

# Limitations of Software Solutions for Soft Error Detection

Adam Brown , Calvin Lin

**Abstract**—This paper discusses limitations of current software solutions to detecting soft errors. We identify three classes of shortcomings and argue that one class is conceptually easy to overcome, one is fundamentally impossible to overcome without added hardware support, and one can be overcome by a new solution that we propose.

## I. INTRODUCTION

Trends toward smaller feature sizes and lower operating voltages are producing increased soft error rates in microprocessors [16]. A wide range of solutions have been proposed, including software solutions in which the compiler performs code transformations to detect such errors [12], [10], [7], [8], [13], [6], [7], [8], [11], [12]. These software solutions are attractive because they do not require changes to the hardware and they theoretically can provide flexible degrees of fault resilience as appropriate for a given application and a given operating environment.

The stated goal of such solutions is to detect transient faults caused by arbitrary single-bit errors in logic. The basic solution is simple: for instructions that do not alter control flow, replicate the instruction—and its operand—and insert code that checks whether the replicated computation matches the original; for instructions that do alter control flow, modify the program so that it declares its intended control flow before the instruction executes and then verifies that the actual control flow matches the expected control flow after the instruction executes.

Unfortunately, to achieve such simplicity, previous work appears to have been unaware of or disregarded various limitations of this approach. Some of these limitations are quite expensive to solve, but the existence of others is surprising because they are simple to correct. Of course, no system can detect all errors, and it's easy to claim that because reliability is a game of percentages, not all errors need to be caught. But we believe that it is important to have a clear understanding of an approach's weaknesses as a guide to future work. Thus, this paper identifies

three classes of limitations facing current software-based solutions.

The first type of limitation involves dynamically discovered code, such as self-modifying code or dynamically loaded libraries. Detecting errors in such code is not conceptually difficult, but they do incur non-trivial engineering costs which should be acknowledged, particularly with the growing use of languages such as Java and C#.

The second type of limitation stems from an implicit assumption about a program's control-flow behavior. We dub these shortcomings *CFG-based limitations*, short for Control-Flow Graph-based limitations. We describe several manifestations of this limitation, and we present a solution that addresses all but one of these limitations, albeit at the cost of added overhead. We also argue that this one specific limitation is infeasible to detect in software, so hardware support is needed.

Finally, the third type of limitation comes from non-architected state that is not accessible to the software and thus fundamentally not detectable by software techniques. We give examples of how such limitations can allow single-bit errors that manifest themselves in wide-reaching manners.

## II. RELATED WORK AND BACKGROUND

Several software solutions have been proposed in the literature. In general, these solutions divide the problem of soft error detection into two subproblems: (1) detecting errors in non-control flow instructions, and (2) detecting errors in control flow instructions.

To detect errors in the execution of non-control flow instructions, the instruction is executed redundantly and the output is compared against the output of the original computation. If the output values disagree, then the system has detected an error. Since most of the software solutions assume that values in registers can be corrupted by soft errors, the original and replicated instructions use both different input registers and different output registers.

A different technique is needed for control-flow instructions. The basic idea is to declare before each change in

Adam Brown is at the University of Texas at Austin, email: [abrown@cs.utexas.edu](mailto:abrown@cs.utexas.edu)

Calvin Lin is at the University of Texas at Austin, email: [lin@cs.utexas.edu](mailto:lin@cs.utexas.edu)

control-flow the expected target destination and to then check after the change in control-flow whether the actual destination was correct. To implement such schemes, control-flow is conceptually represented as a *control-flow graph*, which is a directed graph whose nodes are *basic blocks* and whose edges are potential control flow between basic blocks. Within each basic block, control flow is assumed to be sequential, so only basic block entrances and exits need to be checked. In particular, each basic block is given a unique static identifier, and each basic block has code inserted that writes to some well-defined memory location to record the intended target of the ensuing branch. Each basic block also has code inserted so that upon each entrance to the basic block, a check is performed to verify that the current basic block is the same as the recorded target.

### III. LIMITATIONS

We now discuss the three classes of limitations.

#### A. Dynamically Discovered Code

The first class of limitations arises from dynamically discovered code, whose existence is not acknowledged by previous solutions. Such code is conceptually identical to statically identified code, but the code must be translated before it is executed.

#### B. CFG-Based Assumptions

The second class of limitations occur because previous software solutions have implicitly assumed that faulty programs obey certain aspects of the control-flow graph, namely, (1) that all basic blocks have single-entry single-exit behavior and (2) that all branches are statically identified. In practice, there are at least six ways that a single bit flip can violate these two assumptions.

First, if a target of a branch is corrupted, assumption (1) can be violated so that any instruction becomes the target of a branch. A corrupted branch could thus jump directly past the control flow checking code, avoiding detection. In Figure 1, we see an example where a single bit-flip may cause the control-flow checking at a block's entry to be avoided. For schemes [11] whose control-flow checking does not verify the basic block being exited, this error would go undetected. Any control-flow checking scheme that relies on XORing two statically-determined values [7], [13] will also suffer from this problem when the basic block has a back-edge to itself.

Second, it is possible that a control-flow instruction may erroneously return execution to the same basic block that was supposedly being exited, resulting in

```

...
bne L1
...
L1: ... % Control-flow checking code
... % intervening code
L1': add r1, r2, r3
...

```

Fig. 1. If the Hamming distance for labels L1 and L1' is 1, then a single-bit soft error may cause the `bne L1` instruction to branch erroneously to L1', by-passing the control flow checks.

```

L0: ... % Control-flow checking code
...
L1': add r1, r2, r3
...
br L1
...
L1: ... % Control-flow checking code

```

Fig. 2. If the Hamming distance for labels L1 and L1' is 1, then a single-bit soft error may cause the `br L1` instruction to return control back to the same basic block containing that branch instruction.

what we refer to as an *intra-block branch*. Intra-block branches represent edges that arise dynamically, but previously proposed solutions only insert checks for statically known edges, so such edges can escape control flow checking.

Figure 2 demonstrates the problem of intra-block branches. For control-flow checking schemes that defer control-flow checking to the target basic block [11], [7], [12] this error will go undetected. The later, correct control-flow transfer to block L1 from block L0 will appear to be correct, despite an intervening branch back to L1'.

Third, a single bit flip can cause a non-branching instruction to become a branch instruction if the Hamming distance between the two instructions is 1, thus violating the assumption that all branches have been statically identified. We dub these instructions *potential branches*. For example in the Alpha ISA, the XOR opcode differs by a single bit from the FBEQ (floating branch equal) opcode. Similarly, in the PowerPC ISA, the XORI opcode differs by one bit from the Bx (unconditional branch) opcode. Because the potential branches are not checked, their erroneous control flow will also go undetected.

The fourth and most significant problem also involves dynamically discovered branches. Like any other register in the microprocessor, the program counter (PC) is susceptible to soft errors. Unlike soft errors in other registers, which only affect data values, a soft error in the PC affects control flow. Logically, when errors in the PC are taken into consideration, *every* instruction in the program becomes a potential branch. Unfortunately, we cannot detect errors in the PC in the same manner as

other potential branches, because each instruction that we add is itself a potential branch, so inserting detection code only adds more code to check, with no way to end the recursion.

A fifth problem is that previous software solutions associate static identifiers with each basic block, again assuming that the control-flow graph is statically known. However, this assumption precludes the use of *computed gotos*, in which the target of the control-flow instruction is computed at runtime. This limitation prevents compilers from generating jump tables when compiling complex *switch*-like statements, resulting in larger, less optimized code. Additionally, modern object-oriented languages rely heavily on computed gotos to implement dynamic binding for method calls.

Finally, previous solutions assume that procedure returns always correctly transfer control flow to the calling procedure. To check these return values, a solution must be able to handle computed gotos.

### C. Non-architected State

The third class of limitations is fundamental: Any non-architected state that affects program correctness cannot be detected by a software solution. Consider a soft error in the directory lookup for the instruction cache. If a lookup of an instruction cache line experiences an error and fetches the wrong cache line from the cache, the error will go unnoticed unless the incorrect cache line modifies the value of a register whose data is checked. Thus, program can repeatedly access the wrong cache line until it is replaced, and this single soft error can manifest itself as a long term error. Soft errors in address translation can similarly present themselves as multiple errors.

## IV. STACK-BASED CONTROL-FLOW CHECKING

We now present a solution that addresses the CFG-based limitations. This technique does not detect soft errors that corrupt the PC.

As with schemes like to CFCSS [7], we associate a unique identifier with each basic block. As opposed to using a statically determined block identifier, we use the basic block's memory location as the identifier. In this way, we avoid the limitation that CFCSS-like solutions have with computed gotos.

Unlike previous solutions, our solution checks both basic block entrances and basic block exits. Furthermore, unlike previous solutions, which only remember the last executed change in control flow, our solution uses a stack to verify the invariant that the number of checks

performed is exactly twice the number of control-flow transfers.

In particular, before entering a basic block, we push the identifier for that block onto the stack *twice*. At the beginning of the block, we pop from the stack and verify that the popped value matches that block's identifier. At the end of the block, we pop from the stack and again verify that the value matches that block's identifier. If the first check fails, we know that we have entered the incorrect basic block. If the second check fails, we know that we are exiting from a different block than the one we most recently entered. In this way, a corrupted branch will be detected.

Since we pop the identifier for the current basic block and then push the identifier for the next basic block before a branch, an intra-block branch will be detected when we attempt to exit the block the next time, because the identifier at the top of the stack will not match the exiting basic block but instead will match the identifier of one of the successors to this basic block.

Procedure calls are handled in the same manner as basic blocks. The combination of procedures and basic blocks identifiers that are pushed and popped from the stack provides a way to detect a branch that skips the checking code. If the pop at a basic block's beginning is skipped, it will be detected at the procedure's exit; the last check in the procedure will obtain the identifier for the basic block whose check was skipped instead of the procedure's identifier.

Finally, to address the problem of potential branches, we treat each potential branch as if it were an actual branch. Every instruction is an implicit branch to the next instruction, but these potential branches may also branch to an arbitrary location. Thus, each potential branch instruction breaks its basic block into two basic blocks: one with the instructions before and including the potential branch and another containing all of the instructions following the potential branch. The two basic blocks are treated just as all other basic blocks, and the stack-based control flow checks are added to the two new basic blocks.

As a side note, although we describe this solution in terms of an independent stack, it is possible to use space in the runtime stack by changing the calling convention to reserve space in the stack frame for the identifiers.

## V. CONCLUSIONS

Soft errors will soon be a first-class concern for the reliability of computer systems. Several software solutions have been proposed to detect soft errors, but the solutions share some common limitations. We have

presented a stack-based solution that overcomes many of these limitations, but we explain how fundamental problems exist that cannot be solved by any software solution. Our goals in discussing these problems are (1) to encourage discussion about minor hardware changes that would make software solutions feasible and (2) to encourage future work to consider these problems when evaluating soft error detection schemes.

### Acknowledgments

The authors would like to thank Chuck Moore for his valuable insights and contributions to this work. This research was funded in part by NSF grant ACI-0313263, DARPA Contract #F30602-97-1-0150, and an IBM Faculty Partnership Award.

### REFERENCES

- [1] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault tolerance," *ACM Trans. Comput. Syst.*, vol. 14, no. 1, pp. 80–107, 1996.
- [2] A. Mahmood and E. McCluskey, "Concurrent error detection using watchdog processors—a survey," *Computers, IEEE Transactions on*, vol. 37, no. 2, pp. 160–174, Feb 1988.
- [3] A. Messer, G. Fu, D. Chen, Z. Dimitrijevic, D. D. Mannaru, A. Riska, and D. Milojicic, "Susceptibility of modern systems and software to soft errors," 2001.
- [4] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed design and evaluation of redundant multithreading alternatives," *SIGARCH Comput. Archit. News*, vol. 30, no. 2, pp. 99–110, 2002.
- [5] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, 2003, pp. 29–40.
- [6] N. Oh, P. P. Shirvani, , and E. J. McCluskey, "Ed4i: Error detection by diverse data and duplicated instructions," *IEEE Transactions on Computers*, vol. 51, pp. 180–199, 2002.
- [7] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," *Reliability, IEEE Transactions on*, vol. 51, no. 2, pp. 111–122, March 2002.
- [8] —, "Error detection by duplicated instructions in super-scalarprocessors," *Reliability, IEEE Transactions on*, vol. 51, p. 1, 2002.
- [9] J. Ohlsson and M. Rimen, "Implicit signature checking," in *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, June 1995, pp. 218–227.
- [10] M. Rebaudengo, M. Reorda, M. Torchiano, and M. Violante, "Soft-error detection through software fault-tolerance techniques," 1999. [Online]. Available: 394898
- [11] M. Rebaudengo, M. S. Reorda, M. Torchiano, and M. Violante, "A source-to-source compiler for generating dependable software," in *IEEE Int.l Workshop on Source Code Analysis and Manipulation*, 2001, pp. 33–42.
- [12] 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.
- [13] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee, "Software-controlled fault tolerance," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2005.
- [14] J. Reis, G.A. andChang, N. Vachharajani, S. Mukherjee, R. Rangan, and D. August, "Design and evaluation of hybrid fault-detection systems," in *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*, June 2005, pp. 148–159.
- [15] P. P. Shirvani, N. Saxena, and E. J. McCluskey, "Software-implemented edac protection against seus," *IEEE Transactions on Reliability*, vol. 49, pp. 273–284, 2000.
- [16] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi, "Modeling the effect of technology trends on soft error rate of combinational logic," in *Proceedings of the International Conference on Dependable Systems and Networks*, June 2002.
- [17] K. Wilken and J. Shen, "Continuous signature monitoring: low-cost concurrent detection of processor control errors," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 9, no. 6, pp. 629–641, June 1990.