Runtime Asynchronous Fault Tolerance via Speculation

Yun Zhang Soumyadeep Ghosh Jialu Huang

Jae W. Lee[‡] Scott A. Mahlke[†] David I. August

Princeton University, Princeton, New Jersey, USA
[§] SungKyunKwan University, Suwon, Korea
[†] University of Michigan, Ann Arbor, Michigan, USA

ABSTRACT

Transient faults are emerging as a critical reliability concern in modern microprocessors. Redundant hardware solutions are commonly deployed to detect transient faults, but they are less flexible and cost-effective than software solutions. However, software solutions are rendered impractical because of high performance overheads. To address this problem, this paper presents Runtime Asynchronous Fault Tolerance via Speculation (RAFT), the fastest transient fault detection technique known to date. Serving as a layer between the application and the underlying platform, RAFT automatically generates two symmetric program instances from a program binary. It detects transient faults in a non-invasive way and exploits high-confidence value speculation to achieve low runtime overhead. Evaluation on a commodity multicore system demonstrates that RAFT delivers a geomean performance overhead of 2.83% on a set of 30 SPEC CPU benchmarks and STAMP benchmarks. Compared with existing transient fault detection techniques, RAFT exhibits the best performance and fault coverage, without requiring any change to the hardware or the software applications.

1. INTRODUCTION

Transient faults, also known as soft errors, are caused by external events such as particle strikes [3, 21, 25, 30]. These faults may lead to program crash or system failure, without leaving any trace. A combination of exponentially growing transistor counts and voltage scaling makes transient faults a critical concern for the semiconductor industry. Oracle America Inc. acknowledges that clients including America Online (AOL), eBay and Los Alamos National Labs have suffered from system failures due to transient faults [4, 18]. A recent study shows that a BlueGene/L machine with 104 nodes deployed in Lawrence Livermore National Labs experiences soft errors once every four hours [8]. Given that the reliability per bit is estimated to drop 8% per generation of processors [6], it is critical to ensure fast and effective transient fault tolerance on modern and future architectures.

Recently proposed transient fault detection techniques rely on redundant execution in either hardware or software. Specialized redundant hardware is commonly employed to detect transient faults transparently. For example, IBM S/390 [33], Boeing 777 airplanes [41], and HP's Non-stop [13] all use redundant hardware for fault tolerance. However, these solutions require specialized hardware components and additional verification cost [2, 33, 29]. Moreover, hardware solutions cannot adapt to changes in deployment environment or scope of protection.

Current architectural trends toward multicore microprocessors naturally provide additional resources, making software redundant execution more viable than ever. Existing software proposals [22, 26, 31, 37, 42] typically insert redundant code into a program at compile time or runtime, and check for transient faults at runtime. Among these proposals, compiler-based techniques [26, 31, 37, 42] are only applicable to programs whose source codes are available. Separately compiled modules, such as libraries, cannot be protected using compiler-based techniques due to the absence of source code at compile time. Runtime techniques, such as PLR [31], use dynamic instrumentation to duplicate program execution at the process level and instrument binaries for fault detection. This approach still has high performance overhead due to the cost of dynamic binary instrumentation and barrier synchronizations at every system call.

To address the performance and applicability issues of software fault detection techniques, this paper presents RAFT, a Runtime Asynchronous Fault Tolerance technique that detects transient faults with low overhead. RAFT serves as a light-weight virtual layer between an application and the underlying platform. It takes a program binary as input and duplicates its execution automatically. During execution, it monitors both original and duplicated program instances' behavior at the system call level using a process monitoring utility provided by the operating system. The arguments of system calls from both instances are compared for equality. A value mismatch means a transient fault has occurred and RAFT reports this to the user. Unlike compiler-based techniques that must obtain knowledge of library functions for fault detection, RAFT must only understand the relatively stable and well-defined set of system calls.

The key insight behind RAFT is that redundant execution can be accelerated by *speculatively* removing data dependences. Whenever possible, RAFT allows the process that first invokes a system call to continue execution with a speculated return value, without executing the call. When the other process invokes the same system call, RAFT compares the arguments of the two invocations to check for transient faults. If the arguments mismatch, RAFT reports transient faults and stops program execution. If no fault occurred, the system call is executed and its return value is checked against the specu-

lated one. If these two values differ, a misspeculation occurs, and RAFT uses a fast misspeculation recovery scheme using copy-on-write mechanism to continue program execution. With value speculation, most barrier synchronizations required by prior approaches are eliminated, leading to much lower performance overhead.

The contribution of this paper is the design and implementation of a runtime speculative fault tolerance technique named RAFT. RAFT provides the fastest transient fault detection known to date with full transient fault coverage. Evaluation shows that RAFT delivers a geomean overhead of 2.83% for 30 SPEC CPU benchmarks, 5 times faster than the best available software fault tolerance technique. The implementation in this paper does not support multi-threaded programs. However, RAFT can be extended to protect multi-threaded applications that have deterministic outputs.

Applying RAFT to applications with non-deterministic outputs may result in false alarms. Research on redefining faulty behavior of such applications and extend RAFT to support them is a future work of this paper.

2. MOTIVATION

Many existing solutions detect transient faults during program execution via redundant computation. The sphere of replication (SoR) [24] is defined as the scope of fault coverage and values requiring special handling. Values that enter the SoR must be replicated for redundancy and values that exit the SoR must be checked for faults to ensure their correctness. Table 1 lists several existing representative fault detection techniques.

Hardware techniques rely on duplicated hardware modules, and provide protection for the processor core. Rotenberg's AR-SMT [28] uses an 8-way simultaneous multi-threading trace processor for detecting transient faults in processors. Simultaneous Redundant Threading (SRT) [24] and Chip-level Redundant Threading (CRT) [19] exploit simultaneous multi-threaded processors and multiple cores respectively for redundant execution and value checking. These techniques use duplicate hardware modules, and check values when they escape the SoR for fault detection. In addition to using more chip area and paying extra chip design and verification cost, the hardware techniques do not have the flexibility of changing which modules to duplicate after deployment. ECC memory, as a hardware technique for memory transient fault tolerance, can help protect executions against memory faults with some probability, but may fail on multiple-bit flip events and is too expensive to apply to processor cores.

On the contrary, software redundant execution approaches are more cost-effective and more flexible [5, 7, 26, 31, 37, 42]. Software transient fault detection techniques typically fall into three categories: thread-local duplication, redundant multi-threading, and process-based redundancy, as shown in Table 1. Thread-local duplication techniques such as EDDI [22] and SWIFT [26] redundantly execute instructions within a single thread, exploiting instruction level parallelism to improve performance. Shoestring [11] performs selective instruction duplication to achieve lower overhead than both EDDI and SWIFT, but at the cost of lower fault coverage. Redundant multi-threading techniques (e.g. SRMT [37] and DAFT [42]) use multiple threads to execute program codes redundantly. Process-based redundant techniques (e.g. PLR [31]) use multiple processes instead of threads, at the cost of maintaining multiple memory states.

All these techniques are typically implemented using either compiler transformations or runtime systems. Compiler-based transient fault detection techniques, such as EDDI [22], SWIFT [26], SRMT [37], DAFT [42], and Shoestring [11], all require program source code for recompilation, and cannot detect any transient fault occurring in separately compiled modules. Software redundant multi-threading [37] uses multiple threads to run redundant copies of a program. These techniques cannot issue redundant store instructions because only one shared memory state is maintained. Before a memory operation is executed, its operands are communicated between threads and checked for consistency. Consequently, frequent barrier synchronization is required and adds significant performance cost. DAFT [42] improves the performance of SRMT by allowing one thread to execute the memory operations without waiting for confirmation from the other thread. Removing barrier synchronizations, combined with decoupled execution, helps DAFT to reduce the overhead of fault detection from 200% to 38%.

Compared with compiler-based techniques, runtime techniques do not require source code recompilation and can protect separately-compiled modules. One such implementation called PLR [31], a dynamic instrumentation technique, provides transient fault detection with the minimum runtime overhead (16.9%) among all software solutions with full coverage. This technique duplicates the original program into several instances at runtime, maintaining one private memory space for each instance. Only externally visible values need to be verified before they escape user space.

PLR synchronizes the main and the redundant processes for transient fault detection, when any value escapes user space to the kernel. The main process executes the system call. The redundant process resumes execution only after the system call is completed. This barrier synchronization puts inter-core communication on the critical path of program execution, leading to slower performance.

This paper proposes RAFT, a runtime speculative transient fault framework. Like PLR, RAFT applies to program binaries and only verifies values that escape user space. However, RAFT eliminates frequent barrier synchronization via value speculation. RAFT speculates the return values of system calls to allow the leading process to continue execution past the system call. In addition, RAFT provides wider SoR, by memory duplication at runtime.

Figure 1 shows a simplified code example from genome, a STAMP benchmark. Figure 2 compares PLR and RAFT by showing their execution plan using this example code. This execution plan demonstrates that barrier synchronizations add considerable runtime overhead to the program. Although both process instances are executing the same program binary, the cycles spent on executing each piece of code, such as `B1, B2, ...`, are not the same because of various runtime factors such as cache behavior and process scheduling. Forcing barrier synchronization at every system call requires all processes to wait for the slowest one to reach the synchronization point, consequently slowing down the overall program execution. In contrast, RAFT allows one process to speculate the return values of the `sys_write` system call without actually executing it, therefore does not require waiting till the other process to invoke the same system call and the barrier synchronization between the processes. If a misspeculation occurs, an efficient misspeculation recovery scheme is employed to continue execution from a previous correct program state. Combining all the features above, RAFT achieves very low runtime overhead with full fault coverage.

Duplicating both the register and memory states of the program allows RAFT to provide transient fault detection for transient faults in both processor core and memory subsystems. Compared with previous work, RAFT yields the lowest performance overhead for transient fault detection without compromising fault coverage.

Approach	Technique	Main Memory Usage	Need Source Code	No. of HW Execution Units	No. of Software Contexts	Sphere of Replication		Reported Overhead
						Processor	Memory	
Specialized Hardware	SWAT [17]	1×	No	2 cores	2 threads	Most	None	5%
	AR-SMT [28]	1×	No	8-way trace proc.	8 threads	All	None	16.7%
	CRT [19]	1×	No	2 cores	2 threads	Most	None	Unreported
	SRT [24]	1×	No	2-way SMT proc.	2 threads	Most	None	Unreported
	Hybrid-SRMT [37]	1×	Yes	2 procs.	2 threads	Most	None	19%
Thread-local Duplication	EDDI [22]	2×	Yes	1 proc.	1 thread	Most	All	52.2%
	SWIFT [26]	1×	Yes	1 proc.	1 thread	Most	None	45%
	Shoestring [11]	1×	Yes	1 proc.	1 thread	Most	None	15.8%
Redundant Multi-Threading	SRMT [37]	1×	Yes	2 procs.	2 threads	Most	None	400%
	DAFT [42]	1×	Yes	2 cores	2 threads	Most	None	38%
Process-based Redundancy	PLR [31]	2×	No	4-way SMP	2 processes	Most	None	16.9%
	RAFT[This paper]	2×	No	2 cores	2 processes	Most	All	2.83%

Table 1: Comparison Among Transient Fault Detection Techniques.

```
A: for ( i = 0; i <= vector_size; i++) {
B:          char * charPtr = computePtr(Ptr);
C:          j = compute( charPtr );
D:          printf("Segment %li (@%li) = %s \n",
                        i, j,  vector(Ptr, i) );
}
```

```
  int printf(const char *format, …) {
E:   …
       syscall(sys_write);
F:  …
  }
```

Figure 1: Simplified Code Example from STAMP Benchmark genome

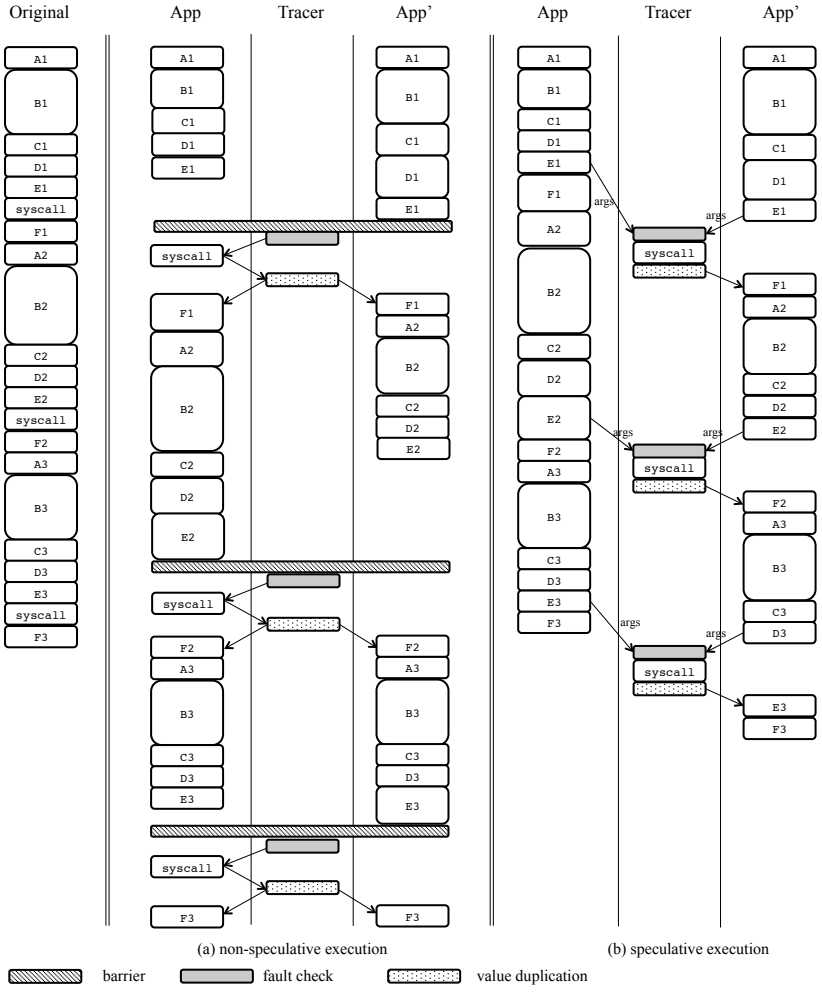

Figure 2: Execution plan of transient fault detection with and without speculation for the program in Figure 1 with `vector_size` = 3. The execution time of the same instruction blocks, such as B1, B2, are not the same across both processes is because of various runtime factors, such as cache behavior and process scheduling.

3. Runtime Asynchronous Fault Tolerance via Speculation

RAFT is a runtime system implemented using an OS-level process monitoring tool. The `ptrace` utility is a POSIX standard that is provided by Linux/Unix, Mac OS, and Solaris systems to provide process monitoring and debugging capabilities. This monitoring utility is exploited in RAFT as a method of trapping system calls and comparing the values of their arguments to detect transient faults. This approach ensures that RAFT can transparently detect transient faults occurring in a non-invasive way.

3.1 Overview

RAFT utilizes a process monitoring utility (`ptrace`) to intercept system calls invoked by a program. As a light-weight interface between the operating system and user applications, `ptrace` adds very little runtime overhead. Figure 3 demonstrates the overall structure of RAFT and the interaction between several components of the system. RAFT first takes the program binary and its input, then spawns one process that execute the binary redundantly. Upon process creation, RAFT immediately pauses that process, and injects a `fork` system call into the just-created child process `App`. The child process `App` spawns another process `App'` from itself, inheriting all its virtual address table and signal handling table. This is critical to eliminate potential false-positives, especially on systems with address space layout randomization (ASLR) enabled. From then on, `App` gives up its parentship of `App'` to the tracer process. Both `App` and `App'` become processes that are traced only by the tracer process RAFT.

During the execution, RAFT serves as a virtual layer between the application and the underlying OS services and devices. It traps every system call invoked by either program instance. Section 3.2 provides details on system call trapping. After a system call is trapped, the calling thread's register file is examined to find out the type of the system call. RAFT then compares the system call's arguments against those in the other program instance to check for transient faults, if available, according to its specific calling context. If the system call reads a process' memory space through pointer arguments, the memory content is also checked. If no transient fault is detected, RAFT executes the system call and lets the program instance continue execution. This runtime system predicts the results of system calls with high confidence, which allows speculative execution of the program. Section 3.3 introduces the misspeculation detection and recovery mechanism in RAFT. If the result of a system call varies from call to call, RAFT skips speculation and executes them in a synchronous fashion to avoid high overhead of logging and rollbacks. When misspeculation is detected after a system call is completed, misspeculation recovery schemes are employed to ensure continuous correct execution of both program instances.

3.2 System Call Trapping

RAFT detects transient faults at system call interface level. The idea is to intercept all system calls initialized from two identical program instances. There is a fixed set of well-defined system calls provided by the operating system to applications. For modern Linux operating systems, a total of 298 system calls are defined. It is reasonable to specify customized system call arguments verification for each system call.

Despite the redundant execution of program binaries, side-effecting work should only happen once. For example, a `printf` (which transitively invokes a `write` system call) cannot be redundantly executed. Based on whether the results of the system call can be speculated and whether the system call needs to be executed in both copies of the program instances, all system calls are classified into three categories. RAFT handles system calls differently depending on which category they belong to.

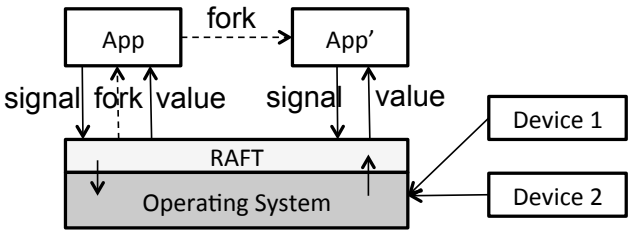

Figure 3: RAFT Structure Overview

- *Synchronous*: This category includes system calls that have no predictable program state after system call execution. Such system calls cannot be executed redundantly. Some read instructions, such as destructive read from standard input or other devices, can only be executed once. After the read, the content will be destroyed and not be available for a redundant read. Wrong argument of these calls will cause either device error, or un-recoverable program states. Therefore, these calls cannot be speculated and must only be executed after its arguments are verified to be fault free. For example, `sys_read` and `sys_lseek` are such system calls. We selected out these system calls out and execute them in a synchornous fashion, to avoid heavy logging and re-execution. Experiments on benchmarks show that this kind of system call accounts for <1% of total number of system calls issued at runtime.
- *Asynchronous duplicate*: Some system calls have no predictable return values, but their return values must be duplicated in both program instances to ensure correct program execution, such as `sys_times`. These system calls need to have identical return values in both program instances, but barrier synchronization is not necessary. RAFT allows the first instance that invokes the call to execute the call and keeps a record of its return values. When the other instance invokes the same call, RAFT compares their arguments, then ensures the call to return the same value for the second instance. About 80% of system calls issues at runtime belong to this category.
- *Asynchronous single*: This category includes system calls with return values that can be speculated with high confidence, but cannot be executed redundantly, e.g. `sys_write`. Among all system calls invoked dynamically on SPEC benchmark programs, 80.66% calls belong to this category. RAFT does not require barrier synchronization for these system calls. If one copy of the process invokes `sys_write` ahead of the other process, it can continue executing the rest of the code without committing the write. The system call's return value is speculated and used in the process' later computation. When the other process invokes the same system call, RAFT compares the arguments of these two system calls for verification. If any fault occurs, RAFT is able to report the fault to users before the write is committed.

3.3 Misspeculation Detection and Recovery

Misspeculation occurs when the results of a system call differ from what RAFT predicts. The speculatively executing program instance must be cancelled if RAFT made a wrong prediction. RAFT features a misspeculation recovery scheme to restart program execution from the point of misspeculation.

In order to efficiently record program state, RAFT exploits the copy-on-write semantics of the `fork` system call. If the tracing process detects misspeculation, it kills the speculative process and duplicates the non-speculative process for later computation. The copy-on-write semantics have previously been used to implement speculative systems [23].

To inject a `fork` system call into the non-speculative process, RAFT follows the algorithm described in Algorithm 1. In the following algorithm, `W1` refers to the first invocation of the program binary, `W2` refers to its replicate, and `W` refers to either one of the worker processes (`W1` or `W2`).

Algorithm 1: Misspeculation Detection and Recovery

```
repeat
    intercepts system call sys from a traced process W(either W1 or W2)
    // Code for fault detection
    if sys's return value was speculated in W2 then
        // symmetric if W1 is the speculative process
        if W1.Regs.rax != W2.specvalue[sys] then
            kill(W2.pid, SIGKILL)
            // duplicate the non-speculative process W2
            make a copy of the register file of W2
            W1.Regs.rax = SYS_fork
            // the instruction SYSCALL on x86_64 is 0F 05,
            // which takes 2 bytes. This number is architecture dependent.
            W1.Regs.rip = W1.Regs.rip-2
            signal W1 to continue the system call
            wait for signals from kernel, upon returning from SYS_fork
            W2.pid = return value
            copy back original register file of W2
            continue executing the program
        end if
    end if
until W1 and W2 both exit
```

For the code example in Figure 1, RAFT speculates the number of bytes being transmitted by system call `sys_write` and allows the process arriving first to proceed without waiting for the other process. When the other process calls `sys_write`, RAFT executes the call and then compares the return value with the speculated one. If the two values are the same, both processes proceed normally. Otherwise, RAFT sends a signal to kill the misspeculated process. Another signal is sent to the program instance with correct values to force a process duplication. A new process is then spawned, and continues the rest of program execution redundantly with the correct program state. Evaluation shows that one misspeculation recovery only costs a couple of millisecond at runtime.

3.4 Virtual Memory Space Synchronization

To compare execution of two processes, the original one and its redundant copy, the fault detection may have to compare virtual memory addresses at some system call. This is crucial to ensure identical memory layout of all processes to eliminate false-positives. As a result, some system calls, such as `sys_mmap`, must be executed in all copies of processes for a program to continue. Additionally, these two processes must have identical virtual address table to start with. In systems with address space randomization (ASR), independent processes executing the same binary can have randomized virtual addresses.

One solution to this problem is to disable address space randomization entirely by configuring the operating system. However, this also prevents other applications, especially security sensitive applications, from benefiting from ASR.

RAFT solves this problem by allowing the first process that issues `sys_mmap` to proceed without waiting for the other process. Upon returning from the system call, RAFT keeps a record of the virtual memory address that this system call allocates. When the other process invokes `sys_mmap`, RAFT first compares their arguments, then forces this call to map to the same virtual address as the first process, by setting the MAP_FORCED flag. Because both processes maintain the same memory layout, both processes should be in the same program state before and after the system call.

There are also cases when `sys_mmap` intends to map files to a process' virtual memory space and performs read and write operations. By intercepting `mmap` system call, RAFT can identify such requests and map that file in RAFT's own memory space. RAFT then returns a protected page address to the two program instances. When these programs intend to access the protected page, a signal is sent to and trapped in RAFT. At that point, RAFT checks for transient faults, then performs the actual read and write access only once on the memory mapped file.

3.5 Signal Handling

In RAFT, signals sent to the program and signals raised from one copy of the program must be handled in a way that is transparent to the user. There can be two kinds of signals, as described below.

Internal Signal: Some transient faults may flip a bit in a register that holds a memory address the program loads from. Since the two identical programs do not share memory space, memory operations are executed without being verified. Consequently, a transient fault may cause a segmentation fault raised by one of the processes. Similarly, other exceptions, such as divide-by-zero, may also occur as a result of a transient fault. To distinguish a transient fault from a normal internal signal raised by the program, such as signals caused by program bugs, RAFT traps all signals raised by both processes. If the two processes both raise the same internal signal, it is a normal program signal. Otherwise, RAFT alerts the users the existence of a transient fault.

External Signal: As a transparent transient fault detection technique, RAFT maintains the original deterministic behavior of the original program. External signals may cause non-determinism among the two copies of programs. For example, the user may press `ctrl-c` from command line, which sends a SIGINT signal to the program. RAFT should make sure the two program copies behave as if only one program is running and abort the program.

To achieve this, RAFT registers special signal handlers for all external signals. Specialized signal handlers are registered at the beginning of the program. When an external signal is received, the corresponding signal handler is called and proper actions are taken. For example, in the case of a SIGINT sending to the program from command line, RAFT communicates SIGINT to both processes, terminates their execution and kills itself as well.

3.6 Automatic Memory Page Walking

To protect program execution from transient faults in memory subsystems, RAFT adopts process-level duplication and uses redundant memory space. However, two processes, one forked from the other, share the same physical memory page, if no value is written to that page. As a result, memory faults occurring in pages that are

shared by the two processes may be read by both program copies hence introducing faulty values.

RAFT solves this problem via automatic memory page walking. The idea is to check the two processes' physical page table periodically. If any of the pages that are physically in memory has the same physical page address for both processes, RAFT automatically loads a word from the page, and immediately writes the same value back, utilizing the copy-on-write feature of forked processes. The operating system will automatically create another physical page as a tainted copy.

3.7 Window of Vulnerability

There are several cases where RAFT-protected program is still vulnerable to transient faults. One case is that faults occurring in RAFT itself may cause an unrecoverable error or erroneous results or undefined program behavior. Although RAFT itself represents a single point of failure, it only occupies the CPU for a very short duration and only consumes a few clock cycles. Typically the RAFT process only takes less than 0.01% CPU time throughout program execution. The probability of transient faults occurring in registers while RAFT code is running is extremely low.

If a transient fault hits the main memory that holds data used in the tracing process, it may lead to program crash or wrong output. However, such faults may still be benign faults or be detected via transient fault detection. For example, if the fault flips a bit of a word in memory, which is overwritten before the value is loaded, the fault is a benign fault. If a memory fault changes a value stored in memory and results in a value mismatch in fault detection code, it can also be detected.

Besides, RAFT does not protect the operating system and its services executed in kernel. Transient faults occurring in the kernel code may still cause program failure or erroneous program output. For OS intensive programs, all application level fault tolerance techniques, including RAFT, fail to provide coverage for the kernel code.

4. EVALUATION

This section presents an evaluation of RAFT on a set of 30 optimized benchmarks from SPEC2000, SPEC2006 and STAMP benchmark suites. The program binaries for all the benchmarks are generated using gcc 4.5.2 with optimization option `-O2`. The benchmarks are selected from a repository of benchmark suites, based on availability. The experiments are performed on a 2.4 GHz Intel Core 2 Quad-Core (Q6600) machine running Linux 2.6.38, with two 4MB L2 caches.

4.1 Benchmark Characteristics

The benchmark programs used for evaluation are from SPEC CPU 2000, SPEC CPU 2006, and STAMP benchmark suites. The reasoning behind the benchmark selection is: (a) All previous software-only fault tolerance techniques used SPEC benchmarks for evaluation. We used these programs for fair comparison with previous work; (b) SPEC benchmarks were selected as representatives of real-world applications, including general-purpose applications and scientific applications. For example, 433.milc is developed by the MIMD Lattice Computation (MILC) collaboration for doing simulations of four dimensional SU(3) lattice gauge theory on MIMD parallel machines. This code is used for millions of node hours at DOE and NSF supercomputer centers; (c) The STAMP benchmark suite consists of real world applications that are deployed for everyday use.

As the implementation of RAFT in this paper does not support multi-threaded programs, evaluation on the STAMP benchmark programs used the single-threaded mode, with the biggest input sets.

4.2 Window of Vulnerability

This paper considers transient faults in register files, physical memory, and control-logic. Table 2 shows various types of transient faults and the way they are modelled in this paper. All reliability evaluation demonstrated in this paper injects faults into *all three processes* (two workers and one tracer). This fault injection method does not inject faults into the operating system itself. However, the time spent in the tracer process includes the time that all system calls spent in the operating system.
To study the window of vulnerability of RAFT, a process tracing tool (`ptrace`) was used to simulate single-event-upset (SEU) [24, 27, 36] transient faults at runtime in both register files and the memory (in a way similar to [14, 32, 15]). Figure 4 shows the fault distribution of injected faults. The horizontal axis shows the faults injected into register files (R) or memory (M) for each benchmark program. The vertical axis is the percentage of faults in each category.

Injected faults are categorized into four groups based on the outcome of the program: (1) Benign faults; (2) Detected; (3) Timeout; and (4) Silent Data Corruption (SDC). An injected fault is a benign fault if it does not affect program execution or change the program output. RAFT does not detect benign faults, since the program exits normally in this case. RAFT can detect injected non-benign faults through redundant computation and value checking. This kind of fault is classified as *Detected*. A timeout scheme is used to detect faults that cause the program to freeze or loop forever. A $scale$ is predefined (set as 10 in this paper) and an estimated execution time ($ExecutionTime$) of the program is set from a profiling run. If the program takes more than $scale \times ExecutionTime$ to finish, our fault injection tool aborts the program and reports *Timeout* as an indication that a transient fault happened. If the program exits normally with incorrect output (compared with standard output) and RAFT fails to report transient faults to the user, this case is classified as *Silent Data Corruption* (SDC).

4.2.1 Register Fault Simulation

The R columns in Figure 4 shows the experimental results from injecting faults into the register file at program runtime. First, a profile run of the original program binary is timed to estimate how long it may take to execute the program. Before fault injection, our tool randomly selects one point in time, one random program instance, one random bit of a register, as well as one random register among general-purpose, floating point, XMM, and flag registers. During program runtime, this fault injection simulation issues an alarm after a random period of time. It then sends a signal to the randomly selected process, stops its execution, and flips the random bit of the selected register. The particular program instance then continues execution. In our evaluation, we inject faults into all three processes: the tracer process and two redundant program instances, proportional to the amount of CPU time consumed during runtime. Since the monitoring process spends very little CPU time (CPU utilization is lower than 0.01%), it is extremely unlikely that a transient fault occurs while the monitoring process is executing. It must also be noted that this fault injection method does not inject faults into the operating system. Usually, an unloaded system (not executing anything but the operating system itself) consumes about 0.1% CPU time on average. Finally, the execution result of

Reason	Symptom	Modelling
Instruction	instruction corruption	Modify the value in a random memory address that maps to the program binary
Physical Memory	instruction corruption or data corruption	Modify random memory address that may contain instruction or data
Memory Bus	instruction fetch error or data corruption	Modify random memory address that may contain instruction or data
Register File	data corruption	Flip bits in a random general-purpose, XMM, floating point, or register
Program Counter	incorrect program flow	Bit flip in the PC register
Control Logic	PC value corruption	Bit flip in the PC register

Table 2: Transient fault types and modelling

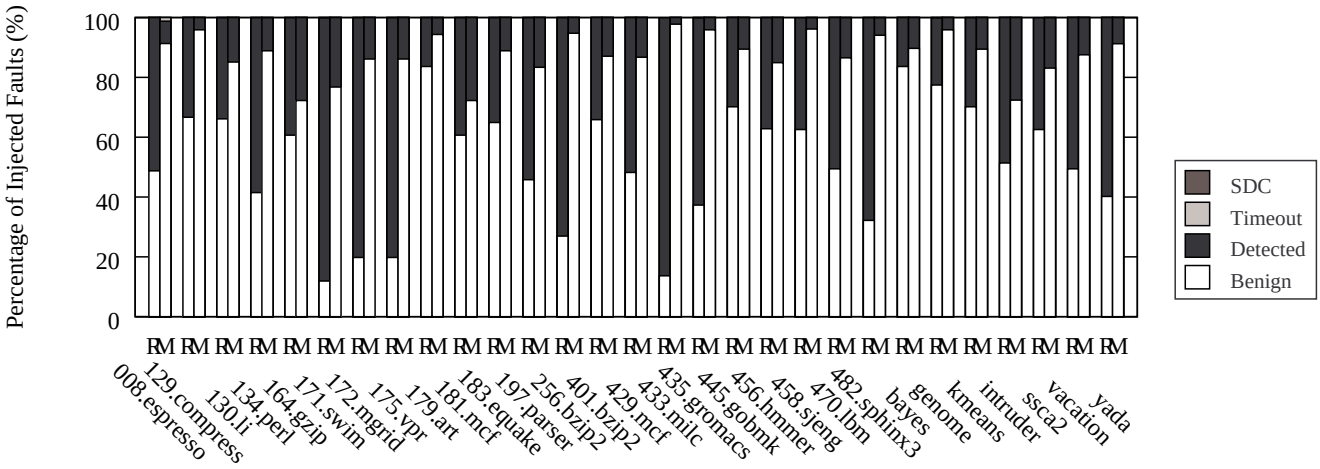

Figure 4: Register (R) and Memory (M) Transient Fault Distribution

the fault-injected program is compared against the reference output to ensure that the RAFT-protected program's externally visible behavior is correct. Figure 4 shows the aggregated fault distribution over 3000 runs.

Figure 4 show that a large percentage of transient faults injected are benign faults, which do not crash the program or modify the output. Among the non-benign faults, all faults injected are detected by RAFT via either value comparison, customized signal handling, or timeout watchdog. As timeout occurs very rarely (0.1% on average), it is hardly visible in Figure 4.

4.2.2 Memory Fault Simulation

Most previous software solutions do not provide memory fault tolerance. The instruction-level redundancy and redundant multi-threading approaches only maintain one memory state, and rely on ECC memory to protect programs against memory transient faults. Previous process-based redundant techniques, such as PLR [31], also lacks protection for memory. Redundant processes share memory in an copy-on-write way. This means the processes share physical pages, if no value is written to that page. Transient faults affecting those pages cannot be detected by PLR. This paper is also simulate and evaluate RAFT against transient faults in memory.

The M columns in Figure 4 demonstrate the results from injecting faults into memory subsystems. Similar to injecting faults into register files, memory fault injection involves a profile run, and a random selection of a program point. Subsequently, memory fault injection randomly selects one virtual memory address owned by one process, and randomly flips a bit of the value stored in that address. This memory address may contain data or text of the program. Memory faults are injected into all three processes, with the likelihood of the occurrence of the faults being proportional to the amount of physical memory used by each process. Each program was executed 3000 times with one memory fault injected each time.

Memory faults injected into RAFT itself may also be detected. For example, if a fault changes the value in memory that stores arguments for later comparison, this fault will be detected during value verification. Also, a faulty value in memory may be loaded to one of the processes and results in a value mismatch in later fault detection. However, if an injected memory fault changed a value in memory that is transitively passed on to both processes, and that value affects the final output, RAFT will not be able to detect it.

As shown in the Figure 4, memory faults usually do not result in erroneous program output. As observed in experiments, a large percentage of memory faults are benign faults. Only an average of 12.83% memory faults are non-benign faults and are detected by RAFT. For 179.art, the original program only consumes 10MB memory at runtime, therefore the probability of memory faults occurring in the monitoring process is as large as 50%. Experimental results shows that among 129 detected faults and 2 timeouts out of 3000 runs, 16 detected faults and 1 timeout are caused by memory faults in the monitoring process.

4.3 Performance

Figure 5 shows the runtime overhead (vertical axis in the figure) of RAFT, normalized with respect to the original program execution without any transient fault detection. The result is compared with RAFT without speculation enabled. We also evaluated the performance overhead of dynamic instrumentation (PIN), which is the base of a previous related work PLR [31], on the same set of benchmarks. This is measured by executing the original program binary within the PIN framework on an unloaded machine, without any instrumentation. The average overhead of the framework alone is as

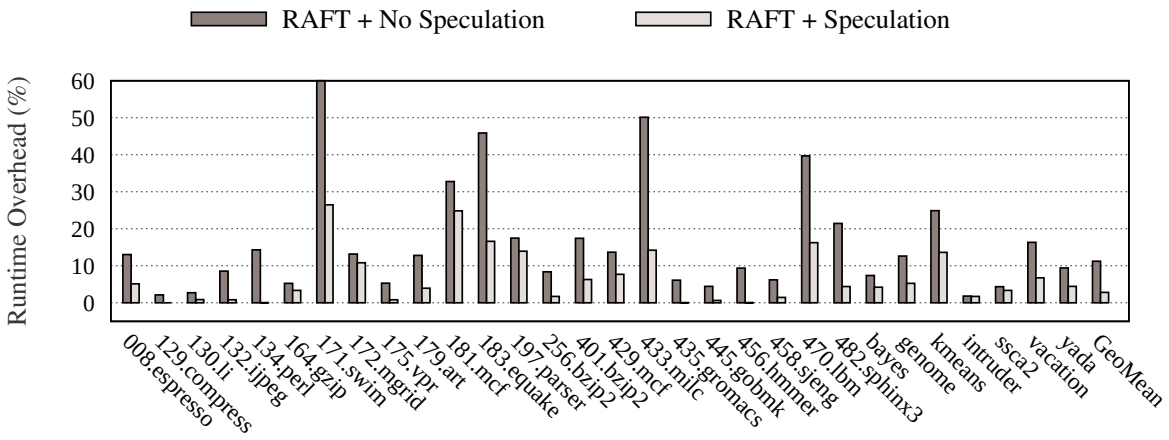

Figure 5: Performance overhead for RAFT, with and without speculation

high as 48.57%. Running redundant instances and runtime instrumentation on top of the framework, in addition to frequent synchronization, can only adds more runtime overhead. Compared with dynamic instrumentation approaches, such as PLR, RAFT provides the same applicability, but better coverage and much lower runtime overhead.

Some benchmarks, such as 183.equake, feature system calls in inner loops. This results in a large amount of inter-process barrier synchronization, if speculation is not available. Frequent synchronizations prevent overlapping of useful computation with fault checking, hence the synchronization overhead is placed on the critical path. As a result, running RAFT on 183.equake without speculation adds 46% overhead compared with the unprotected sequential program, while RAFT (with speculation) added only 16.60% performance overhead. Similarly, 171.swim, 433.milc, genome and kmeans share the same patterns, and gained huge performance improvement from speculation.

In this evaluation, RAFT never misspeculates because the system calls' return values are all correctly speculated. In cases where misspeculation occurs, it costs merely a couple of millisecond for the program to recover from a misspeculation.

As a technique that maintains redundant memory states, RAFT adds memory consumption to the system. A program and its redundant copy together occupy twice as many physical memory pages as the original unprotected program. RAFT itself, however, only consumes stack space and dynamically allocates (and frees) limited amounts of memory space to buffer data system call arguments for transient fault detection. In the experiments conducted in this paper, the tracer process consumes around 9.5MB virtual memory at peak.

5. RELATED WORK

Section 2 summarized the features of different transient fault detection techniques. This section discusses additional related work.

DIVA [39] uses additional hardware checkers to provide fault protection. DieHard [5] proposed by Berger et al. uses redundancy on general-purpose machines for memory fault tolerance. Exterminator [20] uses process replicas to detect memory errors with high probability. RAFT also detects transient faults in memory, as well as register files.

Tapus et al. [34] introduce a syntax and an operational semantics for speculative execution for reliability and fault tolerance. This work proves that the speculative execution model is equivalent to the non-speculative model. Weaver et al. [40] and Vijaykumar et al. [35] use program behavior to direct transient fault detection. These techniques follow the propagation of faults through the program to reduce unnecessary replication. RAFT achieves the same goal by only detecting non-benign faults when they are about to affect program output. Gaiswinkler et al. [12] use the compiler to generate diverse binaries for a program, thus detecting a majority of faults that affect register values. However, this technique has a lower fault coverage than RAFT, and may produce false-positives. Aidemark et al. [1] propose methods to detect software errors by executing an application multiple times and majority voting.

Lee et al. propose Respec, an online multiprocessor replay technique using speculative logging for externally deterministic replay [16]. This technique optimistically logs less information about shared memory dependencies than needed for deterministic replay. If the replayed process diverges from recorded process, misspeculation recovery is performed. This paper and Respec follow the same idea that only externally visible behavior and final state of the program must be correct. Respec was introduced to replay externally visible behavior of shared memory multithreaded programs on commodity multiprocessor architecture. Similarly, RAFT can also be extended to provide transient fault detection for multithreaded programs with deterministic externally visible behavior.

Daniel et al. [10] explore the possibility of using process monitoring utilities of Unix systems for fault tolerance on cluster platforms with NFS. Process monitoring utilities are also exploited for related research topics, such as fault injection and program security checking [32, 15, 9, 14, 38]. Jarboui et al. present a software-implemented fault injection technique to identify different techniques for generating different software faults for operating systems [14]. In this work, they use the UNIX `ptrace` function to trap kernel calls issued by the process where the faults are being

injected. Sieh implements a fault injector using the UNIX `ptrace` process monitoring interface that can inject transient faults into most of the CPU registers, FPU and FPA registers, and into the virtual address space of the running process [32].

FERRARI [15] uses the UNIX `ptrace` function to corrupt the memory image of a process at runtime. The `ptrace` function is used to insert software trap instructions at the specific instruction address where a fault is to be injected. FERRARI can inject faults into the data and code segments of a running process as well as the registers and part of the main memory used by that process. It can also intercept the system calls made by the running process, and change their return values. Buchacker et al. develop a framework for testing the fault-tolerance of systems, where they inject faults into a simulated system of Linux machines using the `ptrace` interface [9]. This paper uses similar methods to inject faults into register files and main memory to study the fault coverage of RAFT.

6. CONCLUSION

Architectural trends toward smaller transistors, higher transistor counts, and lower core voltage make transient faults a more critical reliability concern than ever. Redundant hardware provides transient fault detection at the cost of additional chip area and design cost. Software redundancy is more appealing for its flexibility and low cost. However, even the best available software techniques for transient fault detection have large performance overhead. This paper presents Runtime Asynchronous Fault Tolerance via Speculation (RAFT), the fastest software fault detection technique to date. Combining OS-level process monitoring, value speculation, and efficient misspeculation recovery, RAFT provides transient fault tolerance with only 2.83% runtime overhead. With the flexibility of software and the low runtime overhead, RAFT provides a practical transient fault protection scheme for modern multicore processors.

7. ACKNOWLEDGMENTS

We thank the Liberty Research Group for their support and feedback during this work. We also thank Tim Harris and the anonymous reviewers for their insightful comments and suggestions. This work is supported by DARPA through contract FA8750-10-2-0253, and the National Science Foundation through Grant OCI-1047879. All opinions, findings, conclusions, and recommendations expressed in this paper are those of the Liberty Research Group and do not necessarily reflect the views of our supporters.

8. REFERENCES

[1] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. Experimental evaluation of time-redundant execution for a brake-by-wire application. In *International Conference on the Dependable Systems and Networks (DSN)*, 2002.

[2] H. Ando, Y. Yoshida, A. Inoue, I. Sugiyama, T. Asakawa, K. Morita, T. Muta, T. Motokurumada, S. Okada, H. Yamashita, Y. Satsukawa, A. Konmoto, R. Yamashita, and H. Sugiyama. A 1.3GHz fifth generation SPARC64 Microprocessor. In *Digest of Technical Papers of the 2003 IEEE International Solid-State Circuits Conference*, 2003.

[3] R. C. Baumann. Soft errors in advanced semiconductor devices: the three radiation sources. *IEEE Transactions on Device and Materials Reliability*, 2001.

[4] R. C. Baumann. Soft errors in commercial semiconductor technology: Overview and scaling trends. In *IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals*, April 2002.

[5] E. D. Berger and B. G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006.

[6] S. Borkar. Microarchitecture and design challenges for gigascale integration. In *Proceedings of the 37th International Symposium on Microarchitecture*, 2004.

[7] S. S. Brilliant, J. C. Knight, and N. G. Leveson. Analysis of faults in an n-version software experiment. *IEEE Transactions on Software Engineering*, 1990.

[8] G. Bronevetsky, B. R. de Supinski, and M. Schulz. A foundation for the accurate predication of the soft error vulnerability of scientific applications. In *Proceedings of the 5th Silicon Erros in Logic - System Effects*, 2009.

[9] K. Buchacker and V. Sieh. Framework for testing the fault-tolerance of systems including os and network aspects. In *Proceedings of the 6th IEEE International Symposium on High Assurance Systems Engineering*, 2001.

[10] E. Daniel and G. S. Choi. Tmr for off-the-shelf unix systems. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing*, 1999.

[11] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: probabilistic soft error reliability on the cheap. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010],.

[12] G. Gaiswinkler and A. Gerstinger. Automated software diversity for hardware fault detection. In *ETFA '09: Proceedings of the 14th IEEE Conference on Emerging Technologies Factory Automation, 2009*, 2009.

[13] R. W. Horst, R. L. Harris, and R. L. Jardine. Multiple instruction issue in the NonStop Cyclone processor. In *ISCA*, 1990.

[14] T. Jarboui, J. Arlat, Y. Crouzet, and K. Kanoun. Experimental analysis of the errors induced into linux by three fault injection techniques. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, 2002.

[15] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. Ferrari: A flexible software-based fault and error injection system. *IEEE Transactions on Computers*, 1995.

[16] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: efficient online multiprocessor replayvia speculation and external determinism. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010],.

[17] M. Li, P. Ramach, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. Swat: An error resilient system. In *Proc. of the Fourth Workshop on Silicon Errors in Logic - System Effects*, 2008.

[18] 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*, 2005.

[19] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. *SIGARCH Computer Architecture News*.

[20] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator:

automatically correcting memory errors with high probability. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[21] 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*, 1996.

[22] 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.

[23] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August. Speculative Parallelization Using Software Multi-threaded Transactions. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010],.

[24] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[25] G. A. Reis, J. Chang, D. I. August, R. Cohn, and S. S. Mukherjee. Configurable transient fault detection via dynamic binary translation. In *Proceedings of the 2nd Workshop on Architectural Reliability*, 2006.

[26] 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*, 2005.

[27] 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 *Proc. of the International Symposium on Computer Architecture (ISCA)*, 2005.

[28] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*, 1999.

[29] S. K. Sastry Hari, M.-L. Li, P. Ramachandran, B. Choi, and S. V. Adve. mswat: low-cost hardware fault detection and diagnosis for mukticore systems. In *Proceedings of the 42nd International Symposium on Microarchitecture*, 2009.

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

[31] A. Shye, T. Moseley, V. J. Reddi, J. B. t, and D. A. Connors. Using process-level redundancy to exploit multiple cores for transient fault tolerance. In *International Conference on the Dependable Systems and Networks (DSN)*, 2007.

[32] V. Sieh. Fault-injector using unix ptrace interface. In *Internal Report 11/93, IMMD3, UniversitÃd't ErlangenNÃijrnberg*, 1993.

[33] T. J. Slegel, R. M. Averill III, M. A. Check, B. C. Giamei, B. W. Krumm, C. A. Krygowski, W. H. Li, J. S. Liptay, J. D. MacDougall, T. J. McPherson, J. A. Navarro, E. M. Schwarz, K. Shum, and C. F. Webb. IBM's S/390 G5 Microprocessor design. In *IEEE Micro*, volume 19, pages 12–23, March 1999.

[34] C. Tapus and J. Hickey. Distributed speculative execution for reliability and fault tolerance: an operational semantics. *The Journal of Distributed Computing*, 21(6):433–455.

[35] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery using simultaneous multithreading. In *Proceedings of the 29th International Symposium on Computer Architecture (ISCA)*, 2002.

[36] D. Walker, L. Mackey, J. Ligatti, G. A. Reis, and D. I. August. Static typing for a faulty lambda calculus. *SIGPLAN Notices*, 2006.

[37] C. Wang, H.-S. Kim, Y. Wu, and V. Ying. Compiler-managed software-based redundant multi-threading for transient fault detection. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 244–258, Washington, DC, USA, 2007.

[38] R. N. M. Watson. Exploiting concurrency vulnerabilities in system call wrappers. In *Proceedings of the first USENIX workshop on Offensive Technologies*, 2007.

[39] C. Weaver and T. M. Austin. A fault tolerant approach to microprocessor design. In *International Conference on the Dependable Systems and Networks (DSN)*, 2001.

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

[41] Y. Yeh. Triple-triple redundant 777 primary flight computer. In *Proceedings of the 1996 IEEE Aerospace Applications Conference*, pages 293–307, February 1996.

[42] Y. Zhang, J. W. Lee, N. P. Johnson, and D. I. August. Daft: Decoupled Acyclic Fault Tolerance. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.