PLR: A Software Approach to Transient Fault Tolerance for Multicore Architectures

Alex Shye, *Student Member*, *IEEE*, Joseph Blomstedt, *Student Member*, *IEEE*, Tipp Moseley, Vijay Janapa Reddi, *Student Member*, *IEEE*, and Daniel A. Connors, *Member*, *IEEE*

Abstract—Transient faults are emerging as a critical concern in the reliability of general-purpose microprocessors. As architectural trends point toward multicore designs, there is substantial interest in adapting such parallel hardware resources for transient fault tolerance. This paper presents *process-level redundancy* (PLR), a software technique for transient fault tolerance, which leverages multiple cores for low overhead. 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 uses a *software-centric* approach to transient fault tolerance, which shifts the focus from ensuring correct hardware execution to ensuring correct software execution. As a result, many benign faults that do not propagate to affect program correctness can be safely ignored. A real prototype is presented that is designed to be transparent to the application and can run on general-purpose single-threaded programs without modifications to the program, operating system, or underlying hardware. The system is evaluated for fault coverage and performance on a four-way SMP machine and provides improved performance over existing software transient fault tolerance techniques with a 16.9 percent overhead for fault detection on a set of optimized *SPEC2000* binaries.

Index Terms—Fault tolerance, reliability, transient faults, soft errors, process-level redundancy.

1 Introduction

Transient faults, also known as soft errors, are emerging as a critical concern in the reliability of computer systems [1], [2]. 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 [3], [4]. As the number of available transistors per chip continues to grow exponentially, the error rate 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 percent to 30 percent additional logic to add redundancy to mainframe processors and cover upward of 200,000 latches [5], [6]. Other approaches include specialized machines with custom hardware and software redundancy [7], [8].

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

- **Application specific constraints.** In ultrareliable environments, such as spacecraft systems, the result of a transient error can be the difference between life and death. For general-purpose computing, the consequences of faulty execution may greatly vary. While a fault during the execution of bank transaction software would be disastrous, there are many cases in which the result of a fault is much less severe. For instance, in graphics processing or audio decode and playback, a fault results in a mere glitch, which may not even be noticed by the user. Thus, the focus for reliability shifts from providing a bulletproof 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

- *A. Shye is with the Technological Institute L458, 2145 Sheridan Road, Evanston, IL 60208. E-mail: shye@northwestern.edu.*
- *J. Blomstedt is with the Department of Computer Science, University of Colorado, Campus Box 430, Boulder, CO 80309. E-mail: Joseph.Blomstedt@colorado.edu.*
- *T. Moseley is with the Deptartment of Computer Science, University of Colorado, 430 UCB, Boulder, CO 80309-0430. E-mail: tipp.moseley@colorado.edu.*
- *V.J. Reddi is with the Electrical Engineering and Computer Science Department, Harvard University, 345 Harvard St. Apt. 1D, Cambridge, MA 02138. E-mail: vj@eecs.harvard.edu.*
- *D.A. Connors is with the Department of Electrical and Computer Engineering, University of Colorado, Campus Box 425, Boulder, CO 80309. E-mail: dconnors@colorado.edu.*

Manuscript received 27 May 2008; accepted 24 Sept. 2008; published online 29 Oct. 2008.
For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number TDSC-2008-05-0089.
Digital Object Identifier no. 10.1109/TDSC.2008.62.

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 design and fabrication. For example, during the deployment of the ASC Q supercomputer, the scientists at the Los Alamos National Laboratory documented a high incidence of failures due to transient faults [2]. Also, Sun Microsystems documented a case in which faults to unprotected memories have caused system crashes at customer sites [1]. It is not difficult to imagine history repeating itself with unforeseen problems during hardware design manifesting themselves after deployment. Likewise, conditions such as altitude, temperature, and age can cause higher fault rates [9]. In these cases, it is necessary to employ reliability techniques, which are able to augment systems after deployment.

With such pressures driving general-purpose computing, 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 [10], control flow [11], or both [12]. The compiler-based software techniques suffer from a few limitations. First, the execution of the inserted instructions and assertions decreases performance ($\sim 1.4\times$ slowdown [12] for fault detection). 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 the *process-level redundancy* (PLR), a software-implemented technique for transient fault tolerance. PLR creates a set of redundant processes per original application process and compares their output to ensure correct execution. The redundant processes can be freely scheduled by the operating system (OS) to available parallel hardware resources. PLR scales with the architectural trend toward 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.

This paper makes the following contributions:

- Introduces a *software-centric* paradigm of transient fault tolerance that 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 replication* (SoR) 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. A software-centric approach is able to ignore many of these benign faults.
- Presents a real prototype PLR system that operates transparently to the application and leverages multiple general-purpose microprocessor cores for transient fault tolerance. We evaluate the fault coverage and performance of the prototype and find that it runs a set of the *SPEC2000* benchmark suite with only a 16.9 percent overhead on a four-way SMP system. This represents a significant performance improvement over previous software-implemented transient fault tolerance techniques.
- Maintaining determinism between redundant processes is the biggest challenge in implementing PLR. We present and evaluate software-only approaches for deterministically handling asynchronous signals and shared memory accesses across the redundant processes. We also evaluate the performance impact of using these techniques for maintaining determinism.

The rest of this paper is organized as follows: Section 2 provides background on transient fault tolerance. Section 3 describes the software-centric fault detection model, and Section 4 describes the PLR architecture. Section 5 shows results from the PLR prototype. Section 6 discusses related work. Section 7 concludes this paper.

2 Background

In general, a fault can be classified by its effect on system execution into the following categories [13]:

- **Benign fault.** A transient fault that 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 [14]. Wang et al. [15] shows that less than 15 percent of faults injected into a register transfer level (RTL) model of a processor result in software visible errors indicating that many transient faults are benign faults.
- **Silent data corruption (SDC).** A transient fault that is undetected and propagates to corrupt program output is considered an SDC. This is the worst case scenario where an application appears to execute correctly but silently produces incorrect output.
- **Detected unrecoverable error (DUE).** A transient fault that is detected without the possibility of recovery is considered a DUE. DUEs can be split into two categories. A *true DUE* occurs when a fault that would propagate to incorrect execution is detected. A *false DUE* occurs when a benign fault is detected as a fault. Without recovery, a false DUE will cause the system to unnecessarily halt execution, and with recovery, a false DUE will cause unwarranted invocations to the recovery mechanism.

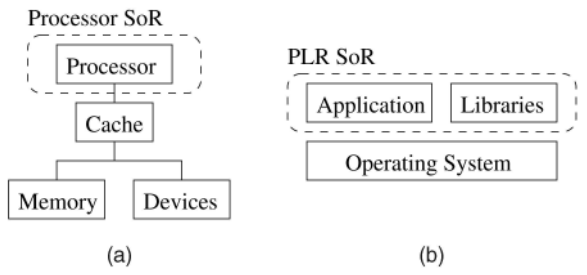

Fig. 1. Hardware-centric and software-centric transient fault detection models. A software-centric model (e.g., PLR) views the system as the software layers and places the sphere of influence around particular software layers. (a) Hardware-centric. (b) Software-centric.

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 raising a trap). 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 that would be false DUEs may cause unwarranted invocations to the recovery mechanism.

3 Software-Centric Fault Detection

The SoR [16] 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 that enters the SoR is replicated and all executions within the SoR are redundant in 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.

The original concept of the SoR is used for defining the boundary of reliability in redundant hardware design. We call this traditional model a *hardware-centric* fault detection model that uses a hardware-centric SoR. The hardware-centric model views the system as a collection of hardware components, which must be protected from transient faults. In this model, the 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. The reason is that 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 [12] places its SoR around the processor, as shown in Fig. 1a. 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 that must execute correctly. A software-centric model uses a software-centric SoR that is placed around software layers, instead of hardware components. Defining the SoR in terms of software provides software-implemented technique with more natural boundaries for fault detection. Also, the software-centric mode makes this key insight: although faults occur at the hardware level, *the only faults that matter are the faults that 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. Benign faults are safely ignored. A software-centric system with only detection covers errors that would propagate into incorrect software output as DUEs. A software-centric system with both detection and recovery will not need to invoke the recovery mechanism for faults that do not affect correctness.

Fig. 1b shows an example of 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 inputs are replicated, execution within the SoR is redundant, and data leaving the SoR are compared.

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 that are hardware-centric are ignoring the potential strengths of a software approach. In Section 5.1, we demonstrate PLR's ability to ignore many benign faults through a fault injection campaign.

4 Process-Level Redundancy (PLR)

PLR is a software approach to transient fault tolerance that is designed to run transparently on general-purpose applications without modifications to the application, OS, or underlying hardware. These specific characteristics are described in detail below:

- **Transparency.** PLR operates transparently to the user and the application. Even though PLR creates multiple redundant processes per original application process, it maintains all user-expected process semantics. The application is also unaware of PLR and does not need to be modified or recompiled to run with PLR.
- **Software-implemented.** PLR is implemented entirely in software and runs in user space under the application. In this manner, PLR is able to provide transient fault tolerance without requiring modifications to the OS or underlying hardware. In addition, software implementation makes PLR extremely

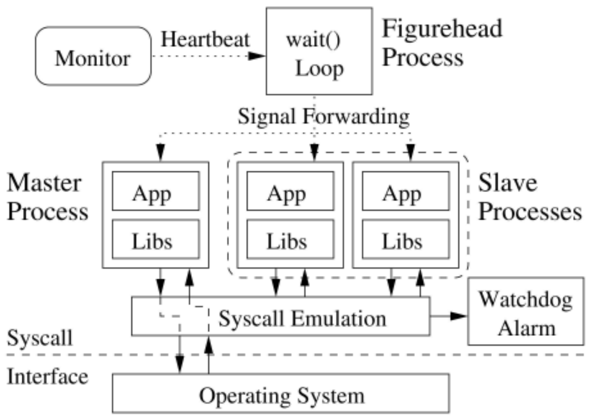

Fig. 2. Overview of the PLR system architecture with three redundant processes.

flexible. Applications that must be reliable can be run with PLR, while other applications run regularly.

- **Software-centric.** PLR uses a software-centric approach to fault detection with an SoR around the user-space application and its associated shared libraries. All user-space executions are redundant and faults are only detected if they result in incorrect data exiting user space. This extends the checking boundaries for fault detection as compared to most other transient fault tolerance techniques and allows PLR to ignore many benign faults.
- **Replica-based.** PLR uses process replicas to provide redundancy. PLR replicates the entire process virtual address space as well as the process metadata such as file descriptors. In addition, PLR automatically creates and coordinates among the redundant processes to maintain determinism among the processes, detect transient faults, and recover from detected faults. Operating at the process level has a distinct advantage in that processes are also a basic abstraction of the OS. Therefore, PLR can leverage multiple hardware resources such as extra hardware threads or cores by simply allowing the OS to schedule the replicas across all available hardware resources.

4.1 PLR Overview

An overview of the PLR system is shown in Fig. 2. PLR gains control of an application before it begins to execute and begins its initialization phase. First, PLR creates a *monitor process* and then initializes metadata including a shared memory segment used for interprocess communication. Then, PLR forks the application N times, with $N = 2$ as the minimum for fault detection and $N = 3$ as the minimum for fault detection and fault recovery. These processes are the *redundant processes*, which actually perform the execution of the application. One of the redundant processes is labeled the *master process* and the others are labeled the *slave processes*. During execution of the redundant processes, the *system call emulation unit* coordinates system I/O among the redundant processes. In general, the master process is allowed to perform system I/O while the slave processes emulate system I/O. The system call emulation unit also enforces the software-centric fault detection model and implements transient fault detection and recovery. A *watchdog timer* is attached to the system call emulation unit, which is used to detect cases in which a fault causes one of the redundant processes to hang indefinitely. After initialization and the redundant processes are created, the original process becomes a *figurehead process*. The figurehead process does not do any real work. It only waits for the redundant processes to finish execution and forwards signals to the redundant processes.

The following sections describe the PLR system in further detail and discuss issues with transparency, maintaining determinism among process replicas, and transient fault detection and recovery. While describing the implementation details and challenges, we attempt to stay as general as possible, but because of our specific system implementation, we often use IA32-specific and Linux-specific features and terminology. However, other OSs and computer architectures typically have their own equivalents and we believe our experience can mostly be translated to other systems.

4.2 Maintaining Process Semantics

PLR creates an entire group of processes per original application process. However, it is important to maintain the process semantics expected by the user and other applications in the case of interprocess communication. Specifically, we are interested in maintaining the following expected process semantics when running an application:

1. When an application is invoked, it is given a specific process identifier (pid). The pid exists during the entire duration of the application and is relinquished to the OS afterward.
2. When the application exits, it returns with the correct exit code of the program.
3. A signal that sent a valid pid will have the intended effects. For example, a SIGKILL will kill the process.

In a previous version of PLR, we simply fork the application multiple times and compare execution behavior using the system call emulation unit [17]. However, when assessing the transparency of such an approach, we quickly realized that it violates process semantics. Suppose an original application begins with a pid of 100 and we fork twice for redundant processes with pids of 101 and 102. If a transient fault causes the original pid of 100 to die, it is impossible to maintain process semantics; although execution of the application continues in pids of 101 and 102, the original pid of 100 does not exist during execution, does not return the correct return code, and is impossible to signal.

In order to maintain process semantics, PLR uses a *figurehead process*. After creating the redundant processes, the original process is relegated to a figurehead process. The figurehead performs three functions, which match the three rules of expected process semantics listed above. First, it sleeps and waits on the redundant processes to complete execution. Second, upon completion, it receives the application's exit value from the system call emulation unit and exits properly. Third, it performs *signal forwarding*. Because every signal intended for the application reaches the figurehead process (it has the correct pid), the figurehead process needs to forward the signals to the children. Thus, a SIGTERM will cause the figurehead process, as well as the redundant processes to all terminate execution and ensure the third rule in maintaining process semantics.

There is one complication on Linux systems with the signal forwarding in the figurehead process; the SIGKILL and SIGSTOP signals cannot be caught with a signal handler. Therefore, with just a figurehead process, a SIGKILL would kill the figurehead process but leave the redundant processes running. To handle this, PLR uses a *monitor process*, which intermittently polls the state of the figurehead process. If the process does not exist, it assumes a SIGKILL and kills itself, along with the rest of the redundant processes. If the parent is in a stopped state, it issues a SIGSTOP to all the redundant processes to emulate the effect of a SIGSTOP signal within the application. The figurehead and monitor processes introduce a delay in receiving signals. However, signals are mostly asynchronous by nature and a slight delay is not a problem.

It should be noted that maintaining process semantics helps with transparency from the user perspective and from the perspective of other applications that may interact with the target application via interprocess communication. PLR does not maintain any transparency from the system perspective. For example, a listing of the active system processes will produce figurehead process, as well as the monitor process and redundant processes.

4.3 Process Replicas

PLR is a technique that uses the software-centric model of transient fault detection. As shown in Fig. 1b, 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, and so forth. Everything outside of the SoR, namely the OS, must be protected by other means. Any data that enter 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 multicore 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.

During execution, one of the redundant 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 system call emulation unit performs the input replication, 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 the redundant processes 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 that alter any system state are only executed once by the master process. The slave processes emulate the system call.
- Execution among the redundant processes must be deterministic. System calls that return nondeterministic data, such as a request for system time or resources, must be emulated to ensure all redundant processes use the same data for computation. Other sources of determinism such as asynchronous interrupts and shared memory accesses must also be intercepted and emulated.
- 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.

4.4 Input Replication

As the SoR model dictates, any data that enters the SoR must be replicated to ensure that all data is redundant within the SoR. Any data that passes into the processes is received once by the master process and then passed to the slave processes. During system call emulation, any read data (such as a read from a file descriptor) is replicated and copied to the slave processes. Also, the return value from all system calls is considered as input data and is also replicated.

4.5 Output Comparison

All data that exit the redundant processes must be compared for correctness before proceeding out of the SoR. If the output data do not match, a transient fault is detected and a recovery routine is invoked. During system call emulation, any write buffers that 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 that leaves the SoR and must also be checked to verify program correctness.

Most output comparisons deal with small amounts of data and are accomplished by simply copying the data to a shared memory segment and comparing the bytes. For larger write buffers, such as the write buffer when checking the `write()` system call, a 32-bit CRC is computed locally, and then the CRC value is compared through shared memory. The local CRC computation avoids copying large chunks of memory through shared memory, which can significantly increase overhead.

4.6 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 during the output comparison of the system call parameters, the type of system call is also 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 that modify any system state, such file renaming and linking, are only executed once by the master process. 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.

4.7 Shared Memory

Shared memory accesses present a source of potential nondeterminism between redundant processes. A read from shared memory represents input data that should be replicated, and a write to shared memory represents output data that should be compared for correctness. The problem is that shared memory accesses masquerade as arbitrary load and/or store instructions. Therefore, only handling system I/O through the system call interface will not suffice. Memory-mapped device I/O shares the same problem as shared memory accesses.

PLR handles shared memory I/O by borrowing the *trap-and-emulate* technique from virtual machines and dynamic binary optimizers. Virtual machines trap on the execution of privileged instructions and defer to a virtual machine monitor to emulate the privileged instruction [18]. Dynamic binary optimizers, which interpret/recompile the application and execute out of a code cache, use the same technique for detecting self-modifying code. In this case, the text section is marked read-only and any self-modifying code will cause an immediate trap to notify the dynamic binary optimizer [19]. Along these lines, there are two ways to handle shared memory accesses:

- **Trap-and-emulate.** PLR treats any system calls regarding shared memory regions as a special case. These include shared memory calls such as `shmat()` or shared memory mapping functions such as `mmap()` called with the MAP_SHARED flag. While emulating these system calls, PLR also performs two extra functions. First, PLR updates a *shared memory map* (SMM), which includes all of the shared memory regions including metadata such as the protection mode of the pages. Second, PLR switches the protection of all of the shared memory pages to disallow both reading and writing. Upon a read or write to one of the shared memory regions, a trap will occur. The trap handler in PLR decodes and begins to emulate the instruction at which the trap occurred. If the instruction accesses data within the SMM, the emulation continues along with the correct input replication and/or output comparison. If not, the original trap handler is invoked.
- **Trap-and-emulate-and-probe.** The trap-and-emulate approach incurs a high overhead by requiring a trap to the OS on every read or write instruction to a shared memory region. The *trap-and-emulate-and-probe* approach avoids this overhead by placing a *probe* at the offending instruction after a trap and emulation of a specific instruction. A probe is simply a branch instruction that overwrites the original instruction and branches to emulation code. Emulation begins by using the memory access address to look up into the SMM. If the instruction accesses a shared memory region, it is emulated appropriately with input replication and output comparison. If not, a copy of the original instruction is executed and control branches back to the original program after the probe. Trap-and-emulate-and-probe pays a one-time cost of a trap and then avoids the trap on subsequent executions of the same instruction.

4.8 Asynchronous Signals

Asynchronous signals present another form of potential nondeterminism among the redundant processes. The figurehead process takes care of the first part of signal processing; it provides the correct pid to signal and then forwards all of the signals it receives to the redundant processes. However, there is still a problem during execution of the signal handlers in the redundant processes. The problem is that signal handlers are called asynchronously and may read or write any data in the process's address space. If the redundant processes do not all call the signal handlers at precisely the same point in their dynamic instruction streams, they may become nondeterministic.

To handle asynchronous signals, PLR inserts probes and marks specific points in the code as *epoch* boundaries [20], [21]. An epoch is a timeslice of a program in which the start and end points are known and identical across the redundant processes. Each redundant process maintains a local epoch counter, which stores the number of epoch boundaries passed during execution. Signal handling is deferred and handled at epoch boundaries.

Specifically, signal handling proceeds as follows:

1. The figurehead process represents the entire group of processes (it has the correct pid) and receives the asynchronous signal.
2. The figurehead process sends a SIGSTOP to all the redundant processes to temporarily stop their execution.
3. The figurehead inspects the epoch counter of each redundant process. A pending signal is set up to be serviced at an epoch count equal to the largest current epoch counter plus one.
4. The figurehead processes resume the execution of all redundant processes.
5. The redundant processes execute and, at each epoch boundary, check if there is a pending signal to be handled. If there is a pending signal, and the current epoch count matches the epoch count set for signal handling, then the redundant process transfers control to the signal handler.

There exists a tradeoff between program overhead and the timeliness of signal handling. For example, if epoch boundaries are placed at each instruction, signals will be handled immediately, but performance will significantly degrade due the checking instructions at epoch boundaries. On the other hand, if epochs are too large, performance impact will be negligible, but the delay until signal handling may not meet user expectations.

We have implemented three policies for the placement of epoch boundaries; at each system call (*SYSCALL*), function call (*FUNC*), or backward branch (*BACK_BRANCH*). Different policies will be suitable for different application types. For example, the *SYSCALL* policy will work well for an application that frequently uses system calls. Applications that have many function calls, such as programs developed with object-oriented languages, would work well with the *FUNC* policy. Compute-intensive programs that execute in tight kernels within a function will need a fine-grained approach such as the *BACK_BRANCH* approach. The three policies are managed with a command line switch specifying the policy to use.

4.9 Transient Fault Detection

A transient fault is detected in one of three ways:

1. **Output mismatch.** A transient fault that propagates to cause incorrect output will be detected with the output comparison within the emulation unit at the point that the data are about to exit the SoR. An output mismatch may occur during system call emulation or during the handling of an instruction that accesses shared memory.
2. **Watchdog time-out.** 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 time-out 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 time-out. A drawback to the watchdog alarm is that a time-out period exists in which the application does not make any progress. In our experience, on an unloaded system, a time-out of 1-2 seconds is sufficient. The time-out value is user specified and can be increased on a loaded system. On a loaded system, spurious time-outs 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, and so forth. Signals handlers are set up to catch the corresponding signals and an error is flagged. The next time the emulation unit is called, it can immediately begin the recovery process.

4.10 Transient Fault Recovery

Transient fault recovery mechanisms typically fit into two broad categories: *checkpoint and repair* and *fault masking*. Checkpoint and repair techniques involve 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 time-out.** As mentioned in Section 4.1, there are two cases for a watchdog time-out. In the first case, where a faulty process is calling the emulation unit while the other processes continue executing, there will only be one process in the emulation unit during time-out. 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 time-out. 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.

4.11 Windows of Vulnerability

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

To maintain process semantics, it is critical that the figurehead stays alive throughout program execution. Although it represents a single point of failure, the figurehead performs almost no real work and the probability of a transient fault affecting its execution is very low. A single monitor process also represents a single point of failure.

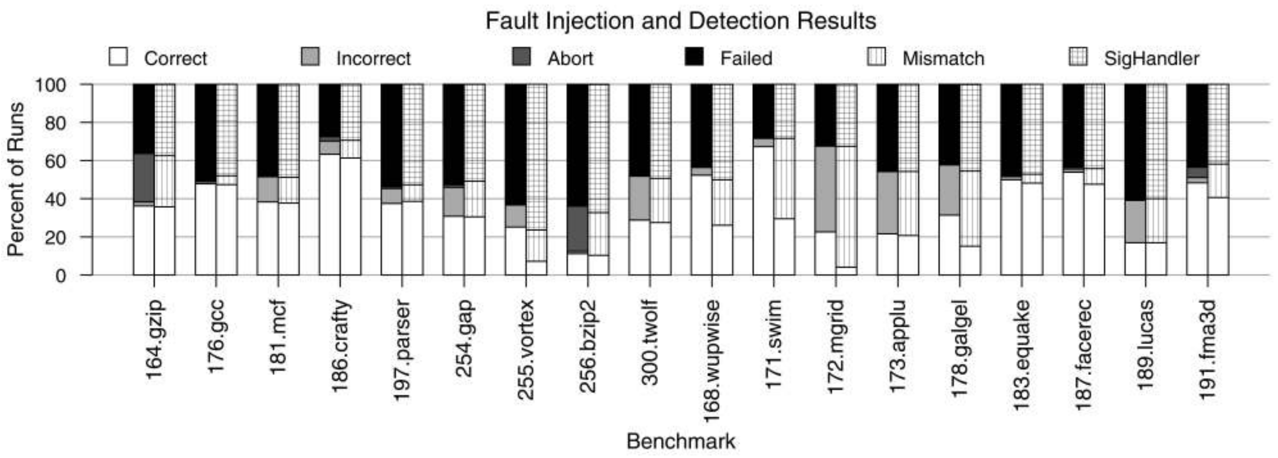

Fig. 3. Results of the fault injection campaign. The left bar in each cluster shows the outcomes with just fault injection and the right bar shows the breakdown of how PLR detects the faults.

However, if the monitor process is a concern, it can easily be replicated.

All fault tolerance techniques have windows of vulnerability, which are usually associated with faults to the checker mechanism. Although not completely reliable, partial redundancy [22], [23] may be sufficient to improve reliability enough to meet user or vendor reliability standards.

5 Experimental Results

This paper presents and evaluates a PLR prototype built using the Intel Pin dynamic binary instrumentation system [24]. 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 [12]. One thousand runs are executed per benchmark. To maintain manageable runtimes, the test inputs are used for fault injection and fault propagation experiments. For each run, a dynamic 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.

PLR performance is evaluated using the *SPEC2000* reference inputs. Performance is measured by running the prototype with both two and three redundant processes without fault injection on a four-processor SMP system; specifically, the system has four 3.00-GHz Intel Xeon MP processors, each with 4,096-Kbyte L3 cache, has 6 Gbytes of systemwide memory, and is running Red Hat Enterprise Linux AS release 4.

5.1 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. Fig. 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 given as follows:

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

Time-outs of the watchdog alarm are ignored because they occur very infrequently ($\sim$ 0.05 percent 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 that do not affect correctness are generally not detected in PLR, thereby avoiding false positives. In contrast, SWIFT [12], which is currently the most advanced compiler-based approach, detects roughly $\sim$ 70 percent 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*,

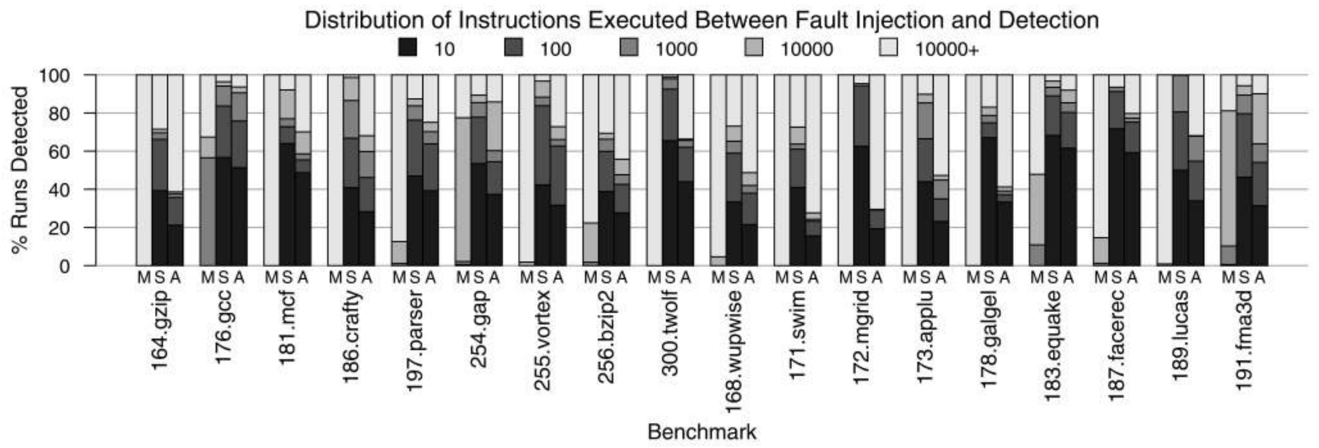

Fig. 4. Distribution of the number of executed instructions between the injection and detection of a fault. Percentages are normalized to all the runs that are detected via output mismatch (*M*), program failure (*S*), or both combined (*A*).

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.

5.2 Fault Propagation

Fig. 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 Fig. 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 *SigHandler*.

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 *SigHandler* 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. However, these issues are outside the scope of this paper.

5.3 Performance Results

Performance is evaluated using two redundant processes for fault detection (PLR2) and three processes to support recovery (PLR3). Fig. 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 percent overhead for PLR2 and 15.2 percent overhead for PLR3. On `-O2` programs, PLR2 incurs a 16.9 percent overhead for PLR2 and 41.1 percent overhead for PLR3. Overhead in PLR is due to the fact that multiple redundant processes are contending for system resources. Programs that place higher demands on system 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 Section 5.4.

5.4 PLR Overhead Breakdown

The performance overhead of PLR consists of *contention overhead* and *emulation overhead*, shown as stacked bars in Fig. 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. Note that this overhead is purely from the redundant processes. The figurehead and monitor processes perform little computation and their

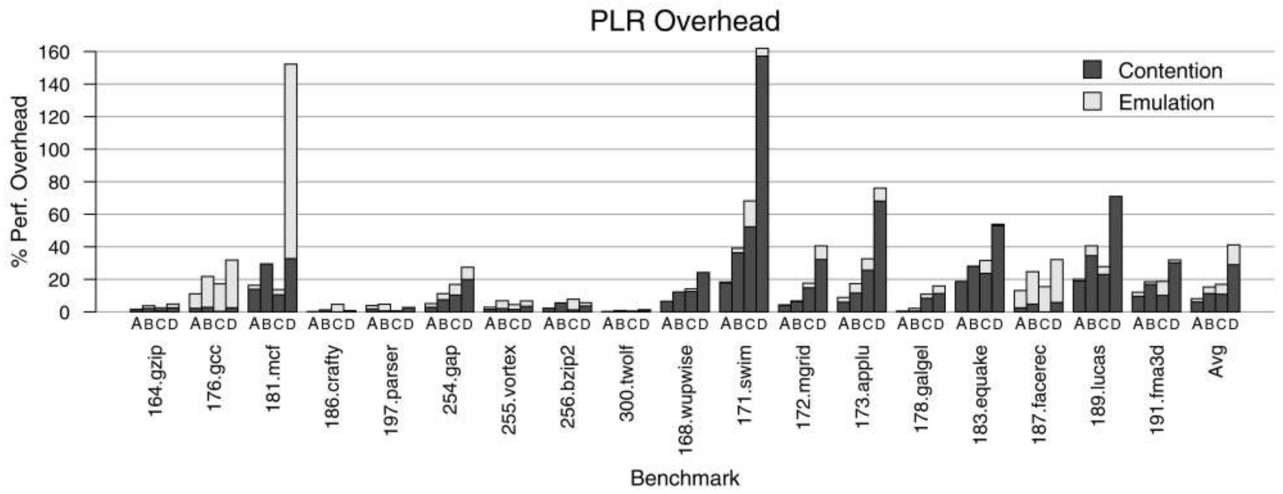

Fig. 5. Overhead of running PLR on a set of both unoptimized and optimized *SPEC2000* benchmarks. The combinations of runs include `-O0` compiled binaries with PLR2 (*A*), `-O0` with PLR3 (*B*), `-O2` with PLR2 (*C*), and `-O2` with PLR3 (*D*).

performance overhead is negligible. 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.

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.

5.4.1 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. Fig. 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 effect on the contention overhead. With less than 5 million L3 cache misses per second, the contention overhead is minimal. However, beyond that point, the contention overhead increases greatly. At 10 million L3 cache misses per second, PL2 incurs a 13 percent overhead and PLR3 incurs a 25 percent overhead. 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.

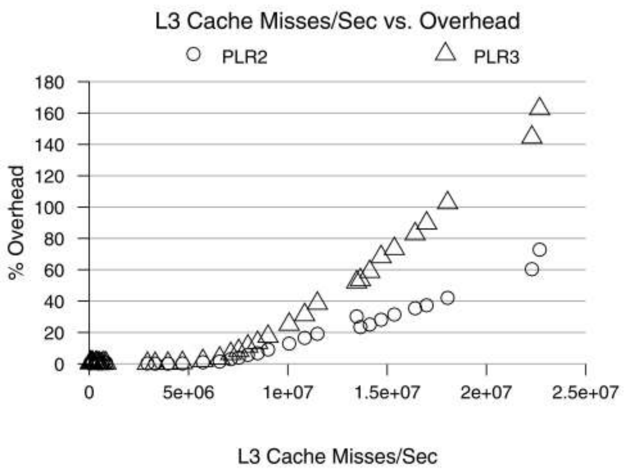

Fig. 6. PLR contention overhead for varying L3 cache miss rates.

5.4.2 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 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 10 times a second and writes a user-specified number of bytes per system call. For each `write()` call, the emulation unit transfers and compares the write data in shared memory.

Fig. 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 percent 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.

Fig. 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 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 1 Mbyte/second but then increases substantially after that point for both PLR2 and PLR3.

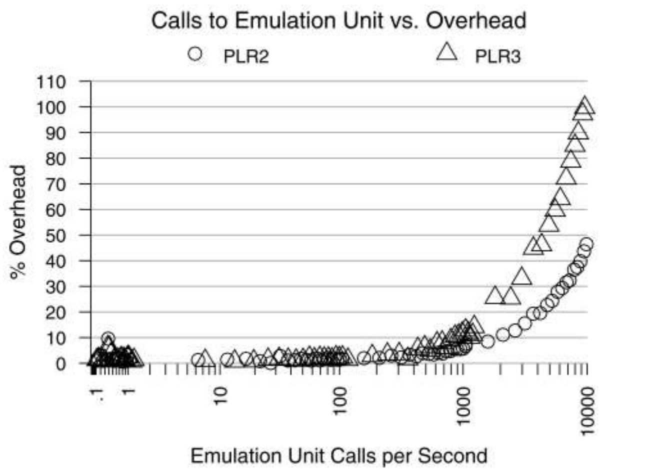

Fig. 7. PLR overhead for varying system call rates demonstrating the synchronization and emulation overhead for a simple system call.

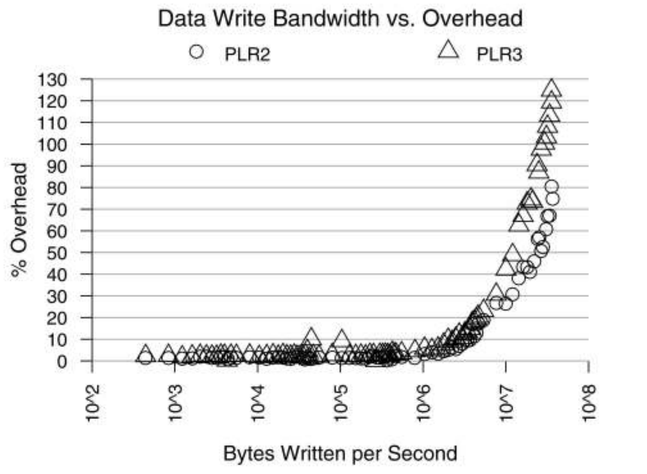

Fig. 8. PLR overhead for various data bandwidths demonstrating the overhead of comparing shared memory during output comparison.

5.5 Shared Memory Support

To measure the performance of supporting shared memory, we have developed a synthetic benchmark that periodically writes to a shared memory region. Fig. 9 shows the additional performance overhead when running the synthetic benchmark at various shared memory write rates for PLR running with two redundant processes using trap-and-emulate and trap-and-emulate-and-probe. With a very low rate of shared memory accesses, lower than 1,000 accesses per second, the performance is negligible. As the shared memory access rate increases to 10,000 accesses per second, the overhead increases to a reasonable 5 percent overhead. Higher access rates result in a large increase in performance overhead. It should be noted that the overhead shown here is purely the overhead due to supporting shared memory. In addition, as shared memory write rates increase, the trap-and-emulate-and-probe approach outperforms the trap-and-emulate approach. Overhead is mostly dominated by the synchronization and interprocess communication, but the use of the probe is able to reduce the trapping overhead.

5.6 Supporting Asynchronous Signals

PLR handles asynchronous signals by deferring signal handling within the redundant processes until epoch boundaries. This approach introduces a delay between

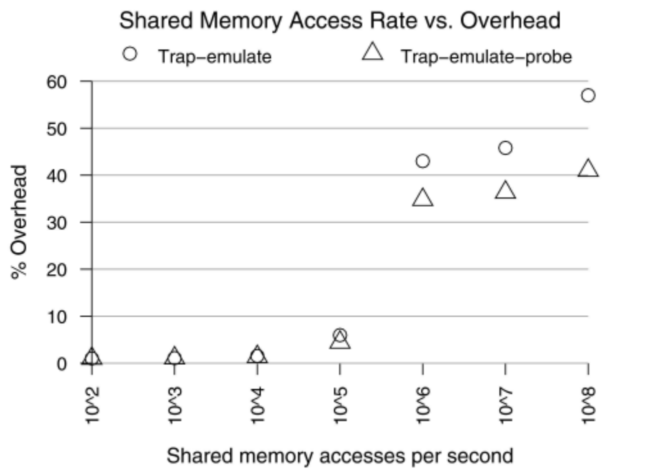

Fig. 9. Additional PLR overhead for supporting shared memory accesses.

receiving and handling asynchronous signals. Because these signals are asynchronous by nature, a slight delay is manageable. However, we would like to avoid large delays until signal handling, which may greatly impact the performance/behavior of the application.

To provide an idea of the delay period before signal handling, we study the number of instructions executed between various epochs within our benchmarks. Fig. 10 shows a breakdown of the number of instructions between epochs using the *SYSCALL*, *FUNC*, and *BACK_BRANCH* policies described in Section 4.8. *SYSCALL* is not a good policy in general. While *176.gcc* executes the most system calls, the majority of epochs still are larger than 1 billion instructions. *FUNC* performs much better, reducing most of the epochs to within a manageable range. However, a few of the benchmarks, such as *171.swim* and *172.mgrid*, still have a significant amount of large epochs. The reason is that these benchmarks execute long running loops within function and do not hit epoch boundaries often. Moving to the *BACK_BRANCH* policy removes this limitation and provides epochs within 1,000 instructions consistently.

We then analyze the additional performance overhead of handling signals on a set of our benchmarks with two redundant processes in Fig. 11. The graph shows the normalized overhead when using our three policies. Overall, the *SYSCALL* and *FUNC* policies incur a negligible overhead across all of the applications. The *BACK_BRANCH* policy incurs a higher overhead that varies across applications depending on the rate of backward branches per application. For example, a branch-intensive application such as *176.gcc* has nearly two times overhead. The other applications vary greatly with *300.twolf* incurring almost negligible overhead. Overall, the *FUNC* policy is the most attractive policy with the ability to handle signals within 100,000 instructions for most applications and with negligible performance overhead.

6 Related Work

PLR is similar to a software version of the hardware SMT and CMP extensions for transient fault tolerance [25], [26], [27], [16], [28]. PLR aims to provide the same functionality in software. Wang et al. [29] proposes a compiler infrastructure for software redundant multithreading, which achieves

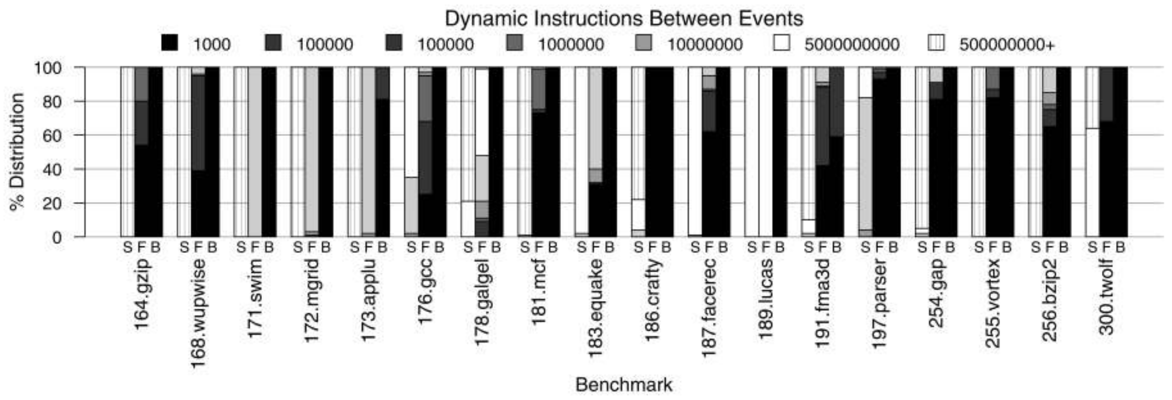

Fig. 10. The number of dynamic instructions executed between epochs of the three policies for supporting asynchronous signals: *SYSCALL* (S), *FUNC* (F), and *BACK_BRANCH* (B).

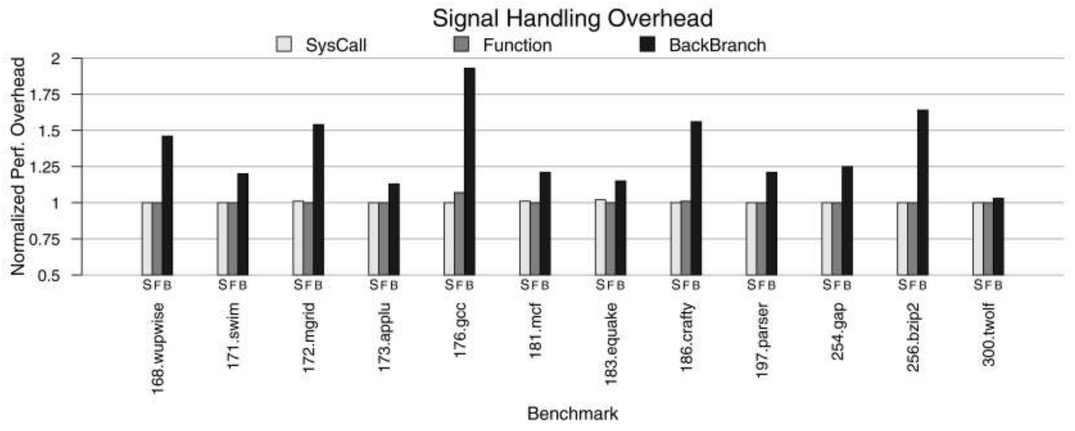

Fig. 11. Additional normalized PLR overhead incurred for supporting asynchronous system calls using epochs at the three policies: *SYSCALL* (S), *FUNC* (F), and *BACK_BRANCH* (B).

19 percent 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 a source code to operate.

Executable assertions [30], [31] and other software detectors [32] explore the placement of assertions within software. Other schemes explicitly check control flow during execution [33], [11]. The software-centric approach provides a different model for transient fault tolerance using a software equivalent to the commonly accepted SoR model. The pi bit [13] and dependence-based checking [34] have been explored as methods to follow the propagation of faults in an attempt to only detect faults that 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 [35], [20], [21], [36], [37], [8]. This body of work targets hard faults (such as hardware or power failures) and assumes fail-stop execution [38] 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 multicore systems.

More recently, process replicas have been proposed for general-purpose systems to provide services other than fault tolerance. DieHard [39] proposes using replica machines for tolerating memory errors and Exterminator [40] uses process replicas to probabilistically detect memory errors. DieHard and Exterminator briefly mention using process replicas and do not elaborate on the challenges of nondeterministic events. Shadow profiling [41] and SuperPin [42] propose using process replicas to parallelize dynamic binary instrumentation. Using process replicas for profiling has the advantage that correctness is not necessary for profiling—if the profile information correctly follows execution trends, it is good enough. As a result, they can get away with not completely handling nondeterministic events. Other projects such as FT-MPI [43] and MPI/FT [44] have extended MPI to implement process replicas on MPI applications for hard faults. PLR applies replicas for transient fault tolerance on general-purpose multicore machines. To the best of our knowledge, PLR is the most robust software implementation of general-purpose process replicas with the ability to deterministically handle shared memory accesses and asynchronous signals.

There have been a number of previous approaches to program replication. N-version programming [45] uses three different versions of an application for tolerating software errors. Aidemark uses a time redundant technique, which executes an application multiple times and uses majority voting [46]. Virtual duplex systems combine both N-version programming and time redundancy [47], [48]. The Tandem Nonstop Cyclone [7] is a custom system designed to use process replicas for transaction processing workloads.

Chameleon [49] is an infrastructure designed for distributed systems using various ARMOR processes to implement adaptive fault tolerance. The figurehead process is similar in some respects to the fault tolerant manager, the monitor process is similar to the heartbeat ARMOR, and the redundant processes are similar to the execution ARMORs. However, the systems are designed with different goals in mind. While Chameleon is for providing an adaptive and configurable fault tolerance on distributed systems, PLR is designed to provide transient fault tolerance on general-purpose multicore systems.

7 Conclusion

This paper has motivated the necessity for software transient fault tolerance for general-purpose microprocessors and proposed PLR as an attractive alternative in emerging multicore 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, OS, 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 that safely ignoring benign faults. Experimental results show that when running an optimized set of *SPEC2000* benchmarks on a four-way SMP machine, PLR provides fault detection with a 16.9 percent overhead. PLR performance improves upon existing software transient fault tolerance techniques and takes a step toward enabling software fault tolerant solutions with comparable performance to hardware techniques.

Acknowledgments

This work was supported by Intel. The authors would like to thank the anonymous IEEE TDSC reviewers and the anonymous reviewers at DSN for their comments on the preliminary work presented in this paper. The authors would also like to thank Robert Cohn, Manish Vachharajani, Rahul Saxena, and the rest of the DRACO Architecture Research Group for their insightful comments and helpful discussion.

References

[1] R.C. Baumann, "Soft Errors in Commercial Semiconductor Technology: Overview and Scaling Trends," *IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals,* pp. 121_01.1-121_01.14, Apr. 2002.

[2] S.E. Michalak et al., "Predicting the Number of Fatal Soft Errors in Los Alamos National Laboratory's ASC Q Supercomputer," *IEEE Trans. Device and Materials Reliability,* vol. 5, no. 3, pp. 329-335, Sept. 2005.

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

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

[5] T.J. Slegel et al., "IBM's S/390 G5 Microprocessor Design," *IEEE Micro,* vol. 19, no. 2, pp. 12-23, Mar./Apr. 1999.

[6] H. Ando et al., "A 1.3 GHz Fifth Generation Sparc64 Microprocessor," *Proc. 40th Conf. Design Automation (DAC '03),* pp. 702-705, 2003.

[7] R.W. Horst et al., "Multiple Instruction Issue in the Nonstop Cyclone Processor," *Proc. 17th Ann. Int'l Symp. Computer Architecture (ISCA),* 1990.

[8] Y. Yeh, "Triple-Triple Redundant 777 Primary Flight Computer," *Proc. 1996 IEEE Aerospace Applications Conf.,* vol. 1, pp. 293-307, Feb. 1996.

[9] J. Ziegler et al., "IBM Experiments in Soft Fails in Computer Electronics (1978-1994)," *IBM J. Research and Development,* vol. 40, no. 1, pp. 3-18, Jan. 1996.

[10] N. Oh et al., "Error Detection by Duplicated Instructions in Super-Scalar Processors," *IEEE Trans. Reliability,* vol. 51, no. 1, Mar. 2002.

[11] N. Oh et al., "Control-Flow Checking by Software Signatures," *IEEE Trans. Reliability,* vol. 51, no. 1, Mar. 2002.

[12] G.A. Reis et al., "SWIFT: Software Implemented Fault Tolerance," *Proc. Int'l Symp. Code Generation and Optimization (CGO),* 2005.

[13] C. Weaver et al., "Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor," *Proc. 31st Int'l Symp. Computer Architecture (ISCA),* 2004.

[14] N. Wang et al., "Y-Branches: When You Come to a Fork in the Road, Take It," *Proc. 12th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT),* 2003.

[15] N.J. Wang, J. Quek, T.M. Rafacz, and S.J. Patel, "Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline," *Proc. 2004 Int'l Conf. Dependable Systems and Networks (DSN '04),* pp. 61-72, June 2004.

[16] S.K. Reinhardt and S.S. Mukherjee, "Transient Fault Detection via Simultaneous Multithreading," *Proc. 27th Ann. Int'l Symp. Computer Architecture (ISCA),* 2000.

[17] A. Shye, T. Moseley, V.J. Reddi, J. Blomstedt, and D.A. Connors, "Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance," *Proc. 37th Int'l Conf. Dependable Systems and Networks (DSN '07),* June 2007.

[18] J. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes.* Morgan Kaufmann, 2005.

[19] D. Bruening and S. Amarasinghe, "Maintaining Consistency and Bounding Capacity of Software Code Caches," *Proc. Int'l Symp. Code Generation and Optimization (CGO '05),* Mar. 2005.

[20] T.C. Bressoud and F.B. Schneider, "Hypervisor-Based Fault-Tolerance," *Proc. 15th ACM Symp. Operating System Principles (SOSP),* 1995.

[21] T.C. Bressoud, "TFT: A Software System for Application-Transparent Fault-Tolerance," *Proc. Int'l Conf. Fault-Tolerant Computing,* 1998.

[22] M. Gomaa and T.N. Vijaykumar, "Opportunistic Transient-Fault Detection," *Proc. 32nd Int'l Symp. Computer Architecture (ISCA),* 2005.

[23] K. Sundaramoorthy, Z. Purser, and E. Rotenburg, "Slipstream Processors: Improving Both Performance and Fault Tolerance," *Proc. Ninth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* 2000.

[24] C.-K. Luk et al., "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI),* 2005.

[25] M. Gomaa et al., "Transient-Fault Recovery for Chip Multiprocessors," *Proc. 30th Int'l Symp. Computer Architecture (ISCA),* 2003.

[26] S.S. Mukherjee et al., "Detailed Design and Evaluation of Redundant Multithreading Alternatives," *Proc. 29th Int'l Symp. Computer Architecture (ISCA),* 2002.

[27] Z. Purser, K. Sundaramoorthy, and E. Rotenberg, "A Study of Slipstream Processors," *Proc. 33rd Ann. ACM/IEEE Int'l Symp. Microarchitecture (MICRO '00),* pp. 269-280, 2000.

[28] E. Rotenberg, "AR-SMT: A Microarchitectural Approach to Fault Tolerance," *Proc. 29th Ann. Int'l Symp. Fault-Tolerant Computing (FTCS-29 '99),* pp. 84-95, 1999.

[29] C. Wang, H. seop Kim, Y. Wu, and V. Ying, "Compiler-Managed Software-Based Redundant Multi-Threading for Transient Fault Detection," *Proc. Int'l Symp. Code Generation and Optimization (CGO),* 2007.

[30] M. Hiller, "Executable Assertions for Detecting Data Errors in Embedded Control Systems," *Proc. Int'l Conf. Dependable Systems and Networks (DSN),* 2000.

[31] M. Hiller et al., "On the Placement of Software Mechanisms for Detection of Data Errors," *Proc. Int'l Conf. Dependable Systems and Networks (DSN),* 2002.

[32] K. Pattabiraman, Z. Kalbarczyk, and R.K. Iyer, "Application-Based Metrics for Strategic Placement of Detectors," *Proc. 11th Int'l Symp. Pacific Rim Dependable Computing (PRDC),* 2005.
[33] M.A. Schuette, J.P. Shen, D.P. Siewiorek, and Y.K. Zhu, "Experimental Evaluation of Two Concurrent Error Detection Schemes," *Proc. Int'l Symp. Fault-Tolerant Computing (FTCS-16),* 1986.
[34] T.N. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient-Fault Recovery Using Simultaneous Multithreading," *Proc. 29th Int'l Symp. Computer Architecture (ISCA),* 2002.
[35] A. Borg, W. Blau, W. Graetcsh, F. Herrmann, and W. Oberle, "Fault Tolerance under Unix," *ACM Trans. Computer Systems,* vol. 7, no. 1, pp. 1-24, Feb. 1989.
[36] P. Murray, R. Fleming, P. Harry, and P. Vickers, "Somersault: Software Fault-Tolerance," technical report, HP Labs White Paper, Palo Alto, CA, 1998.
[37] J.H. Wensley et al., "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," *Proc. IEEE,* vol. 66, no. 10, pp. 1240-1255, Oct. 1978.
[38] R.D. Schlichting and F.B. Schneider, "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems," *ACM Trans. Computing Systems,* vol. 1, no. 3, pp. 222-238, Aug. 1983.
[39] E.D. Berger and B.G. Zorn, "DieHard: Probabilistic Memory Safety for Unsafe Languages," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI),* 2006.
[40] G. Novark, E.D. Berger, and B.G. Zorn, "Exterminator: Automatically Correcting Memory Errors," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '07),* June 2007.
[41] T. Moseley, A. Shye, V.J. Reddi, D. Grunwald, and R.V. Peri, "Shadow Profiling: Hiding Instrumentation Costs with Parallelism," *Proc. Int'l Symp. Code Generation and Optimization (CGO),* 2007.
[42] S. Wallace and K. Hazelwood, "Superpin: Parallelizing Dynamic Instrumentation for Real-Time Performance," *Proc. Int'l Symp. Code Generation and Optimization (CGO '07),* Mar. 2007.
[43] G.E. Fagg, E. Gabriel, Z. Chen, T. Angskun, G. Bosilca, J. Pjesivac-Grbovic, and J.J. Dongarra, "Process Fault-Tolerance: Semantics, Design and Applications for High Performance Computing," *Int'l J. High Performance Applications and Supercomputing,* vol. 19, no. 4, pp. 465-478, 2005.
[44] R. Batchu, Y. Dandass, A. Skjellum, and M. Beddhu, "MPI/FT: A Model-Based Approach to Low-Overhead Fault Tolerant Message-Passing Middleware," *Cluster Computing,* 2004.
[45] A. Avizeinis, "The N-Version Approach to Fault-Tolerance Software," *IEEE Trans. Software Engineering,* vol. 11, no. 12, pp. 1491-1501, Dec. 1985.
[46] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson, "Experimental Evaluation of Time-Redundant Execution for a Brake-by-Wire Application," *Proc. Int'l Conf. Dependable Systems and Networks (DSN),* 2002.
[47] K. Echtle, B. Hinz, and T. Nikolov, "On Hardware Fault Diagnosis by Diverse Software," *Proc. Int'l Conf. Fault-Tolerant Systems and Diagnostics (FTSD),* 1990.
[48] T. Lovric, "Dynamic Double Virtual Duplex Systems: A Cost-Efficient Approach to Fault-Tolerance," *Proc. Int'l Working Conf. Dependable Computing for Critical Applications (DCCA),* 1995.
[49] Z. Kalbarczyk, R.K. Iyer, S. Bagchi, and K. Whisnant, "Chameleon: A Software Infrastructure for Adaptive Fault Tolerance," *IEEE Trans. Parallel and Distributed Systems,* vol. 10, no. 6, pp. 560-579, June 1999.

Alex Shye received the BS degree in computer engineering from the University of Illinois and the MS degree in computer engineering from the University of Colorado. He is currently a PhD candidate in the Electrical Engineering and Computer Science Department, Northwestern University. His research interests include reliability, dynamic optimization, and user-aware computer architectures. He is a student member of the IEEE.

Joseph Blomstedt received the BS degree in computer engineering from the University of Washington and the MS degree in electrical and computer engineering from the University of Colorado. He is currently a PhD candidate in the Department of Computer Science, University of Colorado. His research interests include system reliability, dynamic optimization, and assisted software parallelization. He is a student member of the IEEE.

Tipp Moseley received the BS degree in computer science from Georgia Institute of Technology. He is currently a PhD candidate in the Department of Computer Science, University of Colorado. His research interests include reliability, profiling, and optimization.

Vijay Janapa Reddi received the BS degree in computer engineering from Santa Clara University and the MS degree in computer engineering from the University of Colorado. He is currently a PhD candidate in the Electrical Engineering and Computer Science Department, Harvard University. His research interests include virtual machines for program introspection and optimization. He is a student member of the IEEE.

Daniel A. Connors received the PhD degree in electrical engineering from the University of Illinois, Urbana-Champaign. He is a professor in the Department of Electrical and Computer Engineering, University of Colorado. His research interests are in the areas of architecture, software for high-performance computer systems, dynamic optimization, fault-tolerant computing, and advanced compiler optimization. He is the director of the DRACO Research Group. For his contributions in teaching, he has been nominated for the Eta Kappa Nu C. Holmes MacDonald Outstanding Young Teacher Award for Young Electrical and Computer Engineering Professors. He is a member of the IEEE and the ACM.