Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance

Alex Shye Tipp Moseley[†] Vijay Janapa Reddi[‡] Joseph Blomstedt Daniel A. Connors

Dept. of Electrical and Computer Engineering
U. of Colorado at Boulder
{shye, blomsted, dconnors}@colorado.edu

[†]Dept. of Computer Science
U. of Colorado at Boulder
moseleyt@colorado.edu

[‡]Dept. of Elect. Eng. and Computer Science
Harvard University
vj@eecs.harvard.edu

Abstract

Transient faults are emerging as a critical concern in the reliability of general-purpose microprocessors. As architectural trends point towards multi-threaded multi-core designs, there is substantial interest in adapting such parallel hardware resources for transient fault tolerance. This paper proposes a software-based multi-core alternative for transient fault tolerance using process-level redundancy (PLR). PLR creates a set of redundant processes per application process and systematically compares the processes to guarantee correct execution. Redundancy at the process level allows the operating system to freely schedule the processes across all available hardware resources. PLR's software-centric approach to transient fault tolerance shifts the focus from ensuring correct hardware execution to ensuring correct software execution. As a result, PLR ignores many benign faults that do not propagate to affect program correctness. A real PLR prototype for running single-threaded applications is presented and evaluated for fault coverage and performance. On a 4-way SMP machine, PLR provides improved performance over existing software transient fault tolerance techniques with 16.9% overhead for fault detection on a set of optimized SPEC2000 binaries.

1 Introduction

Transient faults, also known as soft errors, are emerging as a critical concern in the reliability of computer systems [4, 21]. A transient fault occurs when an event (e.g. cosmic particle strikes, power supply noise, device coupling) causes the deposit or removal of enough charge to invert the state of a transistor. The inverted value may propagate to cause an error in program execution.

Current trends in process technology indicate that the future error rate of a single transistor will remain relatively constant [13, 18]. As the number of available transistors per chip continues to grow exponentially, the error rate of for an entire chip is expected to increase dramatically. These trends indicate that to ensure correct operation of systems, all general-purpose microprocessors and memories must employ reliability techniques.

Transient faults have historically been a design concern in specific computing environments (e.g. spacecrafts, high-availability server machines) in which the key system characteristics are reliability, dependability, and availability. While memory is easily protected with error-correcting code (ECC) and parity, protecting the complex logic within a high-performance microprocessor presents a significant challenge. Custom hardware designs have added 20-30% additional logic to add redundancy to mainframe processors and cover upwards of 200,000 latches [32, 2]. Other approaches include specialized machines with custom hardware and software redundancy [16, 39].

However, the same customized techniques can not be directly adopted for the general-purpose computing domain. Compared to the ultra-reliable computing environments, general-purpose systems are driven by a different, and often conflicting, set of factors. These factors include:

Application Specific Constraints: In ultra-reliable environments, such as spacecraft systems, the result of an transient error can be the difference between life or death. For general-purpose computing, the consequences of faulty execution are often less severe. For instance, in audio decode and playback, a fault results in a mere glitch which may not even be noticed. Thus, the focus for reliability shifts from providing a bullet-proof system to improving reliability to meet user expectations of failure rates.

Design Time and Cost Constraints: In the general-purpose computing market, low cost and a quick time to market are paramount. The design and verification of new redundant hardware is costly and may not be feasible in cost-sensitive markets. In addition, the inclusion of redundant design elements may negatively impact the design and product cycles of systems.

Post-Design Environment Techniques: A system's susceptibility to transient faults is often unplanned for and appears after the design and fabrication processes. For example, the scientists at the Los Alamos National Laboratory documented a surprisingly high incidence of single-node failures due to transient faults during the deployment of the

ASC Q supercomputer [21]. Likewise, environmental conditions of a system such as altitude, temperature, and age can cause higher fault rates [40]. In these cases, reliability techniques must be augmented after the design and development phase without the addition of new hardware.

With such pressures driving general-purpose computing hardware, software reliability techniques are an attractive solution for improving reliability in the face of transient faults. While software techniques cannot provide a level of reliability comparable to hardware techniques, they significantly lower costs (zero hardware design cost), and are very flexible in deployment. Existing software transient fault tolerant approaches use the compiler to insert redundant instructions for checking computation [26], control flow [25], or both [29]. The compiler-based software techniques suffer from a few limitations. First, the execution of the inserted instructions and assertions decreases performance ($\sim$1.4x slowdown [29]). Second, a compiler approach requires recompilation of all applications. Not only is it inconvenient to recompile all applications and libraries, but the source code for legacy programs is often unavailable.

This paper presents *process-level redundancy* (PLR), a software reliability technique leverages multiple processor cores for transient fault tolerance. PLR creates a set of redundant processes per original application process and compares their output to ensure correct execution. PLR scales with the architectural trend towards large many-core machines and leverages available hardware parallelism to improve performance without any additional redundant hardware structures or modifications to the system. In computing environments which are not throughput-constrained, PLR provides an alternate method of leveraging the hardware resources for transient fault tolerance. In addition, PLR can be easily deployed without recompilation or modifications to the underlying operating system.

This paper makes the following contributions:

- PLR implies a *software-centric* paradigm in transient fault tolerance which views the system as software layers which must execute correctly. In contrast, the typical *hardware-centric* paradigm views the system as a collection of hardware that must be protected. We differentiate between software-centric and hardware-centric views using the commonly accepted *sphere of influence* concept.

- Demonstrates the benefits of a software-centric approach. In particular, we show how register errors propagate through software. We show that many of the errors result in benign faults and many detected faults propagate through hundreds or thousands of instructions. By using a software-centric approach, PLR is able to ignore many benign faults.

- Presents a software-only transient fault tolerance technique for leveraging multiple cores on a general-purpose microprocessor for transient fault tolerance. We describe a real prototype system designed for single-threaded applications and evaluate the fault coverage and performance of PLR. Overall, the PLR prototype runs a set of the *SPEC2000* benchmark suite with only a 16.9% overhead on a 4-way SMP system.

The rest of this paper is organized as follows. Section 2 provides background on transient fault tolerance. Section 3 describes PLR. Section 4 shows initial results from the dynamic PLR prototype. Section 5 discusses related work. Section 6 concludes the paper.

2 Background

In general, a fault can be classified by its effect on system execution into the following categories [37]:

Benign Fault: A transient fault which does not propagate to affect the correctness of an application is considered a benign fault. A benign fault can occur for a number of reasons. Examples include a fault to an idle functional unit, a fault to a performance-enhancing instruction (i.e. a prefetch instruction), data masking, and Y-branches [36].

Silent Data Corruption (SDC): An undetected fault which propagates to corrupt system output is an SDC. This is the worst case scenario where a system appears to execute correctly but silently produces incorrect output.

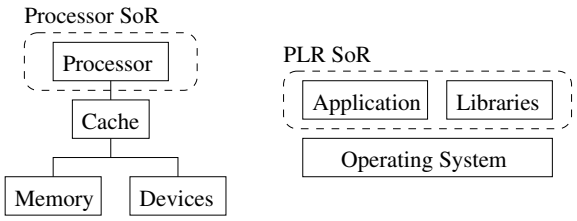
Detected Unrecoverable Error (DUE): A fault which is detected without possibility of recovery is considered a DUE. DUEs can be split into two categories. A *true DUE* occurs when a fault which would propagate to incorrect execution is detected. A *false DUE* occurs when a benign fault is detected as a fault.

A transient fault in a system without transient fault tolerance will result in a benign fault, SDC, or true DUE (e.g. error detected by core dump). A system with only detection attempts to detect all of the true DUEs and SDCs. However, the system may inadvertently convert some of the benign faults into false DUEs and unnecessarily halt execution. Finally, a system with both detection and recovery will detect and recover from all faults without SDCs or any form of DUE. In this case, faults which would be false DUEs may cause unwarranted invocations to the recovery mechanism.

3 Approach

3.1 Software-centric Fault Detection

The *sphere of replication* (SoR) [28] is a commonly accepted concept for describing a technique's logical domain of redundancy and specifying the boundary for fault detection and containment. Any data which enters the SoR is replicated and all execution within the SoR is redundant in



(a) Hardware−centric (b) Software−centric

Figure 1. Hardware-centric and software-centric transient fault detection models.

some form. Before leaving the SoR, all output data is compared to ensure correctness. All execution outside of the SoR is not covered by the particular transient fault techniques and must be protected by other means. Faults are contained within the SoR boundaries and detected in any data leaving the SoR.

Most previous work in fault tolerance is *hardware-centric* and uses a hardware-centric SoR. A hardware-centric model views the system as a collection of hardware components which must be protected from transient faults. In this model, a hardware-centric SoR is placed around specific hardware units. All inputs are replicated, execution is redundant, and output is compared.

While the hardware-centric model is appropriate for hardware-implemented techniques, it is awkward to apply the same approach to software. Software naturally operates at a different level and does not have full visibility into the hardware. Nevertheless, previous compiler-based approaches attempt to imitate a hardware-centric SoR. For example, SWIFT [29] places its SoR around the processor as shown in Figure 1(a). Without the ability to control duplication of hardware, SWIFT duplicates at the instruction level. Each load is performed twice for input replication and all computation is performed twice on the replicated inputs. Output comparison is accomplished by checking the data of each store instruction prior to executing the store instruction. This particular approach works because it is possible to emulate processor redundancy with redundant instructions. However, other hardware-centric SoRs would be impossible to emulate with software. For example, software alone cannot implement an SoR around hardware caches.

Software-centric fault detection is a paradigm in which the system is viewed as the software layers which must execute correctly. A software-centric model uses a software-centric SoR which is placed around software layers, instead of hardware components. The key insight to software-centric fault detection is this: although faults occur at the hardware level, *the only faults which matter are the faults which affect software correctness.* By changing the boundaries of output comparison to software, a software-centric model shifts the focus from ensuring correct hardware execution to ensuring correct software execution. As a result, only faults which affect correctness are detected. Benign faults are safely ignored. A software-centric system with only detection is able to reduce the incidence of false DUEs. A software-centric system with both detection and recovery will not need to invoke the recovery mechanism for faults which do not affect correctness.

Figure 1(b) shows an example software-centric SoR which is placed around the user space application and libraries (as used by PLR). A software-centric SoR acts exactly the same as the hardware-centric SoR except that it acts on the software instead of the hardware. Again, all input is replicated, execution within the SoR is redundant, and data leaving the SoR is compared.

By operating at the software level, the software-centric model caters to the strengths of a software-implemented technique. While software has limited visibility into hardware, it is able to view a fault at a broader scope and determine its effect on software execution. Thus, software-implemented approaches which are hardware-centric are ignoring the potential strengths of a software approach.

3.2 Process-Level Redundancy

Process-level redundancy (PLR) is a technique which uses the software-centric model of transient fault detection. As shown in Figure 1(b), PLR places its SoR around the user address space by providing redundancy at the process level. PLR replicates the application and library code, global data, heap, stack, file descriptor table, etc. Everything outside of the SoR, namely the OS, must be protected by other means. Any data which enters the SoR via the system call interface must be replicated and all output data must be compared to verify correctness.

Providing redundancy at the process level is natural as it is the most basic abstraction of any OS. The OS views any hardware thread or core as a logical processor and then schedules processes to the available logical processors. PLR leverages the OS to schedule the redundant processes to take advantage of hardware resources. With massive multi-core architectures on the horizon, there will be a tremendous amount of hardware parallelism available in future general-purpose machines. In computing environments where throughput is not the primary concern, PLR provides a way of utilizing the extra hardware resources for transient fault tolerance.

A high level overview of PLR is shown in Figure 2 with three redundant processes, which is the minimum number of processes necessary for both transient fault detection and recovery. PLR intercepts the beginning of application execution and replicates the original process to create other redundant processes. One of the processes is logically labeled the *master* process and the others are labeled the *slave* processes. At each system call, the *system call emulation unit* is invoked. The emulation unit performs the input replica-

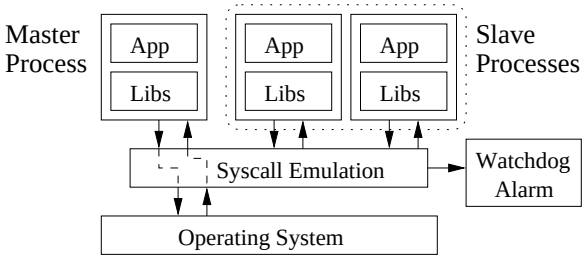


Figure 2. Overview of PLR with three redundant processes.

tion, output comparison, and recovery. The emulation unit also ensures that the following requirements are maintained in order for PLR to operate correctly:

- The execution of PLR must be transparent to the system environment with the redundant processes interacting with the system as if only the original process is executing. System calls which alter any system state can only be executed once with the master process actually executing the system call and the slave processes emulating the system call.
- Execution among the redundant processes must be deterministic. System calls which return non-deterministic data, such as a request for system time or resources, must be emulated to ensure all redundant processes use the same data for computation.
- All redundant processes must be identical in address space and any other process-specific data, such as the file descriptor table. At any time, a transient fault could render one of the redundant processes useless. With identical processes, any of the processes can be logically labeled the master process at any given invocation of the emulation unit.

On occasion, a transient fault will cause the program to suspend or hang. The *watchdog alarm* is employed by the emulation unit to detect such faults. Upon entrance to the system call emulation unit, a timer begins. If the redundant processes do not all enter the emulation unit in a user-specified amount of time, the watchdog alarm times out, signaling an error in execution.

3.2.1 Input Replication

As the SoR model dictates, any data which enters the SoR must be replicated to ensure that all data is redundant within the SoR. In the case of PLR, any data which passes into the processes via system calls (such as a read from a file descriptor) is received once by the master process, and then replicated among the slave processes. Also, the return value from all system calls is considered an input value and is copied for use across all redundant processes.

3.2.2 Output Comparison

All data which exits the redundant processes must be compared for correctness before proceeding out of the SoR. If the output data does not match, a transient fault is detected and a recovery routine is invoked. Any write buffers which will be passed outside of the SoR must be compared. Also, any data passed as a system call parameter can be considered an output event which leaves the SoR and must also be checked to verify program correctness.

3.2.3 Emulating System Calls

The emulation unit is responsible for the input replication, output comparison, and system call emulation. The data transfer during input replication and output comparison is accomplished through a shared memory segment between all of the redundant processes.

At the beginning of each call to the emulation unit, the type of system call is compared to ensure that all redundant processes are at a common system call. If not, a fault is assumed which caused an error in control flow to call an errant system call.

Depending upon the system call, the system call emulation unit will perform different tasks. System calls which modify any system state, such file renaming and linking, must only be executed once. In other cases, the system call will be actually called by all processes; once by the master process in its original state, and once by each redundant process to emulate the operation. For example, in emulating a system call to open a new file, the master process will create and open the new file, while the redundant processes will simply open the file without creating it.

3.3 Transient Fault Detection

A transient fault is detected in one of three ways:

1. **Output Mismatch**: A transient fault which propagates to cause incorrect output will be detected with the output comparison within the emulation unit at the point which the data is about to exit the SoR.
2. **Watchdog Timeout:** There are two scenarios in which the watchdog timer will time out. The first case is when a fault causes an error in control flow which calls an errant system call. The faulty process will cause an entrance into the emulation unit which will begin waiting for the other processes. If the other processes enter the emulation unit, an error will be detected if the system calls mismatch, or if there is a mismatch in data. If the other processes continue execution, a timeout will occur. The second case is when a transient fault causes a process to hang indefinitely (e.g. an infinite loop). In this case, during the next system call, all the processes except the hanging process will enter the emulation unit and eventually cause a watchdog timeout. A

drawback to the watchdog alarm is that a timeout period exists in which the application does not make any progress. In our experience, on an unloaded system, a timeout of 1-2 seconds is sufficient. The timeout value is user specified and can be increased on a loaded system. On a loaded system, spurious timeouts will not affect application correctness, but will cause unnecessary calls to the recovery unit.

3. **Program Failure:** Finally, a transient fault may cause a program failure due to an illegal operation such as a segmentation violation, bus error, illegal instruction, etc. Signals handlers are set up to catch the corresponding signals and an error is be flagged. The next time the emulation unit is called, it can immediately begin the recovery process.

3.4 Transient Fault Recovery

Transient fault recovery mechanisms typically fit into two broad categories: *checkpoint and repair*, and *fault masking*. Checkpoint and repair techniques involves the periodic checkpointing of execution state. When a fault is detected, execution is rolled back to the previous checkpoint. Fault masking involves using multiple copies of execution to vote on the correct output.

PLR supports both types of fault recovery. If checkpoint and repair functionality already exists, then PLR only needs to use two processes for detection and can defer recovery to the repair mechanism. Otherwise, fault masking can be accomplished by using at least three processes for a majority vote. If fault masking is used, the following schemes are used for recovery (the examples use an assumption of three redundant processes).

1. **Output Mismatch:** If an output data mismatch occurs the remaining processes are compared to ensure correctness of the output data. If a majority of processes agree upon the value of the output data, it is assumed to be correct. The processes with incorrect data are immediately killed and replaced by duplicating a correct process (e.g. using the `fork()` system call in Linux).
2. **Watchdog Timeout:** As mentioned in Section 3.3, there are two cases for a watchdog timeout. In the first case, where a faulty process calling the emulation unit while the other processes continue executing, there will only be one process in the emulation unit during timeout. The process in the emulation unit is killed and recovery occurs during the next system call. In the second case, where a faulty process hangs, all processes except one will be in the emulation unit during timeout. The hanging process is killed and replaced by duplicating a correct process.
3. **Program Failure:** In the case of program failure, the incorrect process is already dead. The emulation unit simply replaces the missing process by duplicating one of the remaining processes.

We assume the single event upset (SEU) fault model in which a single transient fault occurs at a time. However, PLR can support simultaneous faults by simply scaling the number of redundant processes and the majority vote logic.

3.5 Windows of Vulnerability

A fault during execution of PLR code may cause an unrecoverable error. Also, a fault which causes an erroneous branch into PLR code could result in undefined behavior. Finally, PLR is not meant to protect the operating system and any fault during operating system execution may cause failure. The first and third windows of vulnerability may be mitigated by compiling the operating system and/or PLR code with compiler-based fault tolerance solutions.

All fault tolerance techniques have windows of vulnerability which are usually associated with faults to the checker mechanism. Although not completely reliable, partial redundancy [12, 33] may be sufficient to improve reliability enough to meet user or vendor reliability standards.

3.6 Shared Memory, Interrupts, Exceptions and Multi-threading

PLR hinges upon deterministic behavior among the redundant processes. However, shared memory, interrupts, exceptions and multi-threaded applications introduce potential non-determinism.

Shared memory could be supported by changing page permissions and trapping upon accesses to the shared memory. A similar approach is used for detecting self-modifying code within dynamic code translators [9]. Interrupts and exceptions present a more difficult challenge because there is not a clear execution point in which to synchronize the redundant processes. Hardware supported techniques have been proposed previously such as hardware counters to support epochs [8]. Multi-threaded applications require a programming model that ensures the same inter-thread memory ordering for each replica. Without this support, PLR is limited to executing on single-threaded applications.

These challenges are still open research problems for all software-implemented fault tolerance techniques. We plan to explore extensions to PLR to support these non-deterministic issues.

4 Experimental Results

This paper presents and evaluates a PLR prototype built using the Intel Pin dynamic binary instrumentation system [20]. The tool uses Pin to dynamically create redundant processes and uses PinProbes (a dynamic code patching system for program binaries) to intercept system calls.

The prototype is evaluated running a set of the *SPEC2000* benchmarks compiled with gcc v3.4.6 and ifort

v9.0. Fault coverage is evaluated using a fault injection campaign similar to [29]. One thousand runs are executed per benchmark. To maintain manageable run times, the test inputs are used during fault analysis. For each run, an instruction execution count profile of the application is used to randomly choose a specific invocation of an instruction to fault. For the selected instruction, a random bit is selected from the source or destination general-purpose registers. To inject a simulated transient error, Pin tool instrumentation is used to change the random bit during the specified dynamic execution count of the instruction. The *specdiff* utility included within the *SPEC2000* harness is used to determine the correctness of program output.

Fault propagation and performance evaluation are both studied using the reference inputs. Performance is measured by running the PLR prototype with both two and three redundant processes without fault injection on a four-processor SMP system; specifically the system has four 3.00Ghz Intel Xeon MP processors each with 4096KB L3 cache, has 6GB of system-wide memory, and is running Red Hat Enterprise Linux AS release 4.

4.1 Fault Injection Results

A fault injection study is performed to illustrate the effectiveness of PLR as well as the benefits of using a software-centric model of fault detection. Figure 3 shows the results of a fault injection campaign with the left bar in each cluster showing the outcomes with just fault injection and the right bar showing the outcomes when detecting faults with PLR. The possible outcomes are:

- **Correct:** A benign fault which does not affect program correctness.
- **Incorrect:** An SDC where the program executes completely and returns with correct return code, but the output is incorrect.
- **Abort:** A DUE in which the program returns with an invalid return code.
- **Failed:** A DUE in which the program terminates (e.g. segmentation violation).
- **Mismatch:** Occurs when running PLR. In this case, a mismatch is detected during PLR output comparison.
- **SigHandler:** Occurs when running PLR. In this case, a PLR signal handler detects program termination.

Timeouts of the watchdog alarm are ignored because they occurs very infrequently ($\sim$.05% of the time).

PLR is able to successfully eliminate all of the *Failed*, *Abort*, and *Incorrect* outcomes. In general, the output comparison detects the *Incorrect* and *Abort* cases, and turns each error into detected *Mismatch* cases. Similarly, PLR detects the *Failed* cases turning them into *SigHandler* cases. Occasionally, a small fraction of the *Failed* cases are detected as *Mismatch* under PLR. This indicates cases in which PLR is able to detect a mismatch of output data before a failure occurs.

The software-centric approach of PLR is very effective at detecting faults based on their effect on software execution. Faults which do not affect correctness are generally not detected in PLR, thereby avoiding false positives. In contrast, SWIFT [29], which is currently the most advanced compiler-based approach, detects roughly $\sim$70% of the *Correct* outcomes as faults.

However, not all of the *Correct* cases during fault injection remain *Correct* with PLR detection as the software-centric model would suggest. This mainly occurs with the *SPECfp* benchmarks. In particular, *168.wupwise*, *172.mgrid* and *178.galgel* show that many of the original *Correct* cases during fault injection become detected as *Mismatch*. In these cases, the injected fault causes the output data to be different than data from regular runs. However, the output difference occurs in the printing of floating point numbers to a log file. *specdiff* allows for a certain tolerance in floating point calculations, and considers the difference within acceptable bounds. PLR compares the raw bytes of output and detects a fault because the data does not match. This issue has less to do with the effectiveness of a PLR, or a software-centric model, and is more related to the definition of an application's correctness.

4.2 Fault Propagation

Figure 4 shows the number of instructions executed between fault injection and detection. Runs are shown as stacked bars showing the breakdown of instructions executed before the fault was detected. The leftmost bar labeled M shows the breakdowns for the *Mismatch* runs shown in Figure 3. The middle bar (S) shows the breakdown for the *SigHandler* runs and the left bar (A) shows all of the detected faults including both *Mismatch* and *SegHandler*.

In general, the *Mismatch* runs tend to be detected much later than the point of fault injection with fault propagation instruction counts of over 10,000 instructions for nearly all of the benchmarks. On the other hand, the *SegHandler* runs have a higher probability of being detected early. Across all of the detected runs, there is a wide variety in amounts of fault propagation ranging from *254.gap* which has a low amount of fault propagation, to *191.fma3d* which has an even distribution of runs among the various categories.

The software-centric model delays the detection of a fault until an error is certain via program failure, or incorrect data exiting the SoR. However, the delayed detection also means that a fault may remain latent during execution for an unbounded period of time. Future work remains in characterizing fault propagation as well as exploring methods for bounding the time in which faults remain undetected.

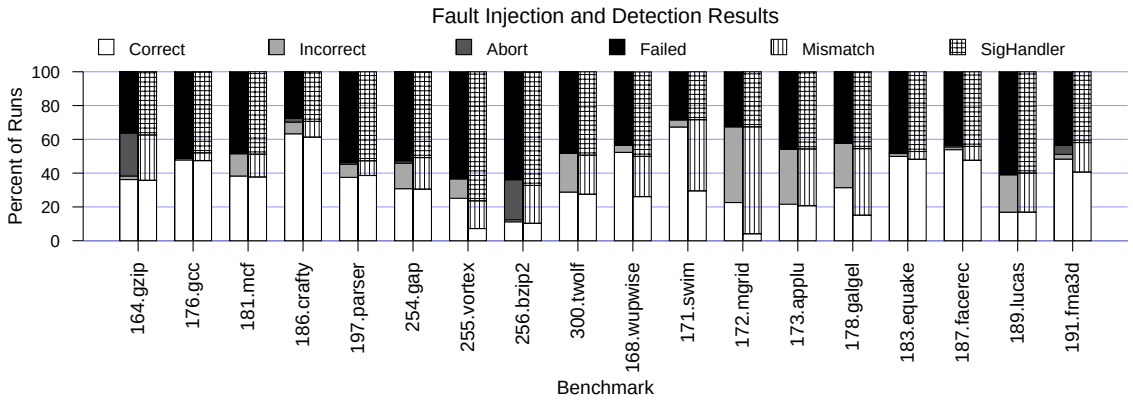


Figure 3. Results of the fault injection campaign. The left bar in each cluster shows the outcomes with just fault injection and the right bar shows the breakdown of how PLR detects the faults.

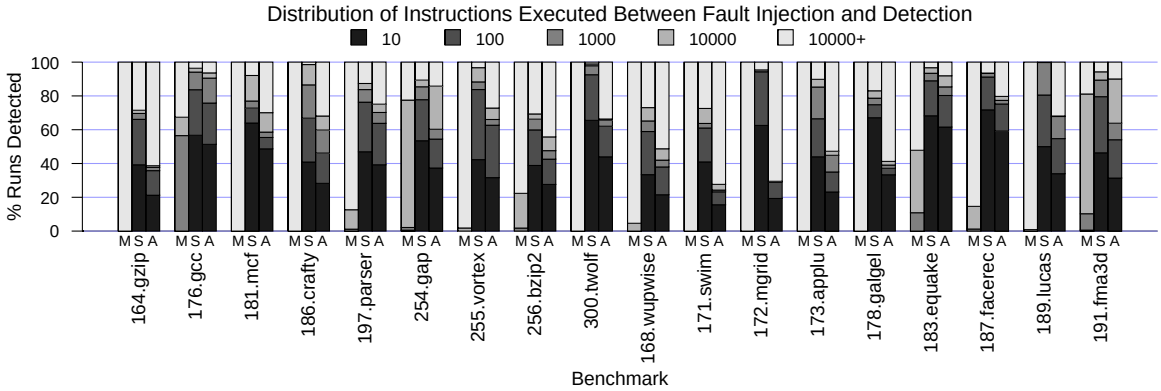


Figure 4. Distribution of the number of executed instructions between the injection and detection of a fault. Percentages are normalized to all the runs which are detected via output mismatch (M), program failure (S), or both combined (A).

4.3 Performance Results

Performance is evaluated using two redundant processes for fault detection (PLR2), and three processes to support recovery (PLR3). Figure 5 shows PLR performance on benchmarks compiled with both `-O0` and `-O2` compiler flags. Performance is normalized to native execution time. PLR provides transient fault tolerance on `-O0` programs with an average overhead of 8.1% overhead for PLR2 and 15.2% overhead for PLR3. On `-O2` programs, PLR2 incurs a 16.9% overhead for PLR2 and 41.1% overhead for PLR3. Overhead in PLR is due to the fact that multiple redundant processes are contending for system resources. Programs which place higher demands on systems resources result in a higher PLR overhead. Optimized binaries stress the system more than unoptimized binaries (e.g. higher L3 cache miss rate) and therefore have a higher overhead. As the number of redundant processes increases, there is an increasing burden placed upon the system memory controller, bus, as well as cache coherency implementation. Similarly, as the emulation is called with more processes, the increased synchronization with semaphores and the usage and shared memory may decrease performance. At certain points, the system resources will be saturated and performance will be severely impacted. These cases can be observed in *181.mcf* and *171.swim* when running PLR3 with `-O2` binaries. PLR overhead and system resource saturation points are explained in more detail in the next subsection.

4.4 PLR Overhead Breakdown

The performance overhead of PLR consists of *contention overhead* and *emulation overhead*, shown as stacked bars in Figure 5. Contention overhead is the overhead from simultaneously running the redundant processes and contending for shared resources such as the memory and system bus. The contention overhead is measured by running the application multiple times independently and comparing the overhead to the execution of a single run. This roughly simulates running the redundant processes without PLR's synchronization and emulation. The rest of the overhead is considered emulation overhead. Emulation overhead is due to the synchronization, system call emulation, and mechanisms for fault detection incurred by PLR.

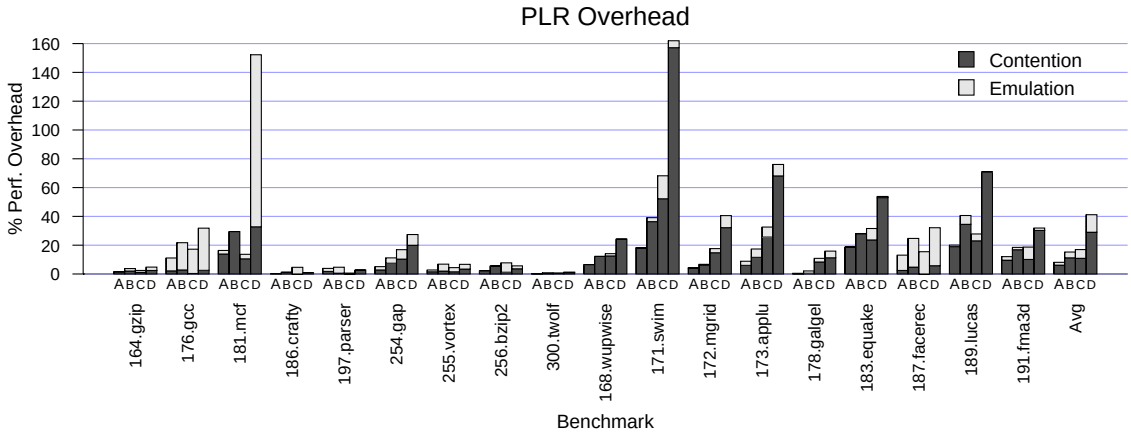


Figure 5. Overhead of running PLR on a set of both unoptimized and optimized *SPEC2000* benchmarks. The combinations of runs include `-O0` compiled binaries with PLR2 (A), `-O0` with PLR3 (B), `-O2` with PLR2 (C) and `-O2` with PLR3 (D).

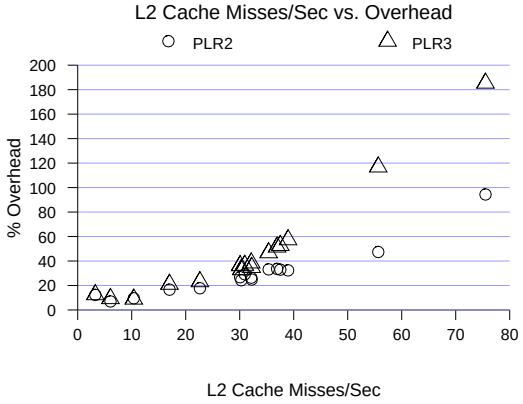


Figure 6. PLR overhead vs. L3 cache miss rate.

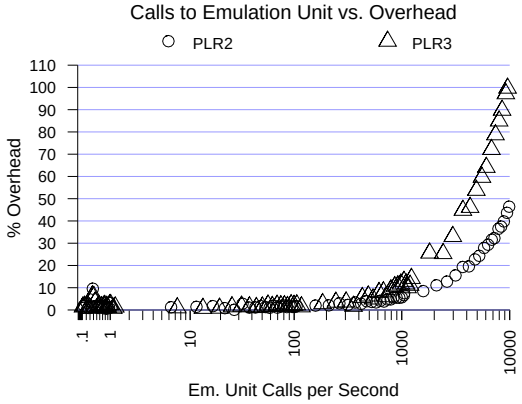


Figure 7. PLR overhead vs. system call rate.

For the set of benchmarks, contention overhead is significantly higher than emulation overhead. Benchmarks such as *181.mcf* and *189.lucas* have relatively high cache miss rates leading to a high contention overhead with increased memory and bus utilization. On the other hand, *176.gcc* and *187.facerec* substantially utilize the emulation unit and result in a high PLR overhead.

4.4.1 Contention Overhead

Contention overhead mainly stems from the sharing of memory bandwidth between the multiple redundant processes. To study the effects of contention overhead, we construct a program to generate memory requests by periodically missing in the L3 cache. Figure 6 shows the effect of L3 cache miss rate on contention overhead when running with PLR. For both PLR2 and PLR3, the L3 cache miss rate has a substantial affect on the contention overhead. With less than 10 L3 cache misses per second, there can be a significant overhead of about 10%. After that point, the overhead increases greatly with over a 50% overhead at about 40 L3 cache misses per second. These results indicate that the total overhead for using PLR is highly impacted by the applications cache memory behavior. CPU-bound applications can be protected from transient faults with a very low overhead while memory-bound applications may suffer from high overheads.

4.4.2 Emulation Overhead

Emulation overhead mainly consists of the synchronization overhead and the overhead from transferring and comparing data in shared memory. To examine each aspect of emulation overhead, two synthetic programs were designed and run with PLR. The first program calls the `times()` system call at a user-controlled rate. `times()` is one of the of simpler system calls supported by PLR and is used to measure the emulation overhead from the barrier synchronizations within the emulation unit. The second test program calls the `write()` system call ten times a second and writes a user-

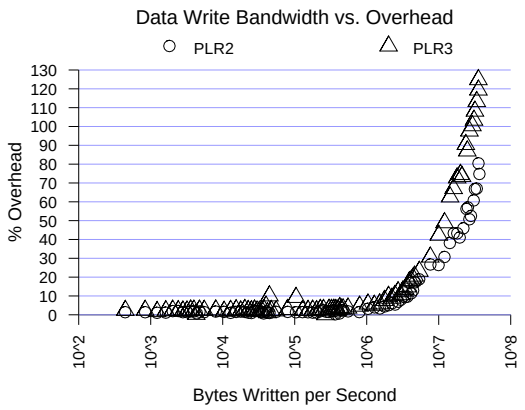


Figure 8. PLR overhead vs. data bandwidth.

specified number of bytes per system call. Each `write()` system call forces the emulation unit to transfer and compare the write data in shared memory.

Figure 7 shows the effect of synchronization on the PLR overhead. Synchronization overhead is minimal up until about 300-400 emulation unit calls per second with less than 5% overhead for using PLR with both two and three redundant processes. Afterward, the emulation overhead increases quickly. Overall, these results indicate that the PLR technique might be best deployed for specific application domains without significant system call functionality.

Figure 8 illustrates the effect of write data bandwidth on emulation overhead. The experiment evaluates the amount of data at each system call that must be compared between redundant process techniques. The write data bandwidth has a similar characteristics as system call synchronization, achieving low overhead until a cut-off point. In this case, for the experimental machines evaluated, the overhead is minimal when the write data rate stays less than 1MB per second but then increases substantially after that point for both PLR2 and PLR3.

5 Related Work

PLR is similar to a software version of the hardware SMT and CMP extensions for transient fault tolerance [11, 23, 28]. However, PLR aims to provide the same functionality in software. Wang [35] proposes a compiler infrastructure for software redundant multi-threading which achieves 19% overhead with the addition of a special hardware communication queue. PLR attains similar overhead and only relies on the fact that multiple processors exist. In addition, PLR does not require source code to operate.

Executable assertions [14, 15] and other software detectors [27] explore the placement of assertions within software. Other schemes explicitly check control flow during execution [31, 25]. The software-centric approach provides a different model for transient fault tolerance using a software equivalent of the commonly accepted SoR model. The pi bit [37] and dependence-based checking [34] have been explored as methods to follow the propagation of faults in an attempt to only detect faults which affect program behavior. The software-centric model accomplishes the same task on a larger scale.

The PLR approach is similar to a body of fault tolerant work which explores the use of replicas for fault tolerance [6, 8, 7, 24, 38, 39]. This body of work targets hard faults (such as hardware or power failures) and assumes fail-stop execution [30] in which the processor stops in the event of failure. For transient faults, this assumption does not hold. As far as we know, we provide the first performance evaluation, and overhead breakdown, of using redundant processes on general-purpose multiple core systems.

There have been a number of previous approaches to program replication. N-version programming [3] uses three different versions of an application for tolerating software errors. Aidemark uses a time redundant technique which execute an application multiple times and use majority voting [1]. Virtual duplex systems combine both N-version programming and time-redundancy [10, 19]. The Tandem Nonstop Cyclone [16] is a custom system designed to use process replicas for transaction processing workloads. Chameleon [17] is an infrastructure designed for distributed systems which uses various ARMOR processes (some similar to process replicas) to implement adaptive and configurable fault tolerance. DieHard [5] proposes using replicas in general-purpose machines for tolerating memory errors. Shadow profiling [22] uses process replicas for low-overhead program instrumentation.

6 Conclusion

This paper motivates the necessity for software transient fault tolerance for general-purpose microprocessors and proposes process-level redundancy (PLR) as an attractive alternative in emerging multi-core processors. By providing redundancy at the process level, PLR leverages the OS to freely schedule the processes to all available hardware resources. In addition, PLR can be deployed without modifications to the application, operating system or underlying hardware. A real PLR prototype supporting single-threaded applications is presented and evaluated for fault coverage and performance. Fault injection experiments prove that PLR's software-centric fault detection model effectively detects faults which safely ignoring benign faults. Experimental results show that when running an optimized set of *SPEC2000* benchmarks on a 4-way SMP machine, PLR provides fault detection with an 16.9% overhead. PLR performance improves upon existing software transient fault techniques and takes a step towards enabling software fault tolerant solutions comparable to hardware techniques.

7 Acknowledgments

The authors would like to thank the anonymous reviewers, Robert Cohn, Manish Vachharajani, Rahul Saxena, and the rest of the DRACO Architecture Research Group for their insightful comments and helpful discussion. This work is funded by Intel Corporation.

References

[1] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. Experimental evaluation of time-redundant execution for a brake-by-wire application. In *Proc. of DSN*, 2002.

[2] H. Ando and et al. A 1.3ghz fifth generation sparc64 microprocessor. In *Proceedings of the Conference on Design Automation*, 2003.

[3] A. Avizeinis. The n-version approach to fault-tolerance software. *IEEE Transactions on Software Engineering*, 11(12):1491–1501, December 1985.

[4] R. C. Baumann. Soft errors in commercial semiconductor technology: Overview and scaling trends. In *IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals*, pages 121_01.1 – 121_01.14, April 2002.

[5] E. D. Berger and B. G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *PLDI*, 2006.

[6] A. Borg, W. Blau, W. Graetcsh, F. Herrmann, and W. Oberle. Fault tolerance under unix. *ACM Transactions on Computer Systems*, 7(1):1–24, 1989.

[7] T. C. Bressoud. TFT: A Software System for Application-Transparent Fault-Tolerance. In *Proc. of the International Conference on Fault-Tolerant Computing*, 1998.

[8] T. C. Bressoud and F. B. Schneider. Hypervisor-based Fault-tolerance. In *Proc. of SOSP*, 1995.

[9] D. Bruening and S. Amarasinghe. Maintaining consistency and bounding capacity of software code caches. In *Proceedings of the International Symposium on Code Generation and Optimization*, March 2005.

[10] K. Echtle, B. Hinz, and T. Nikolov. On hardware fault diagnosis by diverse software. In *Proceedings of the Intl. Conference on Fault-Tolerant Systems and Diagnostics*, 1990.

[11] M. Gomaa and et al. Transient-fault recovery for chip multiprocessors. In *ISCA*, 2003.

[12] M. Gomaa and T. N. Vijaykumar. Opportunistic transient-fault detection. In *ISCA*, 2005.

[13] S. Hareland and et al. Impact of CMOS Scaling and SOI on Software Error Rates of Logic Processes. In *VLSI Technology Digest of Technical Papers*, 2001.

[14] M. Hiller. Executable assertions for detecting data errors in embedded control systems. In *Proc. of DSN*, 2000.

[15] M. Hiller and et al. On the placement of software mechanisms for detection of data errors. In *Proc. of DSN*, 2002.

[16] R. W. Horst and et al. Multiple instruction issue in the NonStop Cyclone processor. In *ISCA*, 1990.

[17] Z. Kalbarczyk, R. K. Iyer, S. Bagchi, and K. Whisnant. Chameleon: A software infrastructure for adaptive fault tolerance. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):560–579, 1999.

[18] T. Karnik and et al. Scaling Trends of Cosmic Rays Induced Soft Errors in Static Latches Beyond 0.18μ. In *VLSI Circuit Digest of Technical Papers*, 2001.

[19] T. Lovric. Dynamic double virtual duplex systems: A cost-efficient approach to fault-tolerance. In *Proceedings of the Intl. Working Conference on Dependable Computing for Critical Applications*, 1995.

[20] C.-K. Luk and et al. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[21] S. E. Michalak and et al. Predicting the Number of Fatal Soft Errors in Los Alamos National Laboratory's ASC Q Supercomputer. *IEEE Transactions on Device and Materials Reliability*, 5(3):329–335, September 2005.

[22] T. Moseley, A. Shye, V. J. Reddi, D. Grunwald, and R. V. Peri. Shadow profiling: Hiding instrumentation costs with parallelism. In *Proceedings of CGO*, 2007.

[23] S. S. Mukherjee and et al. Detailed design and evaluation of redundant multithreading alternatives. In *ISCA*, 2002.

[24] P. Murray, R. Fleming, P. Harry, and P. Vickers. Somersault: Software fault-tolerance. Technical report, HP Labs White Paper, Palo Alto, California, 1998.

[25] N. Oh and et al. Control-flow checking by software signatures. *IEEE Transactions on Reliability*, 51, March 2002.

[26] N. Oh and et al. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, 51, March 2002.

[27] K. Pattabiraman, Z. Kalbarczyk, and R. K. Iyer. Application-based metrics for strategic placement of detectors. In *Proceedings of 11th International Symposium on Pacific Rim Dependable Computing*, 2005.

[28] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *ISCA*, 2000.

[29] G. A. Reis and et al. SWIFT: Software implemented fault tolerance. In *CGO*, 2005.

[30] R. D. Schlichting and F. B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computing Systems*, 1(3):222–238, August 1983.

[31] M. A. Schuette, J. P. Shen, D. P. Siewiorek, and Y. K. Zhu. Experimental evaluation of two concurrent error detection schemes. In *Proceedings of FTCS-16*, 1986.

[32] T. J. Slegel and et al. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 1999.

[33] K. Sundaramoorthy, Z. Purser, and E. Rotenburg. Slipstream processors: improving both performance and fault tolerance. In *Proc. of ASPLOS*, 2000.

[34] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery using simultaneous multithreading. In *Proceedings of ISCA*, 2002.

[35] C. Wang, H. seop Kim, Y. Wu, and V. Ying. Compiler-managed software-based redundant multi-threading for transient fault detection. In *Proceedings of CGO*, 2007.

[36] N. Wang and et al. Y-Branches: When you come to a fork in the road, take it. In *PACT*, 2003.

[37] C. Weaver and et al. Techniques to reduce the soft error rate of a high-performance microprocessor. In *ISCA*, 2004.

[38] J. H. Wensley and et al. SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control. *Proceedings of the IEEE*, 66(10):1240–1255, October 1978.

[39] Y. Yeh. Triple-triple redundant 777 primary flight computer. In *Proceedings of the 1996 IEEE Aerospace Applications Conference*, volume 1, pages 293–307, February 1996.

[40] J. Ziegler and et al. IBM experiments in soft fails in computer electronics (1978 - 1994). *IBM Journal of Research and Development*, 40(1):3–18, January 1996.