Transient Fault Tolerance via Dynamic Process-Level Redundancy

Alex Shye Vijay Janapa Reddi Tipp Moseley Daniel A. Connors
Department of Electrical and
Computer Engineering
University of Colorado at Boulder
{shye, janapare, moseleyt, dconnors}@colorado.edu

Abstract

Transient faults are emerging as a critical concern in the reliability of microprocessors. While hardware reliability techniques are often employed for transient fault tolerance, software techniques represent a more cost-effective and flexible alternative. This paper proposes a software approach to transient fault tolerance which utilizes a run-time system to automatically apply process-level redundancy (PLR). PLR creates a set of redundant processes per application process and compares the processes during run time to guarantee correct execution. Redundancy at the process level allows the operating system to freely schedule the processes across all available hardware resources (i.e. extra threads or cores). PLR is a software-centric approach to transient fault tolerance in which the focus is shifted from ensuring correct hardware execution, to ensuring correct software execution. The software-centric approach is able to ignore many benign faults which do not propagate to affect the program output. In addition, the dynamic deployment creates a very flexible fault tolerant system which transparently applies PLR without prior modifications to the application, shared libraries, or operating system. Experiments using a real PLR prototype on an SMP machine demonstrate that PLR can effectively provide transient fault tolerance with a slowdown of only 1.26x.

1. Introduction

Transient faults, also known as soft errors, are emerging as a critical concern in the reliability of microprocessors. A transient fault occurs when energetic particles (e.g. alpha particles) strike the processor and cause 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 raw error rate of a single transistor will stay relatively constant or decrease slightly [7, 10]. As the number of available transistors per chip continues to grow exponentially, the transient fault rate of future processors can be expected to increase dramatically. Thus, it will be crucial for future systems to employ reliability techniques to ensure correct program execution.

While simple reliability techniques such as ECC and parity can easily be utilized to protect large storage structures such as caches and memory, guaranteeing correct execution through the complex interconnection of gates and logic within modern processor pipelines presents a much more difficult task. The typical solution is to replicate hardware units and check their execution results [9, 21]. However, the design and verification of new redundant hardware is costly and may not be feasible in cost-sensitive markets. In addition, it is unknown how much the inclusion of redundant design elements may impact the design and product cycles of systems. As a result, software approaches have been proposed as a more cost-effective and flexible alternative for transient fault tolerance.

This paper proposes a software approach to transient fault tolerance which utilizes a run-time system to dynamically apply *process-level redundancy* (PLR). PLR is a redundancy technique which creates a set of redundant processes per original application process and compares them to verify execution results. Upon the detection of a transient fault, a majority vote between the redundant processes is taken to verify program results and recover from the fault.

Redundancy at the process level allows the operating system (OS) to freely schedule the redundant processes to all available hardware resources. As microarchitectural trends point towards the design of massively multi-threaded and multi-core architectures, future microprocessors will likely contain extra hardware threads and cores. In computing environments which are not throughput constrained, PLR provides an alternate method of leveraging the hardware parallelism for transient fault tolerance.

PLR implies a *software-centric* paradigm in transient fault tolerance which views the system as software layers which must execute correctly. This differs from the common *hardware-centric* model which views the system as hardware components which must execute correctly. The software-centric fault tolerance model shifts the focus from ensuring correct hardware execution, to ensuring correct software execution. Only the transient faults which affect software output are detected and many benign faults which do not propagate to affect program correctness are ignored.

Dynamically deploying PLR via a user-level run-time system creates an extremely flexible transient fault tolerance framework. A run-time system is a software layer which enables the dynamic introspection and modification

of any program as it executes. As a result, the dynamic PLR implementation is able to provide transient fault tolerance without prior modifications to the application, shared libraries, or operating system.

This paper presents a dynamic PLR prototype which is implemented in a real run-time system [11]. A fault injection campaign is used to determine the result of injected faults, demonstrate the effectiveness of PLR, and show the advantages of using a software-centric fault tolerance model. Performance is evaluated on a variety of hardware platforms to show how PLR scales to differing architectures. On a 4-way SMP machine, the prototype is able to effectively avoid benign faults, and detect true faults with only a 1.26x slowdown.

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

2.1. Transient Fault Preliminaries

In general, a transient fault can be classified by its effect on program execution into the following categories [25]:

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 [24].

Silent Data Corruption (SDC): A transient fault which is undetected and propagates to corrupt program output is considered a 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 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. Without recovery, a false DUE will cause the system to unnecessarily halt execution and with recovery, will cause unwarranted calls to the recovery mechanism.

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, but may inadvertently identify some of the benign faults into false DUEs. Finally, a system with both detection and recovery will detect and recover from all faults without SDCs or any form of DUE.

2.2. Transient Fault Detection

Transient fault tolerance consists of *transient fault detection* and *transient fault recovery*. The detection mechanism continuously monitors program execution to verify correct execution and dominates the performance overhead for transient fault tolerance. The recovery mechanism is invoked each time the detection mechanism detects a fault. As such, a large majority of transient fault tolerance work focuses primarily on the detection mechanism.

Hardware fault tolerance techniques have been extensively explored using replicated hardware units [9, 21], specific checking structures [1, 25], and extensions to superscalar [16], SMT [19, 12, 17, 22, 23] and CMP [6, 12] processors. These hardware approaches can be effective but suffer from a few limitations. First, the approaches incur tremendous costs in the design and verification of dedicated hardware for fault detection. In cost-sensitive markets, the hardware approach may not be viable. Hardware approaches are also not very flexible; once the hardware is created, it is not easily modified or extended. Furthermore, a hardware techniques is often specific to a particular processor model.

Software solutions offer a cheaper and more flexible alternative to the hardware approaches. Current software techniques involve using the compiler to insert fault tolerant instructions [14, 13, 18]. A drawback to such mechanisms is that the execution of the inserted instructions and assertions decrease performance ($\sim$1.4x slowdown [18]). Also, a compiler approach requires recompilation of all code which is to be protected. Not only is it inconvenient to recompile all applications and libraries, but the source code for commercial or legacy programs is often unavailable.

2.3. 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. Checkpoint and repair can occur at different granularities [16, 23]. Fault masking, such as triple modular redundancy (TMR) [27], involves using at least three copies of execution to determine the correct execution results. Fault masking often requires more hardware resources than checkpoint and repair techniques, but has the advantages of simplicity, and the ability to constantly make forward progress during execution.

3. Approach

3.1. Software-centric Fault Detection

The *sphere of replication* (SoR) [17] 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

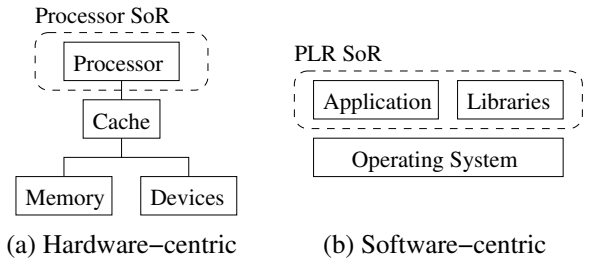

Figure 1. Hardware-centric and software-centric transient fault detection models.

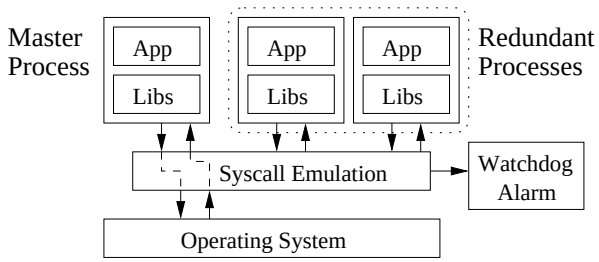

Figure 2. Overview of PLR with three redundant processes.

replicated and all execution within the SoR is 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.

Many of the previous transient fault detection approaches mentioned in Section 2.2 are *hardware-centric*. As shown in Figure 1(a), the hardware-centric model views the system as a collection of hardware components which must be protected from transient faults. The SoR is commonly placed around the processor datapath assuming the cache and memory to be parity or ECC protected. Thus, all loads are replicated, all execution is redundant, and all stores are compared for output correctness. The compiler-based software techniques are also hardware-centric. For example, SWIFT [18] places its SoR around the processor. However, without the ability to duplicate 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.

Software-centric fault detection is a paradigm in which the system is viewed as the software layers which must execute correctly. 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. The main difference is that a software-centric SoR encompasses software execution instead of hardware units, and shifts the focus from ensuring correct hardware execution to ensuring correct software execution. As a result, the software-centric model only detects faults which propagate to affect software execution, and then result in an error in program output. 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 invoke the recovery mechanism for falsely detected faults which do not affect correctness.

3.2. Process-Level Redundancy

Process-level redundancy (PLR) is a redundancy 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 simply allows the OS to schedule the redundant processes to take advantage of hardware extra threads and cores. With massively multi-threaded and multi-core architectures on the horizon, there will be a tremendous amount of course-grained 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. At the beginning of execution, the original process is replicated to create the redundant processes. One of the processes is labeled the *master* process and the others are labeled the *redundant* processes. At each system call, the *system call emulation unit* is invoked. The emulation unit performs the input replication, output comparison, and recovery. The emulation unit also unit 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. Therefore, all system calls which alter any system state can only be executed once. In general, the master process is allowed to actually execute

the system call while the redundant processes emulate the system call.

- Execution among the redundant processes must be deterministic. System calls which return non-deterministic data such as a `gettimeofday()` must be emulated to ensure all redundant processes use the same data for computation. In this paper, we present a single-threaded model of PLR. Ensuring determinism for multi-threaded applications is a much more difficult task and is reserved for future work.
- 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 strike and 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 (i.e. flip a comparison value and cause an infinite loop). A *watchdog alarm* is employed by the emulation unit to detect such faults and is described further in Section 3.3.

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 is received once by the master process, and then replicated among the redundant processes. Examples include the data returned from the`read()` and `gettimeofday()` system calls. Also, the return value from all system calls is considered an input value and is copied for use across all redundant processes. It is also important for PLR correctness reasons that input data be received once and replicated. For example, if all redundant processes are allowed to call `gettimeofday()`, the processes would all receive different timestamps and execute with different data resulting in non-deterministic behavior among the 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. Data may exit the SoR for a couple of reasons. Any write buffers which will be passed outside of the SoR must be compared. Examples include write buffer in the `write()` system call and the syncing of a memory-mapped file via the `msync()` system call. 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

Depending upon the system call, the system call emulation unit will perform different tasks. System calls which modify any system state, such as the file system, must only be executed once. Examples include the `rename()` or `unlink()` system calls. In some cases, a system call will be used to modify state at the process level. When this occurs, the emulation of the system call may include actually calling the same system call. In these cases, the system call ends up being called by all redundant processes. For example, the `open()` system call is first executed by the master process to open a file (and create it if necessary). The redundant processes will modify the system call flags remove the file create flag, and then call `open()` themselves to maintain similar file descriptor tables. The `lseek()` call is another example where all redundant processes must seek within a file the proper distance to maintain consistency between the file offset pointers of each redundant process.

3.3. Transient Fault Detection

In PLR, 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 it is about to exit the SoR.
2. **Watchdog Timeout:** A transient fault may cause a process to hang indefinitely (e.g. an infinite loop). A *watchdog alarm* is set at the beginning of each call to the emulation unit. If the watchdog alarm times out, a fault assumed. A drawback to the watchdog alarm is that a timeout period must exist in which the application does not make any progress.
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. While these cases could be detected with the watchdog alarm, this would cause unnecessary overhead in waiting for the watchdog alarm to time out. Signals handlers are set up to catch the corresponding signals, such as SIGSEGV, and an error is flagged. The next time the emulation unit is called, it immediately begins the recovery process.

3.4. Transient Fault Recovery

PLR uses fault masking with a majority vote for transient fault recovery. An odd number of redundant processes is necessary with a minimum of three processes. The recovery mechanism acts differently based on how a fault is detected. The recovery process is listed below corresponding to the detection cases described in Section 3.3.

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. Any processes with incorrect data are immediately killed and replaced by using a correct process to call the `fork()` system call.

2. **Watchdog Timeout:** A watchdog timeout does not have any data to compare. Instead, it is only important to determine the incorrect process which is the process that has not yet entered the emulation unit. The incorrect process is killed and replace by forking one of the remaining processes.

3. **Program Failure:** In the case of program failure, the incorrect process is already dead. The emulation unit simply replaces the missing process by creating another redundant process via the `fork()` system call.

Most transient fault techniques are designed for the single event upset (SEU) model where only a single transient fault occurs at a time. However, it is possible that multiple simultaneous errors occur. PLR is able to sustain simultaneous faults by simply scaling the number of redundant processes. For example, five redundant processes can be used to detect and recover from two simultaneous faults to different processes. Increasing the number of processes only requires a few extra forks, and scalalable majority vote logic which is trivial to implement in software.

3.5. Run-time PLR Deployment

This paper suggests a dynamic approach to implementing PLR using a run-time system. A run-time system is a user-level software layer which enables the dynamic modification and introspection of any program as it is executing [5, 11]. While the concept of PLR could be implemented with a static compiler, a dynamic approach provides a higher degree of convenience and flexibility. Users need not recompile the application and its associated shared libraries. Hardware and software developers do not need to design with transient fault tolerance in mind. The user who desires reliable execution simply acquires a run-time software package. In addition, the run-time system is easy to turn on and off. For example, some applications, such as MP3 audio decoding in which errors cause minor temporary glitches, do not require the same protection as others, such as financial software.

4. Experimental Results

4.1. Methodology

This paper presents and evaluates a PLR prototype built using the Intel Pin system [11]. The tool uses Pin to dynamically fork the redundant processes and uses PinProbes to intercept and emulate system calls. The prototype is evaluated running a set of the *SPEC2000* benchmarks. Fault coverage is evaluated using a fault injection campaign similar to [18]. One thousand runs are executed per benchmark. For each run, an instruction execution count profile is used to randomly choose a specific invocation of an instruction to fault. Within the instruction, a random bit is picked from the source and destination registers. During run time, Pin is used to flip the random bit during the specified dynamic execution count of the instruction. The *specdiff* utility within the *SPEC2000* harness is used to determine the correctness of program output. The test inputs are used in the interests of keeping experimental run times manageable.

Performance is measured on the reference inputs by running PLR with three redundant processes, without fault injection, on a four processor machine. Each processor supports SMT. Execution is constrained to specific sets of hardware contexts, using `sched_set_affinity()`, to simulate different hardware platforms.

4.2. Fault Injection Results

Figure 3 shows the results of a fault injection campaign split into the various outcome categories. The left bar in each cluster shows the results with fault injection on native runs of the benchmarks. The right bar shows the results using PLR to detect the injected faults. Without fault detection there are three possible outcomes of an injected fault. First, the injected fault may not affect the program execution results (*Correct*). Second, execution may continue and complete but result in incorrect output (*Incorrect*). Third, execution could fail to complete (*Fail*). By applying PLR, there are two additional outcomes. PLR may detect a fault through a mismatch of data on a write event (*Mismatch*), or through a signal handler catching an illegal function (*SegFault*) as described in Section 3.3. The detection case with the watchdog alarm timeout is ignored because it occurs very infrequently ($\sim$.05% of the time).

PLR is able to successfully eliminate all of the *Failed* and *Incorrect* outcomes. In general, the write comparisons detect the *Incorrect* cases and turn them into detected *Mismatch* cases and the *Failed* cases are detected by the watchdog alarm and turned into *SegFault* 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 result in incorrect execution are detected through either a mismatch of write data or the timeout mechanism. Faults which do not affect correctness are not detected in PLR thereby avoiding false positives. In contrast, SWIFT [18], 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 occurs with the *SPEC fp* 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 actually 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

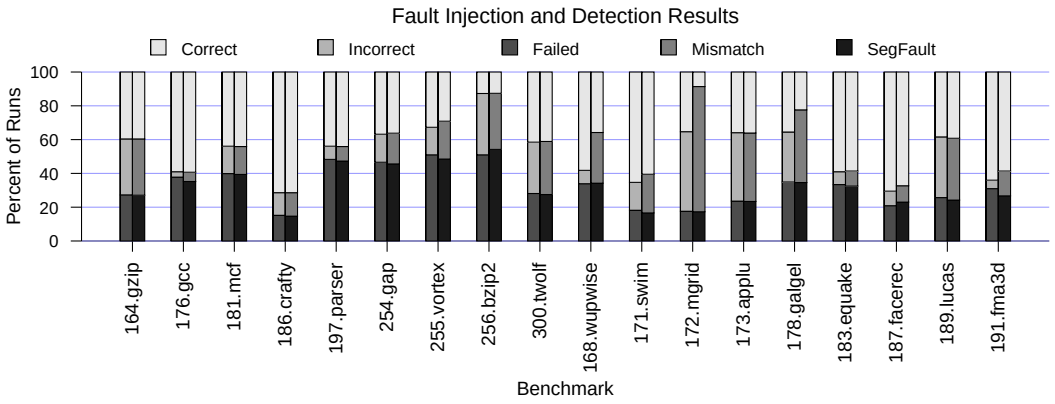

Figure 3. Results of the fault injection campaign. The left bar in each cluster shows the outcomes with just fault injection and the right bar shows the breakdown of how PLR detects the faults.

Name	Description
PLR-1x1	Single processor
PLR-2x2	Two processors, each 2-way SMT
PLR-4x1	Four single processors

Table 1. Hardware platforms for PLR performance experiments.

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.

4.3. Performance Results

Table 4.3 shows the different platforms used for evaluating the performance of the dynamic PLR prototype. The platforms include a single processor, a dual SMT processor, a 4-way SMP processor. Figure 4 shows the execution times for each of the performance runs (using three redundant processes) normalized to native execution time. The runs on the single processor incur a 3.5x slowdown which is reasonable as the application is executing three times with barriers and inter-process communication (IPC). On the dual SMT processor configuration, one of the SMT processors is allocated a single redundant process, while the other is allocated two processes. In this case, the processor with two processes becomes the bottleneck, but the performance is improved to a 2.4x slowdown. Performance improves dramatically when moving to the 4 processor SMP machine with only a slowdown of 1.26x; which is an 36% improvement over the fastest previous compiler-based fault detection technique (1.4x slowdown [18]).

The 4-way SMP configuration still requires IPC to utilize a on-board processor interconnect. A CMP with at least three cores would provide an improved IPC latency and would be the best case scenario for PLR. To provide an idea of the potential benefits, a study is performed to break down the PLR overhead. During execution, overhead can be attributed to simply running three copies of the same application which increases contention for shared resources within the system (e.g. the system bus). This is defined as *contention overhead*. The remaining *PLR overhead* is due to the execution of PLR and the interprocess communication. Figure 5 shows a breakdown of the contention overhead and PLR overhead for the 4-way SMP configuration. On average about 17% of the slowdown can be attributed to the contention overhead and about 8% of the overhead is due to PLR and IPC. Moving to a CMP architecture would enable the 8% PLR overhead to be greatly reduced.

5. Related Work

PLR similar to a software version of the hardware SMT and CMP extensions [6, 12, 17, 19, 22, 23]. To the best of our knowledge, it is the first software technique for transient fault detection which is able to leverage multiple hardware threads or cores.

Executable assertions [8] and other software detectors [15] explore the placement of assertions within software similar to our software-centric model. The software-centric model presents a more formalized method of software checking based on the SoR model. The pi bit [25] and dependence-based checking [23] follow the propagation of faults in an attempt to only detect faults which affect program behavior. By using the software-centric model, and using a broad SoR around the user space, PLR easily accomplishes the same task on a larger scale.

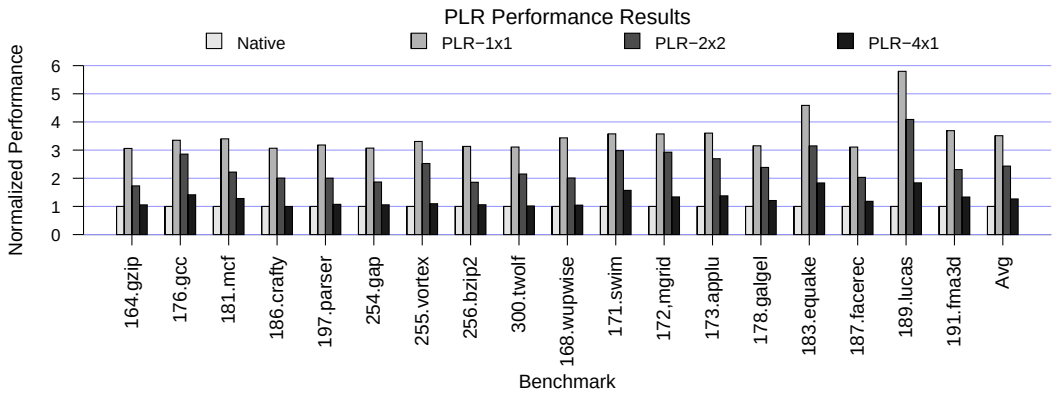

Figure 4. Normalized performance results when running on the platforms shown in Table 1.

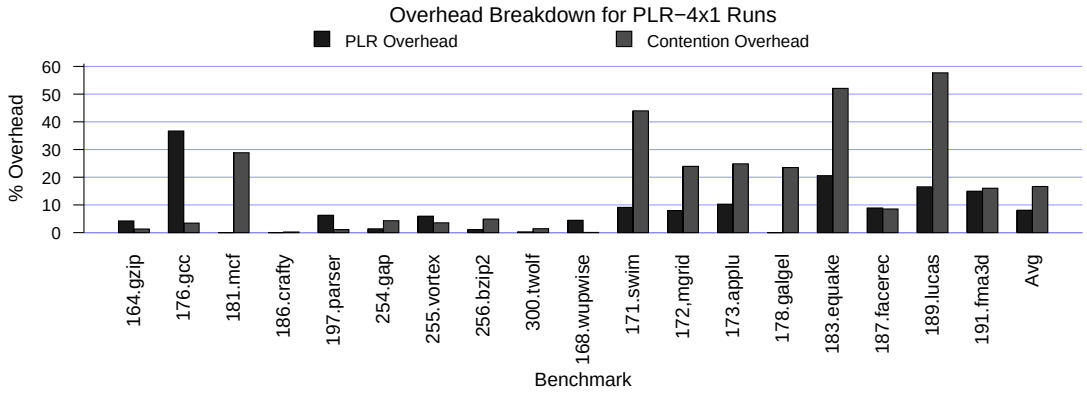

Figure 5. Overhead breakdown of running PLR on the a four processor SMP machine.

The PLR approach is similar to a body of fault tolerant work which explores the use of *replicas* (similar to our redundant processes) within the operating system [2], a hypervisor [4], post-link system [3], and for specialized hardware [26, 27]. This body of work targets hard faults while PLR targets transient faults, which have different issues. For example, output mismatches are not checked in detecting hard faults. The replica work assumes specialized hardware (such as failstop processors [20]) or a modified operating system [2]. In contrast, PLR is designed for general purpose processors running on a generic, unmodified operating system. We do not claim to invent the idea of redundant processes. Instead, we leverage the idea for transient fault tolerance using emerging technologies including run-time systems as well as SMT, CMP and multi-processor architectures.

6. Conclusion

This paper presents a dynamic software transient fault tolerance technique which utilizes process-level redundancy (PLR). PLR is a software-centric model of fault detection which effectively ignores benign faults and only detects faults which propagate to effect program output. By providing redundancy at the process level, PLR scales well in multiprocessing environments. To the best of our knowledge, this is the first software transient fault tolerance technique which can leverage multiple hardware threads or cores. The performance of a prototype is evaluated on a variety of hardware platforms and achieves a slowdown of only 1.26x on a 4-processor SMP, a 36% improvement over the fastest previous software transient fault detection technique. Future work involves exploring policies for dynamically adapting the level of redundancy, as well as discovering methods of ensuring determinism in more complex applications which involve shared memory, signals, interrupts, and threading.

7. Acknowledgments

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

References

[1] T. M. Austin. DIVA: a reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 196–207. IEEE Computer Society, 1999.

[2] 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.

[3] T. C. Bressoud. TFT: A Software System fo Application-Transparent Fault-Tolerance. In *Proceedings of the International Conference on Fault-Tolerant Computing*, pages 128–137, June 1998.

[4] T. C. Bressoud and F. B. Schneider. Hypervisor-based Fault-tolerance. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 1–11, December 1995.

[5] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 265–275. IEEE Computer Society, 2003.

[6] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 98–109. ACM Press, 2003.

[7] 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.

[8] M. Hiller, A. Jhumka, and N. Suri. On the placement of software mechanisms for detection of data errors. In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2002.

[9] R. W. Horst, R. L. Harris, and R. L. Jardine. Multiple instruction issue in the NonStop Cyclone processor. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 216–226, May 1990.

[10] 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.

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

[12] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the 29th annual international symposium on Computer architecture*, pages 99–110. IEEE Computer Society, 2002.

[13] N. Oh, P. P. Shirvani, and E. J. McCluskey. Control-flow checking by software signatures. In *IEEE Transactions on Reliability*, volume 51, pages 111–122, March 2002.

[14] N. Oh, P. P. Shirvani, and E. J. McCluskey. Error detection by duplicated instructions in super-scalar processors. In *IEEE Transactions on Reliability*, volume 51, pages 63–75, March 2002.

[15] 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.

[16] J. Ray, J. C. Hoe, and B. Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 214–224. IEEE Computer Society, 2001.

[17] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 25–36. ACM Press, 2000.

[18] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In *Proceedings of the 3rd International Symposium on Code Generation and Optimization*, March 2005.

[19] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, page 84. IEEE Computer Society, 1999.

[20] 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.

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

[22] K. Sundaramoorthy, Z. Purser, and E. Rotenburg. Slipstream processors: improving both performance and fault tolerance. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 257–268. ACM Press, 2000.

[23] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery using simultaneous multithreading. In *Proceedings of the 29th annual international symposium on Computer architecture*, pages 87–98. IEEE Computer Society, 2002.

[24] N. Wang, M. Fertig, and S. J. Patel. Y-branches: When you come to a fork in the road, take it. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 56–67, September 2003.

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

[26] J. H. Wensley, L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Melliar-Smith, R. E. Shostak, and C. B. Weinstock. SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control. *Proceedings of the IEEE*, 66(10):1240–1255, October 1978.

[27] 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.