Configurable Transient Fault Detection via Dynamic Binary Translation

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

Robert Cohn Shubhendu S. Mukherjee
VSSAD and FACT Groups
Intel Massachusetts
{robert.s.cohn,shubu.mukherjee}@intel.com

ABSTRACT

Smaller feature sizes, lower voltage levels, and reduced noise margins have helped improve the performance and lower the power consumption of modern microprocessors. These same advances have made processors more susceptible to transient faults that can corrupt data and make systems unavailable. Designers often compensate for transient faults by adding hardware redundancy and making circuit- and process-level adjustments. However, applications have different data integrity and availability demands, which make hardware approaches such as these too costly for many markets.

Software techniques can provide fault tolerance at a lower cost and with greater flexibility since they can be selectively deployed in the field even after the hardware has been manufactured. Most existing software-only techniques use recompilation, requiring access to program source code. Regardless of the code transformation method, previous techniques also incur unnecessary significant performance penalties by uniformly protecting the entire program without taking into account the varying vulnerability of different program regions and state elements to transient faults.

This paper presents *Spot*, a software-only fault-detection technique which uses dynamic binary translation to provide software-modulated fault tolerance with fine-grained control of redundancy. By using dynamic binary translation, users can improve the reliability of their applications without any assistance from hardware or software vendors. By using software-modulated fault tolerance, *Spot* can vary the level of protection independently for each register and region of code to provide users with more, and often superior, fault-detection options. This feature of *Spot* increases the mean work to failure from 1.90x to 17.79x.

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 smaller size and sheer number make chips more susceptible to *transient faults* [10]. Transient faults, caused by external particle strikes or process-related parametric variation [1, 13, 23], do not cause permanent damage, but may manifest as soft errors by altering signal transfers and stored values, thereby resulting in incorrect program execution.

Designers frequently introduce redundant hardware to detect or recover from these faults because process and material advances are often insufficient to mitigate their effect. For example, storage structures, such as caches and memory, typically include extra information in the form of parity or error-correcting codes (ECC), which allow these hardware structures to detect and/or recover from such faults. However, protecting all transistors is difficult, particularly those used in combinational logic, without paying a significant penalty in area, power, and/or performance. While higher-level techniques, such as Lockstepping or Redundant Multithreading [19], have been proposed for full-processor fault detection, they 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.

In contrast, software-only approaches to fault detection and recovery [14, 16, 20, 24] can significantly improve reliability without requiring hardware modifications. This makes software redundancy techniques significantly cheaper and easier to deploy, working even on machines already in the field. Deployment of redundancy techniques in the field may become important because of poor estimates of the severity of the soft-error rate by designers and because of the uncertainty in the usage condition of the machine [4, 9]. Changes to the operating environment of the hardware can also have a noticeable effect on reliability, requiring the deployment of software redundancy techniques. For example, the soft-error rate from atmospheric neutrons is 4-5x higher in Denver than in New York City because of Denver's higher altitude [29]. Nevertheless, software-only approaches do suffer from high performance penalties – a problem we tackle in this paper.

Most prior approaches to software-only error mitigation [2, 14, 16, 20] have relied primarily on static compilation techniques that require alterations to the compilation process and access to the application's source code. To use these techniques, a user must collaborate with the software vendor to acquire the application source code, often rendering these techniques impractical for many applications.

The technique presented in this paper, *Spot*, provides fault detection via dynamic binary translation for all types of instructions. *Spot* uses a modified version of the Pin dynamic instrumentation framework [6] to perform reliability transformations for the x86 instruction set. Because Pin uses dynamic, rather than static, instrumentation, *Spot* only requires the program binary; *Spot* is applicable to programs, legacy or otherwise, whose source code is either not easily re-compilable or altogether unobtainable from the software vendor. Even if the application sources are available, users typically do not recompile libraries (such as `libc`) when recompiling an application. *Spot*'s dynamic translation capabilities allow protection to be applied to libraries as well. The dynamic nature of *Spot* also enables it to attach and detach to already running applications to adjust reliability appropriately. For example, a laptop can switch into a more reliable mode of execution whenever it detects adverse environmental factors.

Unlike prior work, *Spot* helps users identify the most vulnerable regions and state of programs and configure these regions and state with the appropriate level of fault coverage. Generally, the higher the number of redundant instructions (and hence fault coverage) added to the original binary application, the greater the application's performance degradation. By allowing users to identify and protect only regions and state of the program that are highly vulnerable to transient faults, *Spot* helps reduce the performance degradation from such software reliability transformations. Thus, *Spot* allows applications to trade off performance for reliability and vice versa. Although previous research has explored such reconfigurability in a limited way at the function level [22] (without actually exploring the performance and fault-coverage trade-off), *Spot* takes advantage of reconfigurability at a much finer granularity. Note, however, that *Spot* does not automatically choose regions of programs to protect, but it provides a foundation for such automation.

To maximize fault coverage for a given performance threshold, we take advantage of the key insight that not all faults in an unprotected application will cause the application to produce incorrect data. For example, a fault to a register outside of its live range will not change the program execution, because that value will be overwritten before it is used. Due to such masking factors, unprotected applications have a natural degree of fault resistance. The natural resilience varies not only from application to application, but also within an application from region to region and from register to register.

While it is well-known that masking effects are common in applications and vary widely within and across applications [5, 12, 26], we demonstrate that such masking effects can be effectively exploited under software control to trade off fault-detection coverage for increased performance. *Spot* identifies vulnerable regions and registers by profiling the vulnerability of regions and registers using fault injection into running programs. Armed with this profile information, *Spot* can selectively apply redundant transformations only for the regions and registers deemed necessary.

Spot is also the first complete software-only technique to target the x86 architecture. Previous software-only techniques have targeted the MIPS architecture [14], the Intel® IPF architecture [20], and a Motorola M68040 microprocessor [16]. Previous work does target the x86 architecture, however, that work only addressed the issue of control-flow protection [2]. Enabling full fault detection on x86 processors is an attractive goal due to their ubiquity; x86 processors are used in a wide variety of applications from low-power laptops which may not require very high levels of data integrity but which may change environments quite often, to high-end Xeon® and Opteron® processors that require high levels of data integrity, such as those used in high availability systems. Hence, it is critical to provide x86 users with the ability to tailor the level of reliability to their needs. However, the x86 architecture presents *Spot* with a novel set of challenges due to the high frequency of memory accesses and the limited number of architectural registers, making the mitigation of the performance degradation a much more challenging problem than it is for RISC instruction set architectures.

The rest of the paper is organized as follows. Section 2 defines the fault detection design space and explains how *Spot* is in a unique point in that space. Sections 3 describe the methodology used to evaluate the different implementations, in terms of reliability (3.1), performance (3.2), and Mean Work To Failure (3.3). Section 4 describes the results from three software modulations options: configuring protecting for registers (4.1), regions (4.2), and both simultaneously (4.3). Finally, the paper concludes in Section 5.

2. FAULT TOLERANCE DESIGN SPACE

Table 1 shows the design space of various fault-detection techniques. Each column in the table represents a different substrate that can be used to implement the fault detection technique. Fault-detection can be done in software alone, in hardware alone, or a combination of the two. The rows of the table represent the different ways of modulating the level of protection. The first row represents existing techniques that apply protection uniformly across all applications and all parts of individual applications. The second and third rows refer to techniques that allow the software and hardware, respectively, to modulate the level of protection. In this section, we explore these two axes of the design space.

Modulation Method	Implementation Method		
	Software	Hybrid	Hardware
None	SWIFT [20] EDDI [14] ACFC [25] Borin et al. [2]	CRAFT [21] Watchdog [7] CFCSS [15]	RMT [11, 19] Lockstepping [27] ECC, Parity
Software	PROFiT [22] *Spot*	PROFiT [22]	
Hardware			PER, IRTR [3]

Table 1: Fault detection design space.

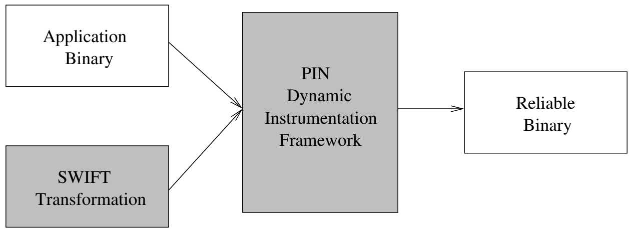


Figure 1: Diagram of the framework.

2.1 Implementation Method

Numerous low-level techniques, including error correcting codes (ECC) and parity bits, are commonly used to increase reliability. While these are effective at protecting regular structures such as caches, they cannot economically scale to cover all bits of a processor, specifically many of the latches and logic gates in the instruction pipeline. To address this, many high-level hardware techniques have also been proposed [11, 19, 27]. While such techniques are typically able to dramatically increase reliability, they still require significant changes to the hardware design and possibly the operating system, significantly increasing design and validation time and often incurring performance penalties. Software-only approaches [14, 15, 16, 20, 25] have been proposed as cheaper alternatives to hardware solutions since they require no hardware modifications. However, most previously proposed software-only techniques have been implemented as compiler passes, and thus they have all required access to source code. Since the source code to many applications is unobtainable, these techniques cannot be widely applied. Recently, Borin et al. [2] introduced a software-only technique using dynamic translation, but that work was limited solely to protecting control-flow. *Spot* is the first software-only approach to increase the reliability of non-control-flow instructions via dynamic instrumentation. *Spot* allows users to protect themselves against the deleterious effects of transient faults *without the cooperation of hardware and software vendors*.

Spot's reliability enhancements were implemented using the Pin dynamic instrumentation framework [6]. Pin allows users to write instrumentation code, our reliability transformation in this case, and apply it to binaries via dynamic instrumentation. Figure 1 shows a diagram of the framework.

Upon execution, an application binary is loaded by Pin, and the reliability transformation is applied. As the code is executed, it is instrumented, yielding reliable code dynamically. While dynamic instrumentation does incur a performance overhead, Pin's caching technique allows this cost to be amortized over the execution of the program. Through dynamic instrumentation, Pin is able to handle many challenging issues that cannot be solved via static instrumentation, such as variable-length instructions, mixed code and data, statically unknown indirect jump targets, dynamically generated code,

```
                             1:  cmp %edx  , %edx2
                             2:  jne faultDetect
mov (%edx), %eax                 mov (%edx), %eax
                             3:  mov %eax  , %eax2
sub %eax, %ebx                   sub %eax  , %ebx
                             4:  sub %eax2 , %ebx2
                             5:  cmp %edx  , %edx2
                             6:  jne faultDetect
                             7:  cmp %ebx  , %ebx2
                             8:  jne faultDetect
mov %ebx, (%edx)                 mov %ebx  , (%edx)

   (a) Original Code              (b) Reliable Code
```

Figure 2: Duplication and validation. Code added by our technique is in bold. Checking instructions are italicized.

and dynamically loaded libraries.

Spot's reliability transformation is based on the SWIFT technique [21]. Our technique dynamically duplicates all instructions, except for those that write to memory. Since a fault causing data corruption will only manifest itself as a program error if it changes the output, checking is delayed until immediately before instructions that may affect output, such as stores. An error in a dynamically dead register or in a bit that will be masked away will not cause a failure because that value (correct or not) will not propagate to the output. By delaying validation until necessary, this technique reduces the probability that errors will be signaled in situations where that error would not have caused a failure. This also greatly reduces the number of checking instructions that are necessary.

Figure 2 is a simple example which illustrates *Spot*'s instruction duplication and checking. To protect the load instruction, instructions **1-2** are added to verify that the address of the load instruction is correct. Instruction **3** is inserted to add redundancy to the data loaded from memory by copying the value to a duplicate virtual register. Instruction **4** is inserted to redundantly compute the subtraction and instructions **5-8** verify that both the address and the value sources of the store instruction are fault-free.

Uncachable loads, such as those from external devices, and loads in multiprocess systems may cause two successive loads to the same address to read different values. Therefore, *Spot* copies loaded values in order to avoid spuriously signaling faults. Such is the case in instruction **3**. Further, since there is only one instruction that fetches data from memory in the reliable version, the address of the load instruction must be validated. If the address is incorrect, an invalid value will be given to both versions of the program, causing an undetected error. Similar solutions have been devised for other software and hardware techniques [11, 19, 20].

In addition to loads, other special instructions require similar handling, such as the `RDTSC` instruction, which reads the hardware time-stamp counter. The x86 also has a CISC instruction set which means that the notion of a load instruction must be applied to a much wider class of instructions than would be the case in a RISC architecture. While this technique has been shown to demonstrably reduce the number of output-corrupting faults, it does suffer from a few flaws which render its protection imperfect. First, any software-only technique can only protect state at the architectural level, since micro-architectural state is generally not visible to software. Second, the technique has inherent "windows of vulnerability," most notably between checks and uses of the checked value. For example, a fault to `edx` after instruction **5** or a fault to `ebx` after instruction **7** will go into the store undetected, corrupting memory. Third, the Pin engine itself is not instrumented. As mentioned earlier, and as we will show in Section 3, Pin consumes only a small fraction of the execution time and therefore does not contribute significantly to the vulnerability of the technique. In addition to these limitations of the transformation, due to current limitations with Pin framework *Spot* does not protect floating point or multimedia instructions, nor does our framework currently incorporate SWIFT's control-flow protection scheme.

Although *Spot* uses the same transformation as previous approaches, it is the first technique to apply it dynamically, making the transformation applicable to libraries, dynamically-generated code, and variable-length instruction sets with indirect jumps. The dynamic nature of *Spot* also removes many hurdles for users wanting to increase the reliability of their system. Previously, users would have had to physically shield their hardware, purchase more reliable hardware, or obtain more reliable software to increase reliability. Static approaches based on compiler transformations require the source code to the application. For most applications, the source code is unavailable, making dynamic approaches such as *Spot* the only available option for increasing reliability.

2.2 Modulation Method

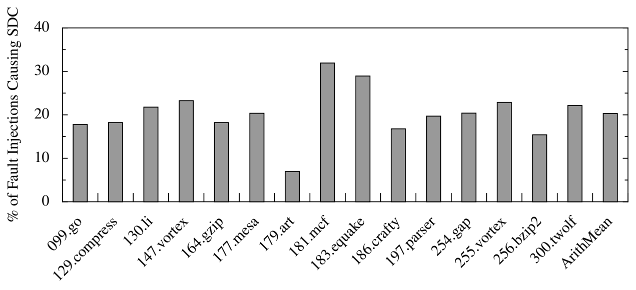


Figure 3: Variations in natural fault resilience.

Different applications have different levels of natural fault resilience. Figure 3 shows the each application's response to inserted faults. On average, the unprotected benchmarks fail 20.23% of the time, although the percentage of executions resulting in failure ranges from 31.92% for `181.mcf` to 6.99% for `179.art`. Just as entire benchmarks vary in their response to faults, individual registers or regions of a particular benchmark also have varying levels of reliability. This is analogous to the variation in Instructions Per Cycle (IPC) among applications. The IPC will vary from application to application, depending on the nature of the application. In addition, the IPC will vary within a single application as different phases of the program are executed.

The PROFiT technique [22] was the first technique to capitalize on the non-uniformity of natural reliability via software modulation. It is a coarse-grained technique, adjusting protection at the function-level. The PROFiT technique created the reliability configuration for the application at compile time and could be used in conjunction with either a software-implemented technique or a hybrid technique.

Both Partial Explicit Redundancy (PER) and Implicit Redundancy Through Reuse (IRTR) [3] add modulation at the hardware level to an RMT processor. PER reduces the performance cost of RMT by performing full RMT redundancy only when there are excess hardware resources and scaling back the redundancy when the original thread is utilizing all of the processor's resources. IRTR also decreases the overhead of the RMT technique at the cost of some reliability by reusing instructions from the original thread in the trailing thread when the instructions and operands are identical.

Our technique, *Spot*, is the first software-only technique to provide fine-grained *Software-Modulated Fault Tolerance* (SMFT). *Spot* can help users tailor the reliability and performance of specific applications, critical registers of an application, or even of critical regions of an application, as well as combinations thereof. While previous work [8, 28] has explored protecting a subset of the architectural registers, we are the first to explore the reliability/performance trade-offs associated with varying the level of protection. Other research has exploited modulation at the function granularity [22], but *Spot* is the first technique to help users enact software modulation at even finer granularities, enabling superior performance and reliability trade-offs. In this work, we explore two fine-grained aspects of reconfigurability in Section 4 and show how they provide a wide range of options in terms of performance and reliability.

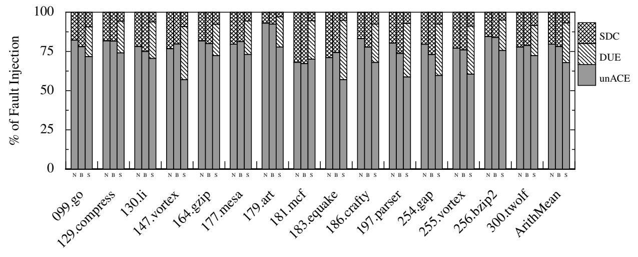


Figure 4: Categorization of fault injections.

3. EVALUATION METHODOLOGY

This section presents the experimental methodology and shows performance and reliability results for native program execution, baseline Pin execution, and baseline *Spot* execution. In order to effectively evaluate reliability, we measure the level of data corruption as well as the Mean Work To Failure (MWTF) [21] for each of the executions. In Section 4 we evaluate various non-uniform applications of *Spot*.

3.1 Reliability

To evaluate the reliability of the original application, the Pin framework, and *Spot* in all of its configurations, we used simulated fault injections under a Single Event Upset (SEU) model. At a random point in time uniformly distributed over the execution of the program, including the Pin framework itself, a single-bit fault was inserted into a random bit of a random architectural register. Each of the 32 bits of the 8 architectural registers were equally likely to receive a fault. In order to obtain accurate results, over 1.03 million executions were run in total on actual hardware.

After injecting a single fault into the execution, the program is run to completion, unless it aborts, and its output is compared to a known good output. If the program's output and exit code were identical to the good execution, that bit was considered Unnecessary for Architecturally Correct Execution (unACE) [10]. If the output and exit code were different, then the fault caused a Silent Data Corruption (SDC); data in the program was affected, but it was not detected. An SDC can occur for numerous reasons, including early termination due to various exceptions (segfaults, illegal instruction, divide by zero, etc.), incorrect output of any form, or infinite execution. The implementation labeled a program as having infinite execution if it took more than 10x the original, non-faulty program execution time. For *Spot* executions, if the framework detected a fault, then the bit was classified as a Detected, Unrecoverable Error (DUE).

SDC and DUE are two mutually exclusive components of the soft error rate of a chip. The SDC rate can be further decomposed into: $\text{SDC} = \text{AVF}_{\text{SDC}} \times \text{intrinsic raw error rate}$, where AVF represents the Architectural Vulnerability Factor, the fraction of faults to a structure that result in a user-visible error. The SDC rate is inversely proportional to the Mean Time To Failure (MTTF).

This reliability evaluation only considers faults into the architectural register file. Hence, it is not a complete hardware reliability analysis. It does not model faults to micro-architectural state such as bypass networks. Nevertheless, the evaluation yields quantitative AVF results for an x86 processor's register file and insights into the fault detection coverage on an important section of a processor [26].

Figure 4 shows the percentage of executions that resulted in unACE, DUE, and SDC for each benchmark and each configuration. The three configurations are *nopin* (N), the original, compiled binary without any instrumentation, *basepin* (B), a binary translation in the Pin framework with no reliability or other additional code, and *Spot* (S), the baseline *Spot* technique with reliability uniformly applied. By running a total of 225,000 (5,000 per benchmark per configuration), the 95% confidence interval on these results is 1.50%.

Because not all faults inserted into an unprotected application will cause the application to fail, the unACE for each of the benchmarks is far from zero. Due to factors such as logical masking, dynamically dead instructions, and silent stores, unprotected applications have a certain amount of natural fault resistance. On average, the unprotected benchmarks have 20.23% SDC, although the SDC percentage ranges from 31.92% for `181.mcf` to 6.99% for `179.art`. Since *Spot* uses the Pin framework, we also evaluated the reliability of the applications running under Pin, but without any protection. Applications running with Pin had a slightly higher SDC than running without Pin. On average, the percentage of SDC executions was 21.94%, a 1.71% difference versus *nopin*.

We also evaluated the reliability of a baseline *Spot* configuration with uniform protection across the entire application and for all applications. This baseline *Spot* increases reliability decidedly, reducing the SDC percentage from 20.23% to 6.46%, an increase in reliability by a factor of 3.13x. Again, different benchmarks exhibit differences in their reliability, with `181.mcf` having the lowest SDC (5.95%), and `300.twolf` having the highest SDC (8.98%). However, using the baseline *Spot* tool does not provide 100% reliability. As mentioned in Section 2.1, *Spot* inherits various vulnerabilities from SWIFT.

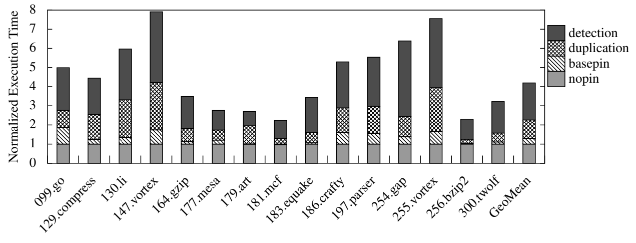
`181.mcf` is an example of a benchmark where the baseline *Spot* performs well. The reliability improves by 10.72x over *nopin*, with little reduction in unACE. Because of the memory-intensive nature of `181.mcf`, most failures in this application are due to segmentation faults and the baseline *Spot* successfully catches most of them. The baseline *Spot* applied to `147.vortex` also greatly increases its reliability, but with a much more noticeable reduction in unACE.

Spot, like most fault detection implementations, will detect some unACE bits as DUE. *Spot* detects faults in the system which, if allowed to propagate through the program, would not cause an error. For example, a silent store which stores an incorrect value will not cause an error because the value in memory will not be used before being overwritten. *Spot* will detect the faulty value propagating to memory and signal a DUE despite the fact that the fault would not have caused an error.

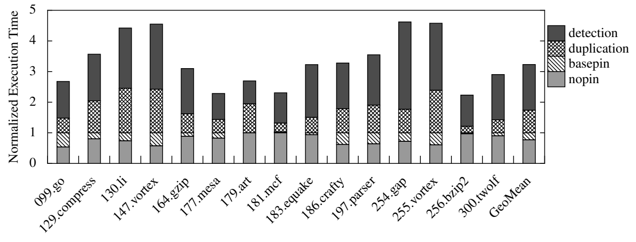
3.2 Performance

We evaluated the performance of each configuration by running natively on an Intel® Pentium® D with a clock frequency of 2.80GHz and 4 GB of RAM running Fedora Core 3 with a 2.6.15 kernel. All binaries were compiled statically with gcc 3.4.4 and the user time of the execution was measured.

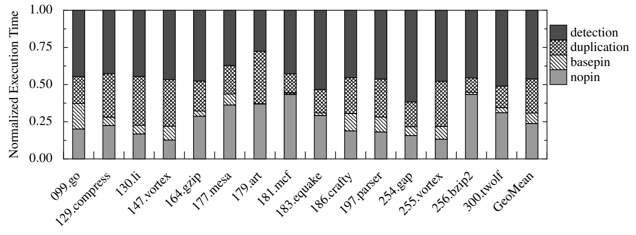
Figures 5(a) shows the execution time of the baseline *Spot* normalized to the native, uninstrumented application. Figure 5(b) shows the same execution time normalized to the execution time of the application running under Pin, but without any reliability transformation



(a) Performance normalized to *nopin*.



(b) Performance normalized to *basepin*.



(c) Performance normalized to *Spot*.

Figure 5: Performance breakdown for dynamic fault detection.

applied.

Pin has two distinct sources of overhead, a one-time Just In Time (JIT) compilation cost, and an steady-state runtime cost. The first is associated with performing a one-time translation of the original instructions to instructions within the Pin framework. Even when no reliability transformation is applied (*basepin*), the code is still nonetheless translated. This compilation cost only occurs once per region in the application, since Pin caches translated regions as mentioned in Section 2.1. The other cost is the increase in execution time due to executing the translated code. The translated code is less efficient than the original binary's version because additional checks are inserted to ensure that control transfers to new regions go through the Pin virtual machine [6].

This steady-state overhead is much more important than the one-time compilation overhead, as the cost of the one-time compilation can be amortized for long running programs. For the benchmarks in the performance evaluations, the percent of time consumed by the one-time compilation ranges from 2% - 11%. By using the notion of persistence in run-time translation systems introduced by Reddi et al. [17, 18], this one-time cost can be further reduced.

On average, the execution time of the baseline *Spot* is 4.19x slower than a native execution. When factoring out the framework overhead, the execution time is 3.22x slower. The performance cost of adding reliability varies greatly for each individual benchmark, with `181.mcf` having the smallest performance cost (2.3x) and `254.gap` having the largest performance cost (4.6x). Benchmarks like `181.mcf` which contain many cache misses have excess instruction level parallelism with which to execute the redundant and checking code without affecting the critical path. Benchmarks with high checking-code costs, such as `254.gap`, do so because of their large number of memory instructions. This includes not only memory loads and stores, but also arithmetic operations that act directly on memory locations, such as `incl (%eax)`.

As Figure 5(c) shows, *Spot* spends 46.2% of its time on average checking memory instructions. As explained in Section 2.1, x86 applications contain numerous memory operations because of the limited number of architectural registers. Floating point applications like `177.mesa` and `179.art` spend less time in *Spot* code since they typically spend most of their time performing floating point operations which are not checked. `254.gap`, on the other hand, spends 62.8% of its time in validation code.

While the baseline *Spot* technique incurs a significant performance cost, we will demonstrate in the following section that this cost can be dramatically reduced by using software-modulation.

3.3 Mean Work To Failure

Although metrics of reliability such as AVF and MTTF are useful in most situations, they cannot be used as easily to compare systems in which both reliability and performance differ. For example, if a system reduces the SDC failure rate by $\frac{1}{2}$ while increasing execution time by 4x, the original system will be able to accomplish more work between SDC failures. Mean Work To Failure (MWTF) [21] is a metric which takes into account both the AVF and the performance to give a more meaningful comparison in these cases.

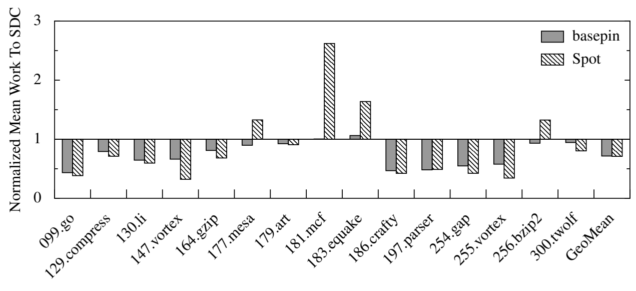


Figure 6: Normalized Mean Work To Failure.

Figure 6 shows the MWTF for *basepin* and uniformly applied *Spot* baseline normalized to the native application without instrumentation. For these MWTF measurements, we consider SDCs to be failures. The mean normalized MWTF of basepin is 0.72x. This metric is less than 1.0, meaning that the basepin will, on average, suffer more SDCs during the time it takes to complete a given application versus running without Pin. This result is not surprising since the reliability of the Pin tool was roughly equivalent to that of the native execution, but the execution time was slightly longer. This results in an application which is just as susceptible to failures, but which will take longer to complete and therefore be exposed for a longer amount of time.

The baseline *Spot* tool also has a worse MWTF than the native application, with an average normalized MWTF of 0.71x. While this may seem surprising because of the dramatic decrease in SDC that *Spot* provides, the execution time of baseline *Spot* is also increased by a factor greater than the reduction in SDC. This example shows that reliability analysis must go beyond the SDC percentage and include performance as well. For certain benchmarks, like

`181.mcf` and `183.equake`, baseline *Spot* improves the MWTF. As demonstrated in Figure 4, the SDC improvements under *Spot* for these benchmarks are among the highest. `181.mcf` has the lowest performance cost under *Spot* while `183.equake` has an average performance cost. The SDC improvement for these benchmarks outweighs the performance cost, resulting in a overall gain in terms of MWTF.

The baseline *Spot* tool suffers from a large performance cost, and in this often negates the benefit from the decrease in SDC. However, *Spot* can apply fault detection non-uniformly via software modulation. By exploiting the variations in SDC throughout the application (and among different applications), *Spot* shows a clear benefit in terms of reliability and MWTF. As the following sections will show, selectively applying fault detection can significantly lower the SDC while keeping the performance cost in check, thereby increasing the MWTF of *Spot* well beyond that of the baseline.

4. SOFTWARE MODULATION RESULTS

Spot enables software-modulated fault detection and can trade-off between reliability and performance at a fine granularities. *Spot* can add reliability in certain environments, for specific applications, or even for critical regions of an application, thus maximizing the reliability while minimizing the costs. This enables *Spot* to protect as much or as little of the program as is required by the user's performance and reliability budget. In the future, *Spot*'s dynamic nature will also allow the level of protection to be dynamically adapted to changing performance and reliability requirements.

In this section, we explore two axes along which the protection can be modulated. We show how these axes provide a wide range of options in terms of performance and reliability. Although we have explored this space across all benchmarks, for brevity's sake, we will examine three benchmarks in depth, `254.gap`, `300.twolf`, and `181.mcf`, which represent the worse, average, and best performing benchmarks from the baseline *Spot* executions. In Section 4.1, we present the variations available when protecting only a subset of the architectural registers, and in Section 4.2 we explore different reliability configurations based on regions.

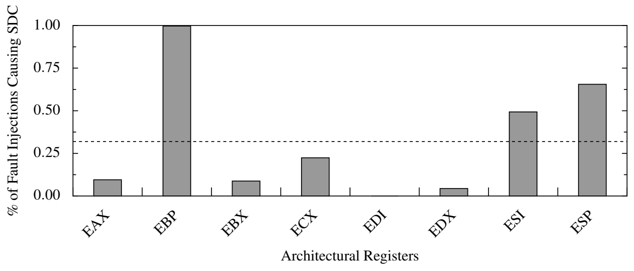
4.1 Varying Protection on Registers

Just as different applications differ in their native reliability, different architectural registers differ in their reliability within an application. Figures 7(a), 7(b), and 7(c) show the AVF of each architectural register for each of our three benchmarks. The horizontal dashed line denotes the average AVF across all registers.

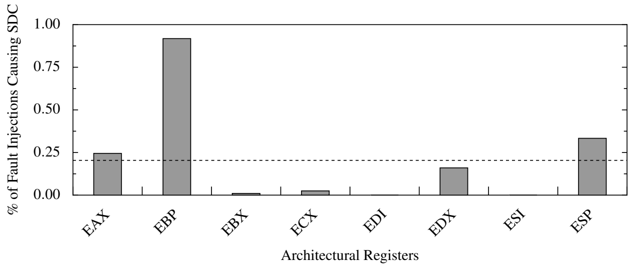
Registers with above-average AVF are good candidates for specialized protection. Figure 7 clearly demonstrates that certain registers, namely `EBP` and to a lesser extent `ESP`, are very susceptible. Since these registers are used primarily as pointers loading and storing to memory, faults to these registers will likely cause segmentation faults. `EDI` is also a significant source of failure for `181.mcf`. Similar to `EBP`, this register is heavily used in certain memory operations.

Spot can individually protect or leave unprotected any subset of the architectural registers. Leaving a register unprotected reduces the steady-state performance overhead of reliability by not duplicating instructions that only affect an unprotected register and also by not having to insert check instructions for that register.

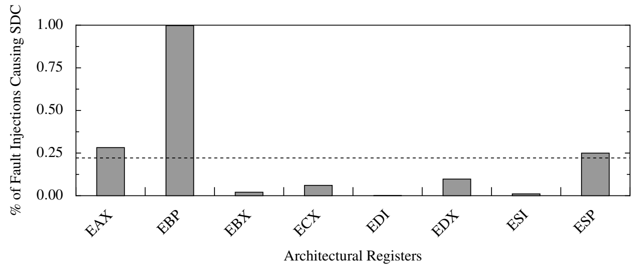
Figure 8 is a simple example to illustrate the instruction duplication and verification of *Spot* when protecting only a subset of the registers. In this example, register `EBX` is left unprotected. Since `EAX` is protected, instruction **1** is inserted, as before, to add redundancy to the data loaded from memory. The `sub` instruction which reads from `EAX` and `EBX` and writes to `EBX` is *not* duplicated, since `EBX` is not protected. The last instruction is a store instruction which stores `EBX` to the address `EDX`. Instructions **2-3** verify that the address, `EDX`, is correct, but the value, `EBX`, is written to memory without checking.



(a) 181.mcf



(b) 254.gap



(c) 300.twolf

Figure 7: Percentage of faults causing SDC for each architectural register.

Spot was run in 256 different configurations for each benchmark. Each configuration protected a subset of the registers and we evaluated the protection and performance for all combinations of the 8 GP architectural registers. For an accurate evaluation, each configuration was injected with 500 faults for each of the 256 combinations for each benchmark, giving a 95% confidence interval for the SDC of 5.00% .

Figures 9(a), 9(b), and 9(c) show the performance and reliability for each of the register configurations for that benchmark. Performance, as normalized execution time compared with a basepin execution, is on the Y-axis. Reliability, as percent of executions which resulted in failure, is on the X-axis. The most desirable position for a configuration to be is the lower-left corner, since lower means less performance cost and left means low failure percentage.

The three figures also have noted a line for the performance/ reliability frontier. The configurations on the frontier are strictly better, in terms of lower performance, higher reliability, or both, than all other points not on the frontier. When choosing the specific configuration to execute, the best options are those on the frontier.

```
mov (%edx), %eax          mov (%edx), %eax
                      1:  mov %eax  , %eax2
sub %eax, %ebx            sub %eax  , %ebx
                      2:  cmp %edx  , %edx2
                      3:  jne faultDetect
mov %ebx, (%edx)          mov %ebx  , (%edx)
```

(a) Original Code (b) Reliable Code

Figure 8: Fault detection, skipping protection for EBX. Code added by our technique is in bold. Checking instructions are italicized.

Figures 10(a), 10(b), and 10(c) show the normalized Mean Work To Failure for the respective benchmarks. Those figures show the basepin Pin and the baseline *Spot* normalized MWTF (as shown in Figure 6). The two new bars represent the best configuration for modulating fault detection with registers and with regions (as will be explained in Section 4.2). The three illustrative benchmarks representing the best, worst, and average responses to the *Spot* tool. Utilizing software modulated fault detection, *Spot* is able to achieve reliability near the baseline *Spot* executions but at near zero performance cost. Notice that for all three benchmarks, the normalized MWTF is 17.79x , 2.03x , and 1.80x . All benchmarks have a greater than one normalized MWTF, meaning it is advantageous to apply reliability.

Notice the two distinct clusters in Figure 9(c) for `300.twolf`. Those two clusters correspond to the EBP register. All in upper left region protect EBP, while all points in lower right do not. EBP is used very frequently in `300.twolf`, so protection of that register has corresponding cost. Protecting that register increase the performance from 1.0-1.9x to 2.0-3.0x.

Also notice in Figure 9(c) the knee in the frontier near 15% SDC. The unprotected version of `300.twolf` has an SDC percentage of 21.15%. Points on the configuration frontier have SDC percentages as low as 16.33% before incurring a noticeable performance cost. (16.33% SDC percentage for 8% executing time increase).

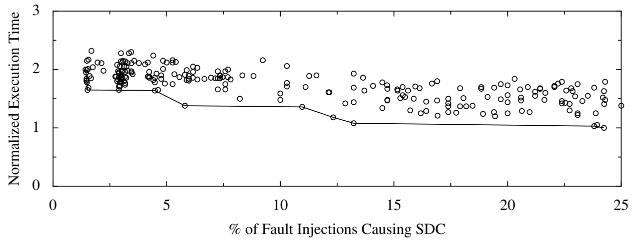
The point with the highest MWTF and represented in Figure 6 protects EAX, EBP, and ESP registers all of which, from Figure 7(c) have greater than average SDC percentage.

The other two benchmarks show similar trends in Figures 9(a) and 9(b). Both of these benchmarks have a long, mostly flat frontier very close to no performance cost until 10.5% SDC. For `254.gap`, one of the worst performing benchmarks, the performance cost sharply rises as SDC is further reduced, whereas for `181.mcf`, the benchmark with the best performance, the performance cost rises much more gradually. As can be seen in Figures 10(a) and 10(b), the normalized MWTF of the configuration techniques shows it is beneficial to apply the transformation.

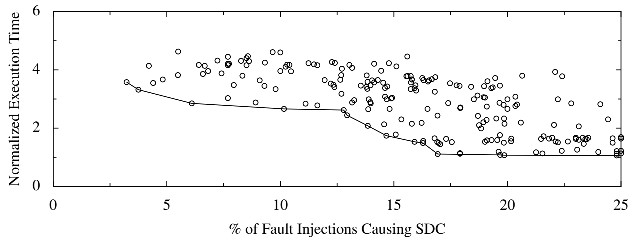
4.2 Varying Protection for Regions

Just as the natural resistance to failures varies for different registers, the natural fault resistance also varies when looking at different code regions of an application. Since different regions respond differently to faults, a non-uniform application of reliability can provide substantial benefits to reliability with reduced performance costs.

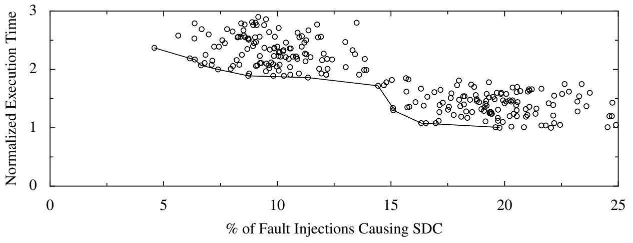
Figure 11(a), 11(b), and 11(c) show the percentage of faults injected into a code region which resulted in a program failure. The entire program was divided into regions of 256 bytes (an x86 instruction takes 1 - 7 bytes) and the faults that were injected were mapped back to the regions into which they were injected. If the first byte of the current instruction was within the range, the fault was considered injected in the range. The X-axis of the graphs show the code regions into which the fault was injected, and the Y-axis



(a) 181.mcf



(b) 254.gap



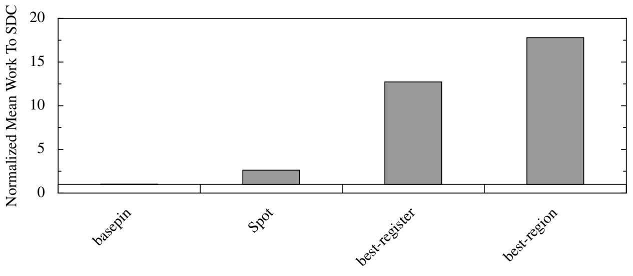
(c) 300.twolf

Figure 9: Performance and reliability when protecting different registers.

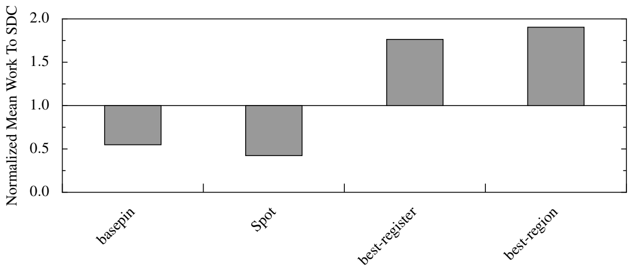
details the percentage of faults which resulted in a failure. The taller lines depict less reliable code regions. Similar to the depiction of the non-uniform per register, the dashed horizontal line represents the benchmark average SDC across all regions.

Those figures show a wide variety in the native percentage of SDC across the ranges of instructions executed. For example, in Figure 11(c) for `300.twolf`, there are a significant number of regions which are much less reliable than the average for that benchmark, 22.15%. In fact, there are 8 code regions which are more than twice as unreliable as the average for this benchmark. We can leverage this non-uniformity to optimize the tradeoff between protection and reliability.

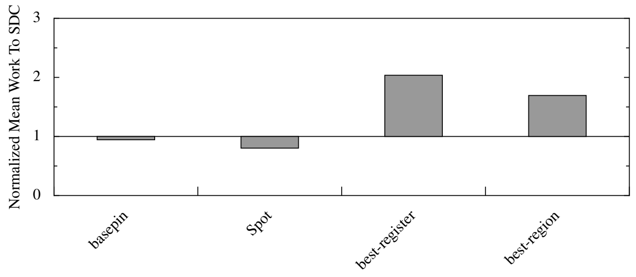
Spot can protect or skip protection for any set of address ranges. When doing the dynamic translation, *Spot* checks the set of regions it knows to add protection to (or skip protection for) in order to determine if the current region should be protected. Since a region with a certain level of reliability may transfer control to a region with a different level of reliability, *Spot* must ensure that the reconciliation code by copying the original version of the registers into their redundant versions. A fault during this phase will be propagated to both the original and the redundant versions. When transferring to



(a) 181.mcf

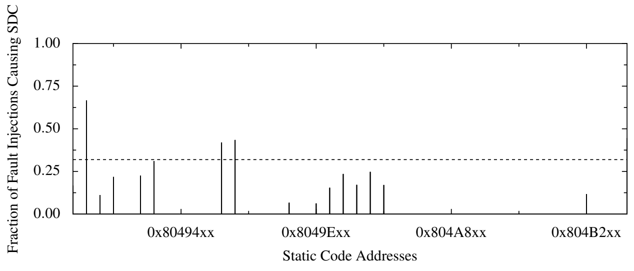


(b) 254.gap

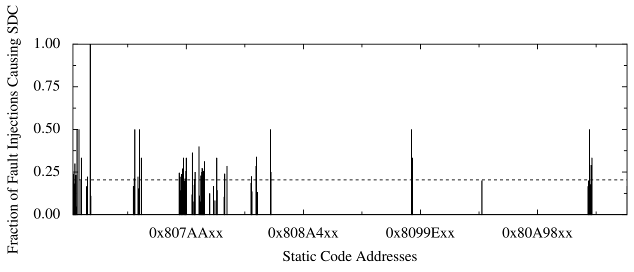


(c) 300.twolf

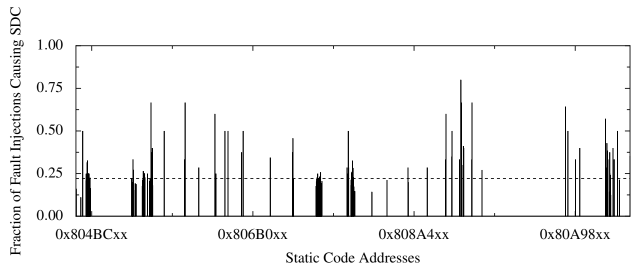
Figure 10: MWTF incorporating software modulated fault detection.



(a) 181.mcf



(b) 254.gap



(c) 300.twolf

Figure 11: Percentages of faults causing SDC for each region.

a region of the same type (reliable to reliable or unreliable to unreliable) no reconciliation code is necessary. When a reliable region transfers to an unreliable region, the unreliable region will simply ignore the redundant registers; in this case there is also no necessary reconciliation code.

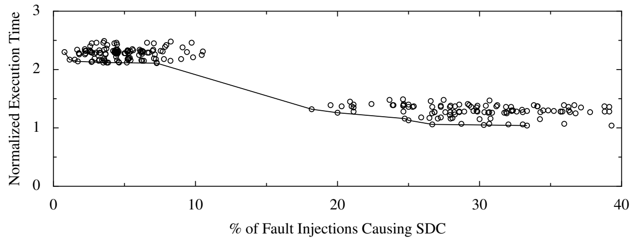
Like the evaluation of register modulation, *Spot* was run in 256 configurations of different regions for each benchmark. We selected the 7 regions with the worst reliability and computed the reliability and performance when protecting the entire application except the current combination of regions selected for this experiment. Each combination of the 7 regions were evaluated for a total of 128 configurations. Since the selected regions did not cover the entire address space of executed instructions, we also computed the performance and reliability when protecting the inverse of each configuration; *Spot* was configured to protect none of the application except for the regions selected. We also evaluated each of the 128 inverse configurations for a total of 256 region-based configurations.

We evaluated the performance and reliability in the same manner as the rest of the paper. For confidence in the reliability percentages, each of the 256 configuration for each benchmark was injected with 500 faults for a total of 384,000 fault injections.

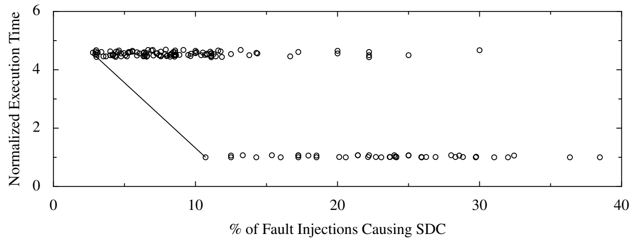
Figures 12(a), 12(b), and 12(c) show the performance and reliability for the various regions of the three illustrative benchmarks. Each graph shows two distinct clusters. The regions chosen were based on their impact on the SDC percentage rather than their impact on performance. The two clusters represent protecting the entire application except the subset of the regions (upper left clusters), and protecting none of the application except the subset of the regions (lower right clusters). The regions selected do not have a large impact on performance. This causes the clusters to be very flat, one near 1.0 (no performance cost) and one near the baseline *Spot* performance cost.

The wide range in the X-axis (SDC) is due to protecting or leaving unprotected the vulnerable regions of the application. Figure 12(c) shows that the best reliability for `300.twolf` is the configuration that has no performance cost (at a normalized execution time of 1.0 and 17.5% SDC).

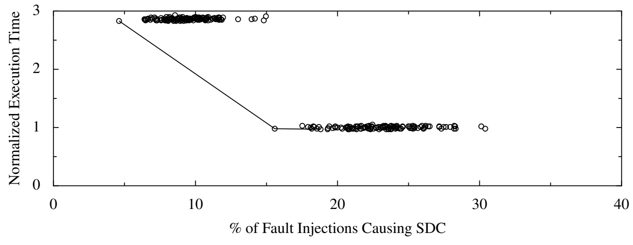
Just as for the register analysis, the best configuration points are shown on the plot with a solid frontier line. Figure 12(c) has a frontier boundary that encompasses only two main points, one for the cluster with no performance cost, and the other point for the cluster with baseline *Spot* performance. The frontier actually encompasses 3 points in the high performance cost cluster, but they are extremely close together (the variation in MWTF is less than 0.08). Although



(a) 181.mcf



(b) 254.gap



(c) 300.twolf

Figure 12: Performance and reliability when protecting different regions.

these regions only provide a binary decision as to the best configurations to consider, the lower right points have the same execution time as the baseline, but at $\frac{3}{4}$ the SDC percentage (a reduction of 4.5%). For this benchmark, as shown in Figure 10(c), the MWTF for the best region configuration is 1.60x the native application, but slightly less reliable that the MWTF for the register configurations (2.03x).

For the other benchmarks shown in detail there are also two distinct clusters. While the other two benchmarks only have 2 points on their frontiers, `181.mcf` has thirteen lines along its frontier line. The region configuration MWTF is more reliable and better performing than the register configuration MWTF. Notice in Figures 10(a) and 10(b) that the MWTF of the region configurations for `181.mcf` and `254.gap` are 17.79x and 1.90x respectively, whereas the MWTF for register configurations are 12.72x and 1.76x respectively.

While these regions provide a large range in SDC options and provide beneficial MWTF, they do not provide a wide range of performance options. The regions were chosen based on their SDC region ranking, but regions based on frequency of execution or percentage of execution time could be selected. This would give a wide range in execution time, and the more reliable configuration within acceptable performance loss can be selected.

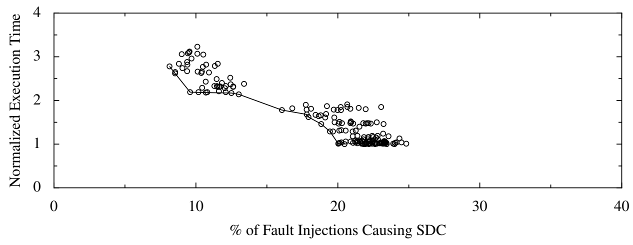


Figure 13: Performance and reliability when protecting different regions and different registers (300.twolf).

4.3 Varying Protection for Regions & Registers

In Sections 4.1 and 4.2 we showed two different fine-grained configurations for the tradeoff between reliability and performance. These configurations are not mutually exclusive; they can be combined for an even wider range of reliability and performance options.

Figure 13 shows the combinations of the 3 most unreliable registers and 4 most unreliable regions in all possible configurations (256 in all, when including the inverse of the region protection). We injected 500 faults for each experiment to determine the unACE, DUE, and SDC percentages.

Unsurprisingly, the plot of SDC and performance for `300.twolf` when using configurations protecting regions and registers resembles the two configuration plots, Figures 9(c) and 12(c), combined. The plot is segmented into two clusters, but there is less variation within a cluster relative to the register-only plot in Figure 9(c) due to the influence of different region configurations. Conversely, there is more variation within a cluster compared to the region-only plot in Figure 12(c) due to different register configurations.

5. CONCLUSION

As transient faults become more prevalent across a wide range of markets, techniques which can tailor the level of protection to each user's specific performance and reliability requirements will be needed. The technique presented in this paper, called *Spot*, is the first software-only fault-detection technique to address this need by providing software-modulated fault tolerance at fine granularities. *Spot* is also the first technique to use dynamic binary translation to provide full instruction protection, allowing users to dramatically improve the reliability of their applications without relying on hardware modifications or access to application source code. *Spot*'s ability to vary the level of protection for different registers and regions of code provides users with more, and often superior, fault detection options, allowing it to increase the mean work to failure from 1.90x to 17.79x.

6. REFERENCES

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

[2] E. Borin, C. Wang, Y. Wu, and G. Araujo. Software-Based Transparent and Comprehensive Control-Flow Error Detection. *Code Generation and Optimization, 2006. CGO 2006. International Symposium on*, pages 333–345, 2006.

[3] M. A. Gomaa and T. N. Vijaykumar. Opportunistic transient-fault detection. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 172–183, 2005.

[4] K. W. Harris. Asymmetries in soft-error rates in a large cluster system. *IEEE Transactions on Device and Materials Reliability*, 5(3):336–342, September 2005.

[5] X. Li, S. Adve, P. Bose, and J. Rivers. SoftArch: An Architecture Level Tool for Modeling and Analyzing Soft Errors. *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 496–505, 2005.

[6] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2005.

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

[8] G. Memik, M. Kandemir, and O. Ozturk. Increasing Register File Immunity to Transient Errors. *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*, pages 586–591, 2005.

[9] S. E. Michalak, K. W. Harris, N. W. Hengartner, B. E. Takala, and S. A. Wender. Predicting the number of fatal soft errors in los alamos national labratory's ASC Q computer. *IEEE Transactions on Device and Materials Reliability*, 5(3):329–335, September 2005.

[10] S. S. Mukherjee, J. Emer, and S. K. Reinhardt. The soft error problem: an architectural perspective. In *Proceedings of the 11th International Conference on High-Performance Computer Architecture*, pages 243–247, 2005.

[11] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 99–110. IEEE Computer Society, 2002.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] V. J. Reddi. Deploying dynamic code transformation in modern computing environments. Master's thesis, Department of Electrical and Computer Engineering, University of Colorado, Boulder, CO, November 2005.

[18] V. J. Reddi, D. A. Connors, and R. S. Cohn. Persistence in dynamic code transformation systems. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, September 2005.

[19] 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.

[20] 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.

[21] 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.

[22] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee. Software-controlled fault tolerance. In *ACM Transactions on Architecture and Code Optimization (TACO)*, volume 2, December 2005.

[23] J. Segura and C. F. Hawkins. *CMOS Electronics: How It Works, How It Fails*. Wiley-IEEE Press, April 2004.

[24] 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.

[25] 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.

[26] 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.

[27] A. Wood. Data integrity concepts, features, and technology. White Paper, Tandem Division, Compaq Computer Corporation, 1999.

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

[29] J. F. Ziegler and H. Puchner. *SER - History, Trends, and Challenges: A Guide for Designing with Memory ICs*. Cypress Semiconductor Corporation, 2004.