimized code, about 15% of fault injections that are derated are the result of dynamically dead code; in the OSF optimized code this number increases to almost 30%.

4.3. Fault Propagation and Derating Time-scale

Each injection trial result can be classified not only by whether the trial passed or failed, but also whether the fault propagated beyond the injected instruction to subsequent instructions, as shown in Figure 5. As can be seen, even when faults are masked, it often does not happen immediately. Over half of the faults that are eventually masked propagate to at least one other instruction.

When we measure how far faults propagate, we observe that there are two events that mark the end of a derated (passing) fault injection's lifetime: 1) the point of the last use of any erroneous value (either the fault injected value or a value to which it propagated) and 2) the point at which

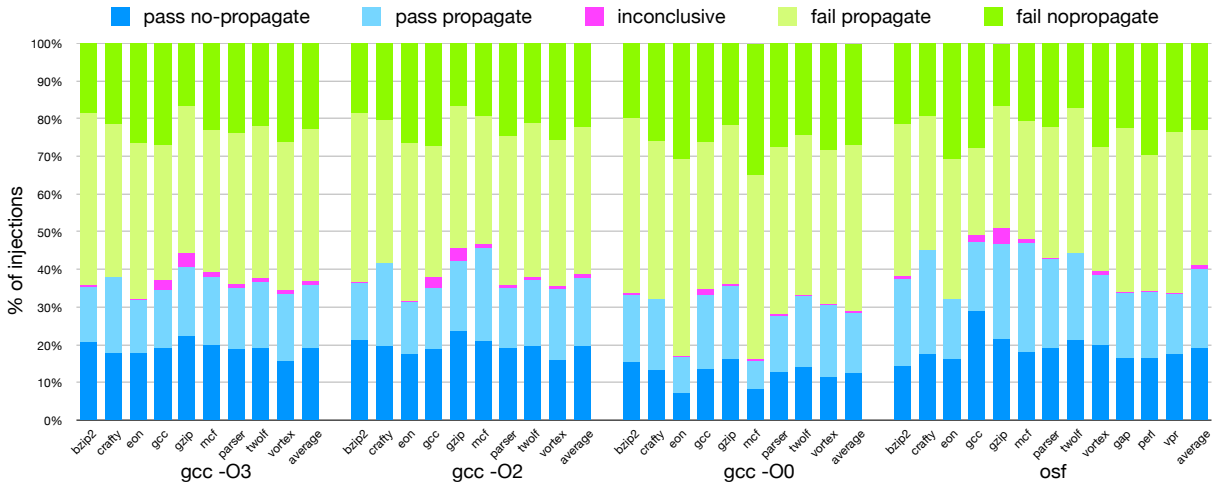


Figure 5: **Detailed fault injection classification.** Broken down by whether the fault propagated.

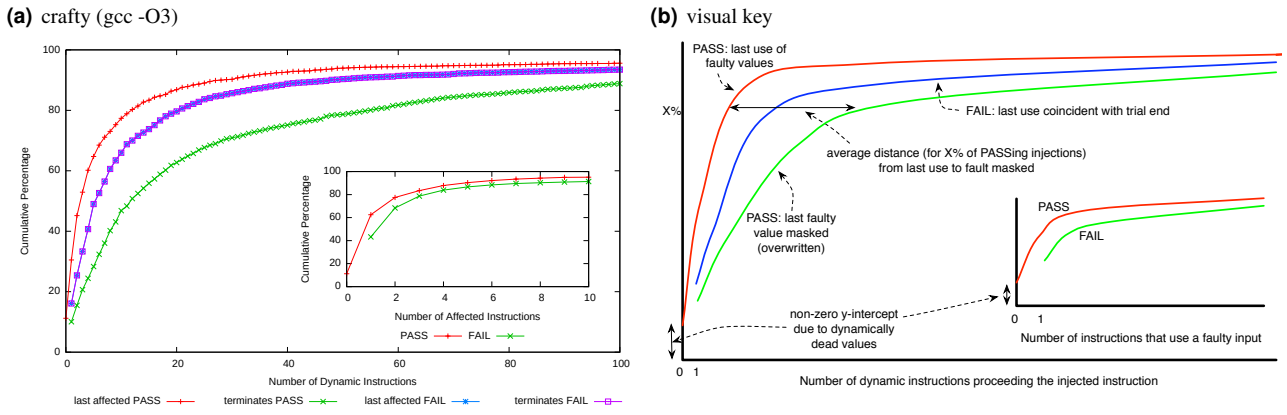


Figure 6: **Cumulative distribution of propagating injections by distance** A representative chart from `crafty` (gcc -O3) and a cartoon explaining how to interpret the chart.

all erroneous values have been expunged from architected state, *i.e.*, when they have been derated and masked, respectively. While these points occur at the same time (because the last use of an erroneous value overwrites the erroneous value with a correct value) in 28.3% of derated fault injections, there is a substantial discrepancy between these points in time in general.

As shown in Figure 6 (additional data in [8]), it is common to find that 90% of last uses occur within 20 instructions of the fault injection, yet the erroneous architectural register values are not expunged until much later, with 25% or more persisting for more than 50 dynamic instructions. This data implies that *most of the time a faulty value is present in the register file, it is in fact dynamically dead.* As we discuss in Section 5, this can be exploited to reduce the number of false positives reported by a error detection mechanism used in dual modular redundancy.

As shown in the inset figure in Figure 6 (additional data in [8]), most of the passing fault injections affect a relatively small number of instructions (*i.e.*, over 90% propagate to 4 or less instructions), but these instructions typically span multiple basic blocks (data not shown). Perhaps it is unsurprising that most of the passing fault injections are both short lived and have only local impact to the program's data flow, as we find that over 85% have a linear data-flow graph (*i.e.*, the data-flow chain that propagates faulty values consists entirely of nodes with out-degree of one).

In contrast, for fault injections that fail, there is a single event that marks the end of the injection trial: when a faulty value is used and an error occurs. As shown in Figure 6, this point generally occurs a relatively short time after the injection, but is generally later than the last use for passing fault injections. In addition, failing injections tend to affect a larger number of instructions on average which is likely

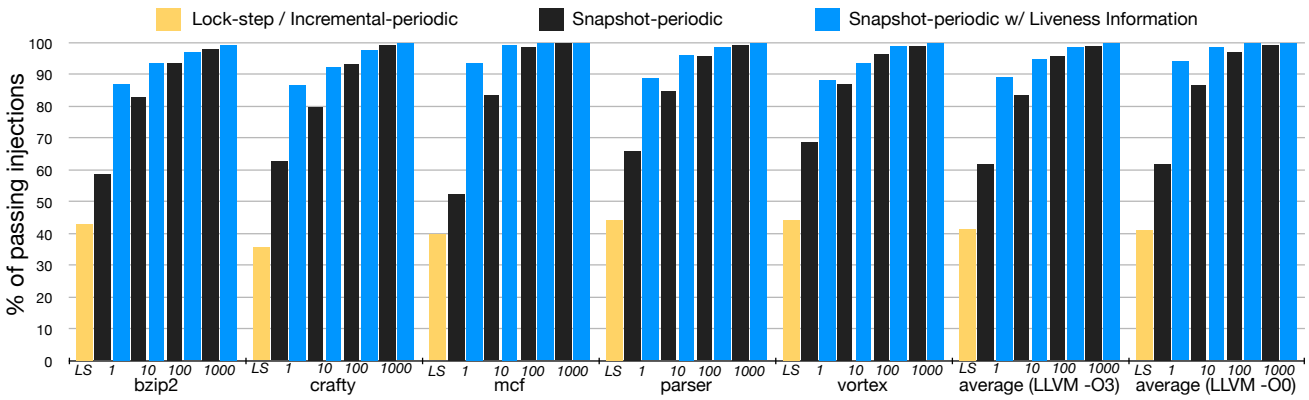


Figure 7: **Software error derating is a function of the error detection mechanism used.** The first bar of each data series represents the invariant derating rate of both the lock-step (LS) and incremental-periodic error detection schemes. The remaining bars show not only that the snapshot-periodic scheme has a higher derating rate, but also that it is a function of time; here, error detection intervals are swept from 1, to 10, 100, and 1000 check-fence instructions. A second experiment demonstrates that by using static register liveness information, the snapshot-periodic scheme can achieve most of the benefit of instruction-level derating even when performing error detection every few instructions.

in part why those fault injections fail.

5. Implications for Error Detection

In the introduction, we mentioned a trend in research toward processor designs that use robust fault tolerance techniques (*e.g.*, dual modular redundancy) to increase performance and/or reduce power. At present, however, manufacturers intentionally under-clock and over-provision voltage in processors to reduce the likelihood of processor faults. This new research seeks to eliminate the inefficiencies introduced by safety margins by running each part at the highest frequency and lowest voltage it can without frequent faults. By appropriately setting the clock and voltage, faults will occasionally occur, but the performance and power cost of recovering from them (using backward error recovery, for example) will be outweighed by the benefits achieved from higher frequency and lower power during the periods of correct execution.

Error derating plays an important role in these architectures because faults that are masked before error detection will not invoke a recovery nor incur the performance and power costs associated. As a result, when a larger fraction of faults can be masked, the result is a performance improvement because clock frequency may be scaled up further for the same number of recoveries.

It would be easy to assume that all error detection techniques that compare architected state would report the same number of errors, but the data from the previous section suggests that this is not the case, for two reasons: first, those faults that propagate to other instructions have the potential to be masked before being observed, and, second, the temporal separation between a faulty value's last use and its subsequent masking creates a time period where it would be detected as an error but could not affect the program's outcome (a false positive).

To support these assertions, we evaluate three implementations of error detection techniques for use in dual modular redundant systems — lock-step, incremental periodic, and snapshot periodic — which we describe below. Each of these techniques ensures correct execution by comparing the updates to architected state performed by the pair of processors, relying on mechanisms like ECC to prevent errors from being introduced into architected state by means other than incorrect execution. While logically performing the same comparisons, the granularity and mechanism of the techniques differ.

The *lock-step* approach ensures that both processors perform the same operations each cycle and compares the updates performed by each processor to architected state every cycle. In the event of any deviation, an error is reported. This scheme not only requires very high bandwidth between the two processors, but also, in relation to this work, will detect any propagating fault as an error to be corrected. Thus, the derating rate resulting from this error detection technique corresponds precisely to the passing no-propagate rates reported in Figure 5.

To reduce the required bandwidth between the two processors, the other two techniques summarize the the changes to the register file (which are most of the changes to architectural state) in the form of a signature, and compare these signatures *periodically* in addition to comparing a trace of branch outcomes as well as store addresses and data [22]. These techniques reduce the required bandwidth at the cost of introducing a small possibility of false nega-

tives, but the rate of false negatives can be controlled by the size of the signature.

The first periodic technique, *incremental periodic*, constructs the signature by incrementally folding into the signature the information relating to writes to the register file as they retire from the processor. Because it includes the values produced by every instruction, incremental periodic's derating rate is identical to that of lock-step.

In contrast, *snapshot periodic*, constructs its signature from a snapshot of the architected state. As the snapshot is only taken periodically, faulty values have the opportunity to be masked (overwritten) before the snapshot is created. As a result, the software derating available includes both that of the passing no-propagate as well as a fraction of the passing propagate; as we show in the black bars in Figure 7, the fraction depends on how frequently the comparisons are performed. Therefore *snapshot periodic* is guaranteed to achieve a derating rate that equals or exceeds that of the other two techniques, supporting our assertion that error detection rates can be different due to propagating errors that are later masked.

The data shown in Figure 7 was collected using the same fault injection methodology described in Section 3, augmented with the error detection mechanisms described above. The first bar in each graph (LS) denotes the error derating of the *lock-step* and *incremental periodic* techniques. For *snapshot periodic*, the error derating rate depends on the frequency of the checking, so we plot this function of frequency. In our implementation, we identify a subset of instructions (control, store, and system call instructions), which we will refer to as *check-fence* instructions. We only generate snapshots after these instructions, because we have found that doing so increases the number of masked errors without significantly introducing complexity in checking. We measure the derating rates that occur when checking at every check-fence instruction and at intervals of 10, 100, and 1000 checkfence instructions.[1] We show data for five of the twelve SPEC CINT2000 benchmarks run at full optimization (-03), whose results we believe to be representative for the whole suite; in addition, we include the average results for these programs without optimization (-O0) which result in the same basic trends with slightly higher levels of masking.

As previously noted, the black bars demonstrate that *snapshot periodic* error derating increases as we increase the interval between checks. While increased derating is beneficial, it is important that it not come at a huge increase in the latency to detect errors. For example, if an error occurs once every 10^5 instructions, then performing error detection once every 5,000 instructions will result in roughly 2.5% loss in performance, plus the overhead for recovery, as *on average* the faulting instruction will occur in the middle of the error detection interval. With a check-fence instruction occurring roughly every 4 instructions in our experiment, the intervals in Figure 7 correspond to checking for errors every 4, 40, 400, and 4,000 instructions.

A significant factor leading to increased derating with larger intervals is more temporal opportunity for a register containing an incorrect, and usually dead, value to be overwritten (*i.e.*, the average fraction of registers with faulty values decreases with time, as previously shown in Figure 6). Clearly, comparing the entire architected state is sufficient but not necessary for correctness; any values that are dead (*i.e.*, it is known that they will not be referenced again) need not be checked, since they cannot affect further computation. To explore what further fraction of unnecessary error recoveries could be eliminated, we modified the LLVM compiler to record static[2] register liveness information [16] associated with each check-fence instruction. In a second set of experiments shown in Figure 7, we show that by comparing only statically live register values between processors, almost 90% of the instruction-level error derating can be achieved, even with very small error detection intervals.

6. Conclusion

In this work, we demonstrated the mechanisms that result in *instruction-level* error derating; that is, how incorrect architectural state or incorrect instruction execution can result in correct program behavior. We classified the mechanisms into six categories: *value comparison*, *sub-word operations*, *logical operations*, *overflow/precision*, *lucky loads*, and *dynamically dead values*. Unlike previous work on software derating that exploits the structure of the computation (*e.g.*, [26]) or the numerical properties of specific applications (*e.g.*, [15]), these are general-purpose instruction properties found in all of the programs that we studied. Even if we conservatively restrict the opportunity for instruction-level derating by considering a fault as an error if it propagates to the memory system or affects control flow, we find that 36% of architecturally visible faults are derated and masked.

Knowledge of the mechanisms of software derating provides system builders the opportunity to exploit them. We considered two instruction-level derating-motivated opportunities in the context of systems that exploit the presence of error detection mechanisms to improve performance or reduce power consumption. First, we demonstrated that comparing architected state via periodic snapshots permits faults to be masked that would otherwise lead to error recovery

[1] In these experiments, we do not actually reduce the register state to a signature, as doing so only adds the possibility of false negatives.

[2] To be clear, static liveness information makes no assumptions about the further path of execution, so some false positives will still occur for values that are dynamically dead (will not be used on the control flow path that ends up to be taken) and will later be recognized as statically dead at some later point in the code.

actions if the result of every instruction were incorporated. Second, we demonstrated that the rate of derating can be further increased by excluding the contents of known dead registers from the comparison.

Looking forward, we believe there are other applications that are enabled by an understanding of the mechanism of instruction-level derating. In particular, we are interested in investigating the degree to which instruction-level derating can be increased by the optimizations performed by a compiler and how it handles code generation.

Acknowledgment

This research was supported in part by NSF CCF-0702501 and NSF CAREER award CCF-0347260. The authors would also like to thank Lee Baugh, Brian Greskamp, Edward Lee, Naveen Neelakantam, and Pierre Salverda for their many useful comments.

References

[1] The LLVM Compiler Infrastructure. Home page: `http://llvm.cs.uiuc.edu/`.

[2] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, Feb. 2002.

[3] T. M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proc. of the Intl. Symp. on Microarchitecture*, pages 196–207, 1999.

[4] J. Blome, S. Mahlke, D. Bradley, and K. Flautner. A microarchitectural analysis of soft error propagation in a production-level embedded microprocessor. In *Proc. of the Workshop on Architectural Reliability*, 2005.

[5] S. Borkar et al. Parameter variations and impact on circuits and microarchitecture. In *Proc. of the Annual Conf. on Design Automation*, pages 338–342, 2003.

[6] J. A. Butts and G. S. Sohi. Characterizing and predicting value degree of use. In *Proc. of the Intl. Symp. on Microarchitecture*, pages 15–26, Nov. 2002.

[7] H. Cha, E. M. Rudnick, J. H. Patel, R. K. Iyer, and G. S. Choi. A gate-level simulation environment for alpha-particle-induced transient faults. *IEEE Trans. on Computers*, 45(11):1248–1256, Nov. 1996.

[8] J. J. Cook and C. Zilles. Characterizing instruction-level error derating. In *Proc. of the IEEE Workshop on Silicon Errors in Logic System Effects*, Mar 2008.

[9] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Zeisler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *Proc. of the Intl. Symp. on Microarchitecture*, pages 7–18, 2003.

[10] M. Favalli and C. Metra. Optimization of error detecting codes for the detection of crosstalk originated errors. In *Proc. of the Conf. on Design, Automation and Test in Europe*, 2001.

[11] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. In *Proc. of the Intl. Symp. on Computer architecture*, pages 98–109, 2003.

[12] B. Greskamp and J. Torrellas. Paceline: Improving single-thread performance in nanoscale cmps through core overclocking. In *Proc. of the Intl. Conf. on Parallel Architecture and Compilation Techniques*, pages 213–224, 2007.

[13] W. Gu, Z. Kalbarczyk, R. Iyer, and Z. Yang. Characterization of linux kernel behavior under errors. In *Proc. of the Intl. Conf. on Dependable Systems and Networks*, June 2003.

[14] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. FERRARI: A flexible software-based fault and error injection system. *IEEE Trans. on Computers*, 44(2):248–260, Feb. 1995.

[15] X. Li and D. Yeung. Application-level correctness and its impact on fault tolerance. In *Proc. of the Intl. Symp. on High-Performance Computer Architecture*, 2007.

[16] M. M. Martin, A. Roth, and C. N. Fischer. Exploiting dead value information. In *Proc. of the Intl. Symp. on Microarchitecture*, pages 125–135, 1997.

[17] F. Mesa-Martinez and J. Renau. Effective optimistic-checker tandem core design through architectural pruning. In *Proc. of the Intl. Symp. on Microarchitecture*, 2007.

[18] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K. S. Kim. Robust system design with built-in soft-error resilience. *IEEE Computer*, 38(2):43–52, Feb. 2005.

[19] 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 *Proc. of the Intl. Symp. on Microarchitecture*, pages 29–40, 2003.

[20] G. P. Saggese, A. Vetteth, Z. Kalbarczyk, and R. Iyer. Microprocessor sensitivity to failures: control vs. execution and combinational vs. sequential logic. In *Proc. of the Intl. Conf. on Dependable Systems and Networks*, pages 760–769, June 2005.

[21] 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 *Proc. of the Intl. Conf. on Dependable Systems and Networks*, pages 389–398, June 2002.

[22] J. C. Smolens, B. T. Gold, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk. Fingerprinting: bounding soft-error detection latency and bandwidth. In *Proc. of the Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 224–234, 2004.

[23] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The case for lifetime reliability-aware microprocessors. In *Proc. of the Intl. Symp. on Computer Architecture*, 2004.

[24] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: Improving both performance and fault tolerance. In *Proc. of the Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 257–268, Nov. 2000.

[25] D. D. Thaker, D. Franklin, V. Akella, and F. T. Chong. Reliability requirements of control, address, and data operations in error-tolerant applications. In *Proc. of the Workshop on Architectural Reliability*, 2005.

[26] N. Wang, M. Fertig, and S. Patel. Y-branches: When you come to a fork in the road, take it. In *Proc. of the Intl. Conf. on Parallel Architectures and Compilation Techniques*, page 56, 2003.

[27] N. J. Wang, A. Mahesri, and S. J. Patel. Examining ACE analysis reliability estimates using fault-injection. In *Proc. of the Intl. Symp. on Computer Architecture*, pages 460–469, 2007.

[28] 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 *Proc. of the Intl. Conf. on Dependable Systems and Networks*, pages 61–70, June 2004.