Software Fault Detection Using Dynamic Instrumentation

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

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

Abstract

In recent decades, microprocessor performance has been increasing exponentially. A large fraction of this performance gain is directly due to smaller and faster transistors enabled by improved fabrication technology. While such transistors yield performance enhancements, their lower threshold voltages and tighter noise margins make them less reliable [1], rendering processors that use them more susceptible to *transient faults*. Transient faults, also known as *soft errors*, are intermittent faults caused by external events, such as energetic particles striking the chip, that do not cause permanent damage but may result in incorrect program execution by altering signal transfers or stored values.

To detect or recover from these faults, designers have typically introduced redundant hardware. For example, storage structures such as caches and memory often include error correcting codes (ECC) or parity bits while techniques like lockstepping or Redundant Multithreading [5] have been proposed for full processor protection. Although these techniques are able to increase reliability, they all require changes to the hardware design.

Software-only approaches to reliability have been proposed and evaluated as alternatives to hardware modification [3, 4, 6]. These techniques have shown that they can significantly improve reliability with reasonable performance overhead.

Software-only techniques do not require any hardware support and thus are far cheaper and easier to deploy. In fact, these techniques can be used for systems that have already been manufactured and now require higher reliability than the hardware can offer. This need can occur because of poor estimates of the soft error problem or changes in the environment, such as moving to higher altitudes.

Software-only approaches also benefit from reconfigurability after deployment. Since reliability is achieved via software, the system can dynamically configure the tradeoff between reliability and performance. Software techniques can be configured to only add reliability in certain environments, for specific applications, or even for critical regions of an application, thus maximizing the reliability while minimizing the costs.

Although software-only error mitigation techniques do exist, all previous proposals have been static compilation techniques that relied on source code transformations or alterations to the compilation process. Our proposal is the first application of software fault detection for transient errors that increases reliability dynamically. Our proposal uses a modified version of the PIN dynamic instrumentation framework [2] to enact the reliability transformations.

Using dynamic instrumentation in this way to increase reliability, rather than static compilation, is advantagous for a number of reasons. The application of our technique is trivial since the only requirement is the program binary. It does not require a recompilation, which is necessary in situations with legacy applications that no longer have readily available or easily re-compilable source code. Even if the application source is available, users typically do not recompile libraries (such as `glibc`) when recompiling an application.

It is possible to create a binary to binary translation that enhances reliability, but our dynamic reliability technique can seamlessly handle variable-length instructions, mixed code and data, statically unknown indirect jump targets, dynamically generated code, and dynamically loaded libraries. Our technique is also able to modify executing programs. It can attach to a running application and increase its reliability, and detach when appropriate, thus returning to faster (and unreliable) execution.

We based our fault detection implementation on the SWIFT software-only reliability technique [6]. The SWIFT technique is composed of two mainly orthogonal parts, instruction duplication with detection and control flow protection. In this work, we focus on the duplication and detection and do not implement the control flow protection mechanism. Our technique dynamically duplicates all instructions, except for those instructions that write to memory. Since a fault will only manifest itself as a program error if it changes the output, it is desirable to delay validation until an instruction that may affect output. In our mechanism, we assume that data going to memory can affect output, so we delay validation until store instructions. This delay reduces the number of false errors detected. Although a fault may have changed a data value, if that value is dynamically dead or if the faulted bits are masked away, then our technique will not halt the execution because that fault is irrelevant.

There are certain instructions that we cannot duplicate and these instructions must be handled differently. Our technique does not duplicate load instruction, but rather creates a copy of the loaded value into a redundant register. This copy needs to occur because two loads from the same address may not always produce the same value. For example, in a multi-process environment, another process may change the value stored in memory between the original and redundant loads. This would cause the reliable program to signal a detected error although no fault has occurred. Since out technique targets the x86 instruction set,

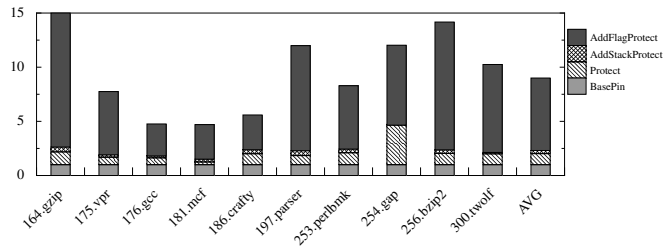

Figure 1. Performance for duplication only, accounting for specialized registers.

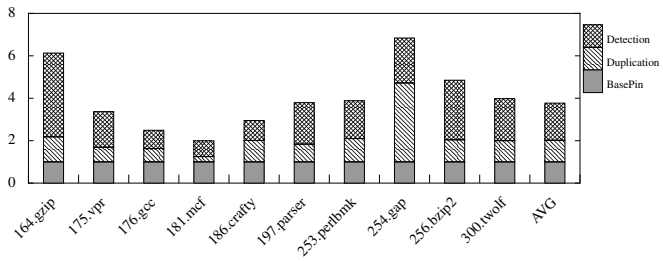

Figure 2. Performance of detection compared to duplication.

certain other instructions also require special handling of this type. The RDTSC instruction, which reads the hardware time-stamp counter, must have its destination value copies because two invocations of this instruction will always result in two distinct values.

Our reliability enhancements were implemented in the PIN dynamic instrumentation framework [2]. Redundant instructions, as well as validation and copying instructions, are inserted during the dynamic instrumentation. We use the existing PIN framework to register allocate the additional code, as well as perform other basic optimizations like data liveness analysis. Due to current limitations with register allocation in the PIN tool, we do not duplicate floating point or multimedia instructions, but this is part of our future work to increase reliability.

To analyze the performance of our reliability approach, we first calculated the cost of duplicating instructions without data verification. We compared the performance executions relative to a base PIN execution with no reliability and no instrumentation tools. We ran all SPECINT2000 executions on native hardware using reference inputs.

We found that inserting redundancy was dominated by the duplication of the EFLAGS register. Figure 1 shows the normalized execution times when duplicating instruction without duplicating the stack pointer and EFLAGS registers, as well as performance when duplicating those registers. The average normalized executing time without the EFLAGS register is 2.31x slower that the base, but when protecting that one register, the time increases to 9.00x. Duplicating the EFLAGS register is extremely expensive due to the restricted manner in which it may be fully accessed. It can only be completely moved to and from the memory stack, whereas non-EFLAGS registers can be moved into other architectural register. In addition, moving the entire EFLAGS register is a very expensive operation.

Duplication of the stack pointer was the second most expensive register, bringing the normalized execution time from 2.02x to 2.31x. This degradation was mainly due to instructions that implicitly read or write to the stack pointer. Since this required the ESP architectural register, the allocation of the two virtual stack pointers was limited.

Instruction duplication adds the redundancy necessary for independent computation, but comparison of the independent versions is necessary for fault detection. Figure 2 shows the normalized performance for full detection, attributing the performance costs for duplication and verification. These performance execution use only a single version of the stack pointer and EFLAGS register.

On average, the normalized execution time for instruction duplication alone is 2.02x while duplication plus data verification is 3.77x. The per benchmark degradations vary, ranging from 254.gap with a cost of 6.84x to 181.mcf with a cost only 1.99x. Benchmarks like 181.mcf which contain many cache misses have extra instruction level parallelism to execute the redundant and detection instruction without affecting the critical path.

Our technique shows that a dynamic software-only approach to reliability is possible with acceptable performance degradation. Our future work involves targeting ways to further increase performance of the reliable execution though smarter register allocation and scheduling. We also plan to simulate fault injections to determine the precise fault coverage which will guide the dynamic trade-off between reliability and performance.

References

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

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

[3] 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.

[4] M. Rebaudengo, M. S. Reorda, M. Violante, and M. Torchiano. A source-to-source compiler for generating dependable software. pages 33–42, 2001.

[5] 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.

[6] 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.