Techniques for Efficient Software Checking *

Jing Yu, María Jesús Garzarán, and Marc Snir

University of Illinois at Urbana-Champaign
jingyu,garzaran,snir@cs.uiuc.edu

Abstract. Dramatic increases in the number of transistors that can be integrated on a chip make processors more susceptible to radiation-induced transient errors. For commodity chips which are cost- and energy-constrained, we need a flexible and inexpensive technology for fault detection. Software approaches can play a major role for this sector of the market because they need little hardware modifications and can be tailored to fit different requirements of reliability and performance. However, software approaches add a significant overhead.
In this paper we propose two novel techniques that reduce the overhead of software error checking approaches. The first technique uses boolean logic to identify code patterns that correspond to outcome tolerant branches. We develop a compiler algorithm that finds those patterns and removes the unnecessary replicas. In the second technique we evaluate the performance benefit obtained by removing address checks before load and stores. In addition, we evaluate the overheads that can be removed when the register file is protected in hardware.
Our experimental results show that the first technique improves performance by an average 7% for three of the SPEC benchmarks. The second technique can reduce overhead by up-to 50% when the most aggressive optimization is applied.

1 Introduction

Dramatic increases in the number of transistors that can be integrated on a chip will deliver great performance gains. However, it will also expose a major roadblock, namely the poor reliability of the hardware. Indeed, in the near-future environment of low power, low voltage, relatively high frequency, and very small feature size, processors will be more susceptible to transient errors. Transient faults, also known as soft errors are due to impacts from high-energy particles that change the logic values of latches or logic structures [1–4].

In this new environment, we believe that a Software Checking System has a fundamental role in providing fault detection and recovery. It is possible that high-end architectures will include several hardware-intensive fault-tolerance techniques that are currently supported by IBM mainframes [5], HP NonStop [6] or mission-critical computers [7]. However, commodity multicore chips will likely be too cost- and energy-constrained to include such hardware. Instead, we believe that they will likely include only relatively simple hardware primitives, such as parity for certain processor buses and structures, error correction codes (ECC) and scrubbing in the

* This material is based upon work supported by the National Science Foundation under the CSR-AES program Award No. 0615273

memory hierarchy [8] and low-cost support for memory checkpointing and rollback (e.g., ReVive [9] or SafetyNet [10]). Then they will rely on flexible and inexpensive software technology for error protection.

Current software approaches address the problem by replicating the instructions and adding checking instructions to compare the results, but they add a significant overhead. In this paper we propose two novel techniques to reduce the overhead of the software error checking approaches. The first technique is based on the fact that programs already have redundancy, and if the compiler can determine the programs sections where such redundancy exists, it can avoid the replication and later checking. We use boolean logic to identify a code pattern that corresponds to outcome tolerant branches and develop a compiler algorithm that automatically finds those patterns and removes the unnecessary replicas. The second technique is based on the observation that faults that corrupt the application tend to quickly generate other noisy errors such as segmentation faults [11]. Thus, we can reduce replication of the instructions that tend to generate these type of errors, trading reliability for performance. In this paper we remove the checks of the memory addresses and discuss situations where removing these checks affect little to the fault coverage. This occurs when a check of a variable is covered by a later check to the same variable, and thus errors in the first check will be detected by the later checks, and in pointer-chasing, when the data loaded by a load is used immediately by another load. Finally, We also consider the situation where the register file is protected with parity or ECC, such as Intel Itanium [12], Sun UltraSPARC [13] and IBM Power4-6 [14]. We call them register safe platforms.

We have implemented the baseline replication and the proposed techniques using the LLVM Compiler Infrastructure [15] and run experiments on a Pentium 4 using Spec benchmarks. Our results show that the boolean logic technique achieves 7% performance speedup on three benchmarks, and 1.6% on average. If we do not check load addresses, the performance is improved by 20.2%. If we do not check addresses of both load and store, the performance is improved by 24.8%. On platforms where registers are protected in hardware, we can combine these techniques and obtain an average speedup of 35.2% and 40.8%, respectively, and decrease the software checking overhead by 44.9% and 50%, respectively. Our fault injection experiments show that removing address checks before loads only increases Silent Data Corruption (SDC) from 0.27% to 0.35%, and removing address checks for loads and stores raises SDC to 1.11%.

The rest of the paper is organized as follows. Section 2 presents the background and the baseline software checking; Section 3 describes the techniques to detect outcome tolerant branches; Section 4 describes the removal of address checks; Section 5 discusses the benefits of having a register file that is checked in hardware; Section 6 presents our experimental results; Section 7 presents related work, and finally Section 8 concludes the paper.

2 Background and Baseline Software Checking

The use of software approaches for fault tolerance has received significant attention in the research domain. Software techniques such as SWIFT [16] replicate the instructions of the original program and interleave the original instructions and their

replicas in the same thread. Memory does not need to be replicated because the memory hierarchy is protected with ECC and scrubbing. Stores, branches, function calls and returns are considered "synchronization" points and checking instructions are inserted before these instructions to validate certain values. Before a store, checking instructions verify that the correct data is stored to the correct memory location. Before a branch, checking instructions verify that the branch takes the appropriate path. Before a function call checking instructions verify the input operands by comparing them against their replica. Before a function return, checking instructions verify the return value by comparing the return register and its replica.

Stores are executed only once, but loads are replicated because the loaded data can be corrupted. However, uncachable loads, such as those from external devices, and loads in a multithreaded program may return different values when executing two consecutive loads to the same memory address; so rather than replicating the load, checking instructions are also added before loads to verify that the address of the load matches its replica. After that verification, the loaded value can be copied to another register [16–18]. Thus, since loads are not replicated, they are also considered "synchronization" points. An example with the original and its corresponding replicated code is shown in Figure 1-(a) and (b), respectively. The replicated code contains additional instructions and uses additional registers marked with a `'`. The additional instructions are shown in bold and numbered. Instructions 1 and 2 check that the `load` is loading from the correct address, instruction 3 copies the value in `r3` to `r3'`, instruction 4 replicates the addition, and instruction 5-8 check that the store writes the correct data to the correct memory address.

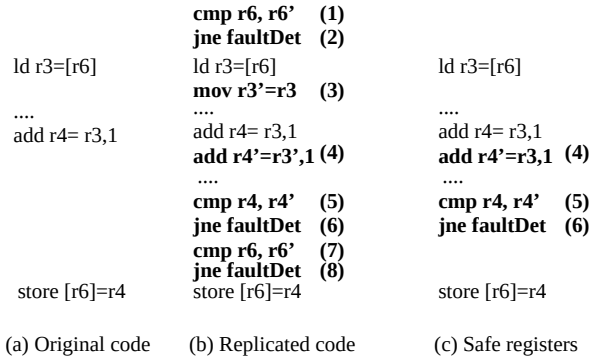

Fig. 1. Example of baseline software replication and checking

3 Use of Boolean Logic to Find Outcome Tolerant Branches

In this Section we explain how to use boolean logic to reduce the amount of replicated instructions. We first do an overview (Section 3.1) and then explain the compiler algorithm (Section 3.2).

3.1 Overview

Our technique is based on the fact that programs have redundancy. For instance, Wang et al. [19] performed fault injection experiments and found that about 40% of all the dynamic conditional branches are outcome tolerant. These are branches that, despite an error, converge to the correct point of execution. These branches are outcome-tolerant due to redundancies introduced by the compiler or the programmer. An example of outcome-tolerant branch appears in a structure such as `if (A || B || C) then X else Y`. In this case if `A` is erroneously computed to be true, but `B` or `C` are actually true, this branch is outcome tolerant, since the code converges to the correct path. The control flow graph of this structure is shown in Figure 2-(a).

The state-of-the-art approach to check for errors is to replicate branches as shown in Figure 2-(b), where the circles correspond to the branch replicas. However, we can reduce overheads by removing the comparison replica when the branch correctly branches to `X`. If the original comparison in `A` is true we need to execute the comparison replica to verify that the code correctly branches to `X`. However, if `A` is false, we can skip the execution of the `A` replica and move to check `B`. We will only need to execute the A replica if both `B` and `C` are also false. The resulting control flow graph is shown in Figure 2-(c). In situations where `A` and `B` are false, but `C` is true, we can save a few comparisons.

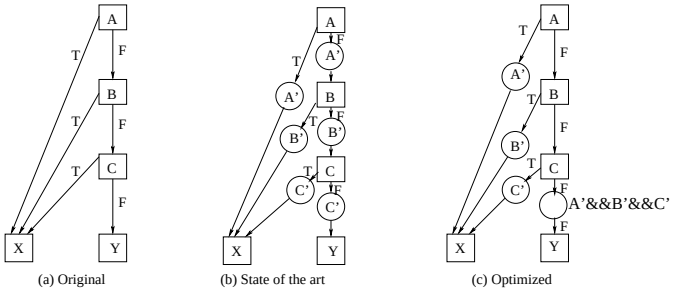

Fig. 2. Eliminating replicated predicate evaluation.

Outcome tolerant branches also appear in code structures such as `if (A & B & C) then X else Y`, and in general in all the code structures that contain one or more shortcut paths in the control flow graph. A basic *shortcut path* is `edge(A->X)` in Figure 3-(a), where both `A` and its child point to the same block. However, most shortcut paths are more complex. For instance, in Figure 3-(b), block `A` points to the same block pointed by its grandchild (not its direct child). Thus, the optimizer should move `A'` from `edge(A->B)` to `edge(B->Z)` and `edge(C->Y)`. The example in Figure 3-(c) can be optimized in two different ways. If `A` and `B` are considered as a whole unit, `edge(B->Y)` is the shortcut path, and the graph can be optimized as shown in Figure 3-(d); otherwise, it can be optimized as shown in Figure 3-(e).

Detecting the existence of a shortcut paths is not sufficient to determine that there is an outcome tolerant branch. The reason is that one of the blocks involved

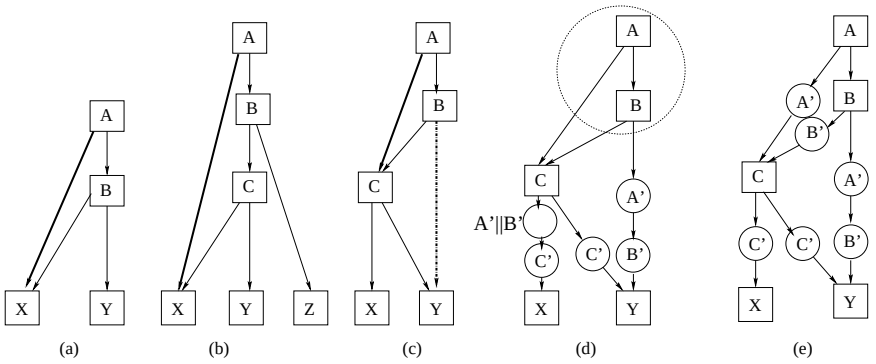

Fig. 3. Shortcut graphs and optimizations

in the shortcut can modify a variable that is later used by instructions outside the block. That block needs to be replicated or the error could propagate outside the block. Next we show two examples:

```
(a) if (*m > 0) && (m < N) then X else Y
(b) if (t=(*m > 0)) && (m < N) then X else Y
```

In the example in (a), if `(*m>0)` is mistakenly computed as True, but `(m<N)` is False, we can safely ignore the error on `(*m>0)` and take the `Y` path. However, if the error occurs to the example in (b), and `t` is used in `Y`, ignoring the error will result in a wrong value for `t` being propagated to `Y`, which may end up corrupting the system. To avoid this type of errors our compiler algorithm only considers blocks that are involved in a shortcut path and produce values that are only used by the block itself.

3.2 Compiler Algorithm

Our algorithm analyzes the control flow graph of the original program and extracts the shortcut paths and the related blocks. A *shortcut graph* always has a head node (block `A` in all the examples in Figure 3), one or more intermediate nodes (like `B` and `C`), two or more leaves (like `X` and `Y`), and one or more shortcut paths. Notice that in this paper we call a *block* to a single basic block or a list of basic blocks connected one by one with edges of unconditional branches.

Our algorithm has two phases: first a search of all potential shortcut graphs, and second, the optimization and appropriate placement of the replicas.

Shortcut Graphs Search The searching process starts by classifying each block as an intermediate node or a leaf, and building an intermediate node set and a leaf set. A block is called "intermediate node" if it ends with a conditional branch and does not contain side effects (does not contain a function call, a memory write or generates a value used by another block). In addition, to avoid being trapped in loops, we require that none of the outgoing edges of an intermediate node is a backward edge. If the node does not classify as intermediate node, then it is

considered a "leaf", meaning that this block can be at the most an ending node in a shortcut graph. At the same time we build the intermediate and leaf sets, we also build a separate head node set, which contains all intermediate nodes and some of the leaves. A head node is classified as leaf when the block has function calls or memory writes before the conditional branch. However, as long as a block ends with a conditional branch and none of the outgoing edges is backwards, it is also considered a head node.

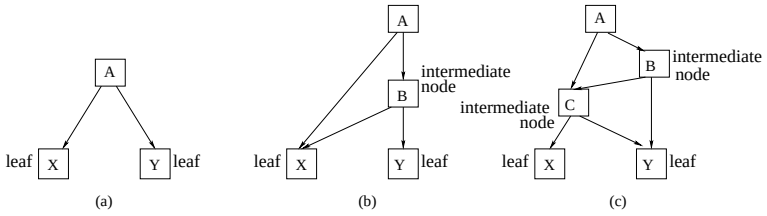

Fig. 4. Constructing potential shortcut graphs.

After building the intermediate node set, the leaf set, and the head node set the shortcut graphs are built from bottom to up by scanning the head node set repeatedly. We start by initializing an empty set "graph-head-set", which will contain temporary graph head nodes. For any node(`A`) in the head node set, we check its two children (see Figure 4):

1. If the two children are leaves, this node is added to the graph-head-set (Figure 4-(a)).
2. If one child is a leaf(`X`) and the other child is an intermediate node(`B`) and node(`B`) is already in the graph-head-set, node(`B`) is replaced by the current node(`A`) in the graph-head-set (Figure 4-(b)). We also check if the leaf(`X`) is a child or grandchild of node(`B`), in which case a shortcut path for node (`A`) is marked.
3. If the two children are both intermediate nodes((`B`) and (`C`)) and both are in the graph-head-set, nodes (`B`) and (`C`) are replaced by node(`A`) in the graph-head-set (Figure 4-(c)). We also check if (`A`) introduces new shortcut paths.

The scan continues until all the nodes in the head node set have been visited. Then, a node in the graph-head-set represents a graph led by this node together with the shortcut paths found. A final pass traverses the graph-head-set and removes those heads that do not contain any shortcut path.

Optimization After the shortcut paths are found we start applying the optimization, but we first check when it is legal to perform it. In Figure 2-(a), our optimization will move the replica `A'` from `edge(A->B)` to `edge(C->Y)`. However, this is only legal if `A` dominates `C`. Otherwise `A'` may use undefined values in the new position. Thus to apply our optimization phase we first verify the domination relationship of all shortcut paths.

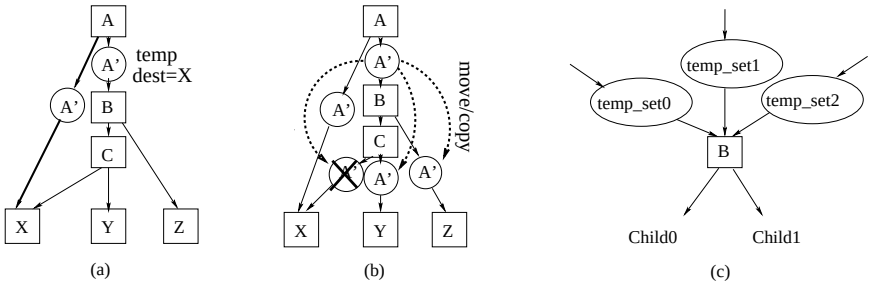

Fig. 5. Optimizing shortcut graphs.

The goal of our optimization pass is to move replicas of the non-shortcut path down to the edge/s between the last child and the leaf/leaves. Next, we explain how this algorithm proceeds using the example in Figure 5. For each shortcut graph in the graph-head-set the algorithm finds all the shortcut paths (`edge(A->X)` in Figure 5-(a)), marks the replica (`A'`) on the other path as temporary (temp), and records the destination of the shortcut path (`X`). Next the optimization pass scans all the intermediate nodes in the shortcut graph in a top-down fashion, and moves temporary replicas from the incoming edges to all the outgoing ones, except to those where the recorded destination of the replica and the destination of the intermediate node that we are processing are the same (an example is shown in Figure 5-(b)). Notice that when an intermediate node has multiple incoming edges (as shown in Figure 5-(c)) we only move the replicas that appear on all the incoming edges. Also notice that this optimization pass processes nodes top-down, and it does not treat multiple nodes as a single unit. Thus, for the example in Figure 3-(c), the optimized version after this pass will be the one shown in Figure 3-(e).

Finally note that `A`, `B` and `C` can contain computations like `(s+1) == 5`. In this case, if the computations are only used to determine the outcome of the branch, the computation replicas are also eliminated when the branch replica does not need to execute.

4 Removal of Address Checks

Recent experiments have shown that faults produce not only data corruption, but also events that are atypical of steady state operation and that can be used as a warning that something is wrong [11]. Thus, we can reduce the overhead of the software approaches and trade reliability for performance by reducing the replication, hoping that the error will manifest with these atypical events.

In this Section we consider the removal of address checks before load and store instructions. Errors in the registers containing memory addresses may manifest as segmentation faults. However, any fault-tolerant system must also include support for roll-back to a safe state and thus, on a segmentation fault we can roll-back and re-execute, and only communicate the error to the user if it appears again. However, by doing this the system will be vulnerable to errors, since some of these faulty addresses will access a legal space and the operating system will not be able

to detect the error. Thus, this technique will decrease error coverage. Next, we discuss two techniques that the compiler can use to determine which load and store instructions are most suitable for address check removal.

Address checks can be removed when there are later checks checking the same variable. For example, in Figure 1-(b), checking instructions (1-2) and (7-8) are checking the register `r6`. This makes the first check (1-2) unnecessary, because if an error occurs to `r6` it will manifest as a segmentation fault or will be eventually detected by the checking instructions (7-8). We have observed many of these checks in the SPEC benchmarks due to the register indirect addressing mode, since the same register is used to access two fields of a structure, or because two array accesses share a common index. Removing these replicated checks can significantly reduce the software overhead.

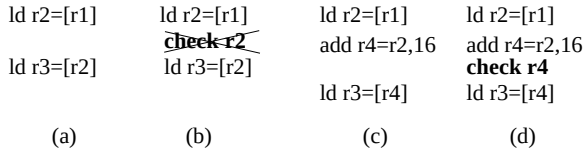

Fig. 6. Address check removal for pointer chasing

Address checks can also be removed when the probability of error is small. This case appears in pointer chasing, where the data loaded from memory is used as the address for a subsequent load. An example is shown in Figure 6-(a) and (b). In this case, since the processor will issue the second load as soon as the first one completes, the probability of error is very small. In some cases, however, the value loaded by the first load is not exactly the one used by the next load, if not that it may be first modified by an `add` instruction. This occurs when accessing an element of a structure that is different from the first one. In this case, the probability of error is higher, and the checking instructions will also determine if an error occurred during the computation of the addition. An example is shown in Figure 6-(c) and (d).

In this paper we evaluate the removal of the address checks for only the loads, or for both loads and stores. Thus, our results are an upper bound on the performance benefit that we can obtain and the reliability that we can lose. In the future we plan to write a data flow analysis to identify the checks that are safe to remove, as explained above.

5 Register Safe Platforms

In this Section we consider the situation where the register file is hardware protected with parity or ECC, or other cost-effective mechanisms as the ones proposed by [20–23]. In fact, the register file of the Intel Itanium [12], Sun UltraSPARC [13] and IBM Power4-6 [14] are already protected by parity or ECC. However, the ALUs and other portions of the processor are not protected, so arithmetic and logic operations can return wrong results. Thus, all the instructions that imply ALU operations need to

be replicated; however, memory operations such as load and stores are safe. As a result, a register that is defined by a load does not need to be replicated, saving the instruction to perform the copy and the additional register. An example is shown in Figure 1. The replicated code in Figure 1-(b) can be simplified as shown in Figure 1-(c). Register `r3'` is not necessary because registers and memory are safe, and instruction 4 can use directly the contents from register `r3`. Instructions 1, 2, 7 and 8 can be removed if we assume register `r6` has been defined by a load. Instructions 5 and 6 cannot be removed because register `r4` is defined by an addition, and we need to validate the results of the addition.

6 Evaluation

In this Section we evaluate our proposed techniques. We first discuss our environmental setup (Section 6.1), analyze our techniques statically (Section 6.2), evaluate performance (Section 6.3), and measure reliability (Section 6.4).

6.1 Environmental Setup

We use LLVM [15] as our compiler infrastructure to generate redundant codes. Replicated and checking instructions are added at the intermediate level, right after all the static optimizations have been done. We replicate all the integer and floating point instructions. Previous implementations have replicated instructions at the backend, right before register allocation [16, 24] or via dynamic binary translation [25]. However, the advantages of working at the intermediate level are: i) the redundant code can be easily ported to other platforms, ii) we do not need to fully understand the assembly code for that platform, and iii) at the intermediate level we see a simple memory access model rather than complex one of the x86 ISA. To prevent optimizations done by the backend generator such as common subexpression elimination and instruction combination, we tag the replicated instructions, and the backend optimizations are applied separately to the tag and the untag instructions.

For the evaluation we use SPEC CINT2000 and the C codes from SPEC CFP2000, running with the ref inputs. Experiments are done on a 3.6GHz INTEL Pentium 4 with 2GB of RAM running RedHat9 Linux.

6.2 Static Analysis

In this Section we characterize load addresses depending on whether the register is checked by a later checking instruction (Covered), or if the register used by the load was just loaded from memory (Loaded), as in the pointer chasing example of Section 4. All the remaining load addresses are classified as (Other). The breakdown is shown in Figure 7. On average more than 40% load addresses have nearby later checks on the same value. About 20% of the loads use registers whose contents where just loaded from memory. As we have discussed in Section 4, the probability of error of any of these addresses is very small, because the processor will likely issue the second load as soon as the first one completes. Also, if we assume a register safe platform these checks are unnecessary. For the remaining 40% of the addresses, an error in the most significant bits will be detected as a form of segmentation faults, but an error in the least significant ones can cause a silent error.

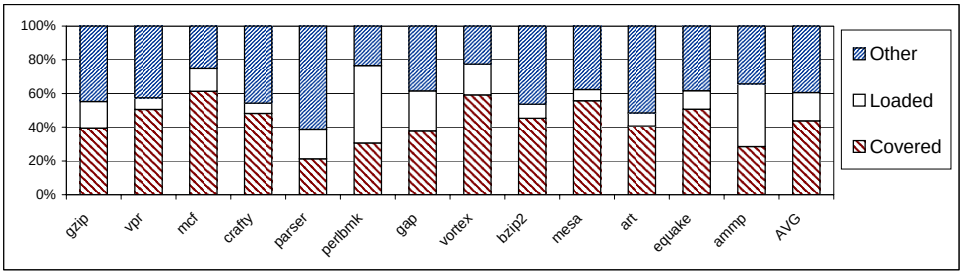

Fig. 7. Characterization of load addresses

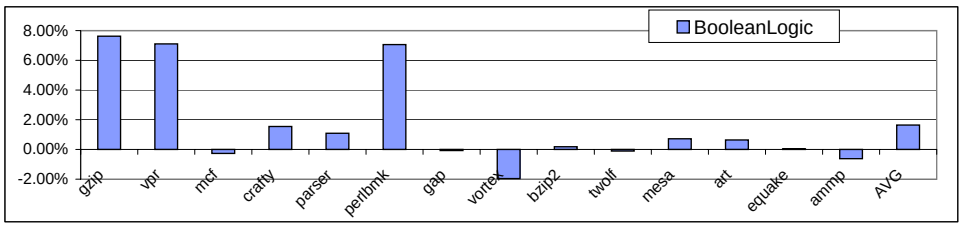

Fig. 8. Performance speedup with boolean logic optimization compared to baseline replication

6.3 Performance

Figure 8 shows the performance speedup obtained when using boolean logic to eliminate replication and checks on outcome tolerant branches (Section 3). Three benchmarks (gzip, vpr, and perlbmk) achieve 7% performance gains, though the average speedup is 1.6% through all tested benchmarks. Notice that there is also a negative impact on vortex, where we observe more load/store instructions after the optimization, meaning that this optimization introduces additional register spills that hurt the benefit of less dynamic instructions.

Figure 9 evaluates the performance benefit of our second technique (Section 4): baseline Fully Replicated(FullRep), No checks for Address of Loads(NAL), No checks for Address of Load and Store(NALS), and No checks when the Register file is safe (R). The Fully Replicated code(FullRep) is on average 2.38 times slower than the original code. This large overhead is due to high register pressure and additional instructions. On average, register safe optimization (R) runs 16.0% faster than the (FullRep).

After we remove checks for address of loads (NAL), we get an average 20.2% speedup over the baseline Fully Prelicated (FullRep). If we further remove checks for address of stores (NALS), we improve 4.6% more. And if the register is protected in hardware and we combine (NAL) or (NALS) with (R), we can obtain an average speedup of 35.2% and 40.8% respectively, what will reduce the the software checking overhead by 44.9% and 50%, respectively. Notice that with (NALS) all address checks before loads and stores are removed, so the performance benefit of (R+NALS) versus (NALS) is due to the reduced register pressure (the register of the load does

not need to be replicated) and the removal of a few additional checks before the data being stored.

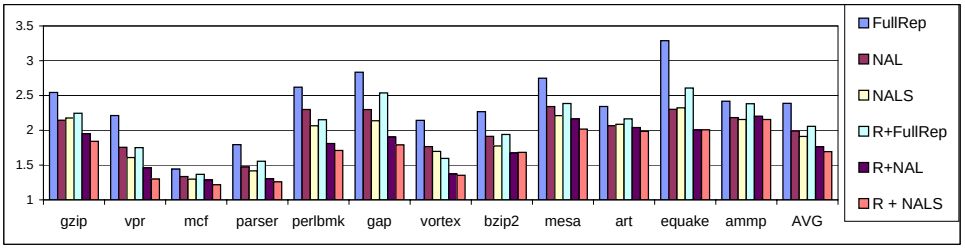

Fig. 9. Performance of the different optimizations normalized against the original non-replicated code.

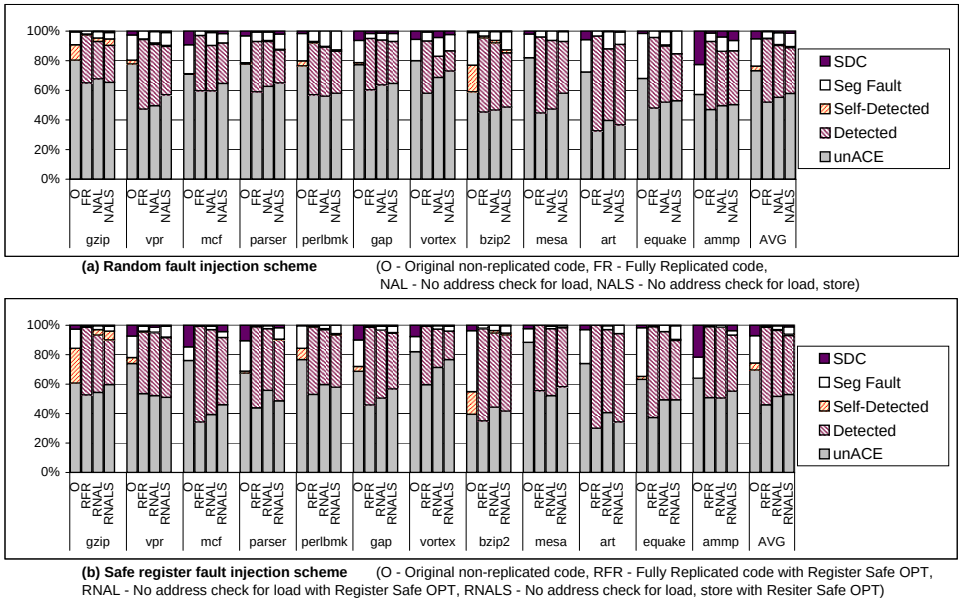

Fig. 10. Fault-detection rates break down

6.4 Reliability

Our first technique is very conservative and should not affect the fault coverage. But for the second technique, since we remove all the checks for memory addresses, memory can be corrupted. In order to evaluate the loss of fault coverage, we use Pin [26] and inject faults to the binary file (excluding system libraries). We assume

a Single Event Upset(SEU) fault model, so only one bit fault is injected during the execution of the program. In total 300 faults are injected for each program. Although both integer and floating point registers can be corrupted, in order to magnify the impact of the errors we only inject fault to the 8 32-bit integer registers and the status flags EFLAGS. When we consider that the register file is not protected in hardware we mimic the fault distribution by randomly selecting a dynamic instruction and flipping a random bit in a random register (we call this scheme "random fault injection"). When the register file is protected in hardware, we do the same, but flip the random bit from its "output". The output can be in a register or in memory if it has been spilled. In this scheme, memory load instructions are avoided (we call this scheme "safe register fault injection").

After injecting an error into the binary, the program is run to completion (unless it aborts) and its output is compared to a correct output. Depending on the result the error will be categorized as: (unACE), the bit is unnecessary for Architectural Correct Execution [27]; (Detected), the error is detected by our checking code; (Self-Detected), the error is detected by the program assertions; (Seg Fault), the error manifests as an exception or a segmentation fault; (SDC), Silent Data Corruption, when the program finishes normally but the produced output is incorrect. (SDC) is the first type of errors we want to prevent. Then, we also want to minimize (self-Detected) errors and (Seg Fault), because it is usually hard to determine if the error is due to a program bug or a soft error. But with proper support, if we can roll-back and re-execute, these faults can be recovered, so they are less harmful.

Figure 10-(a) and (b) show the experimental results for random fault injection and safe register fault injection, respectively. The fault detection rates for these two schemes are very similar. Notice that the original program (O) has on average 75% (unACE) and less than 10% (SDC), which means that the software itself has a certain fault maskability. With the safe register scheme more faults result in SDC than with the random scheme (8.5% over 5%) and less Seg Fault (15.6% over 17%). The reason is that the random scheme is more likely to pick up a dynamic dead register or a register that holds the index for addresses.

After the program is replicated (FR), most (Seg Fault), (Self-Detected) and (SDC) go to the (Detected) category. (SDC) errors appear because some faults are injected before the value is used but after is checked. If we remove checks for addresses, reliability does not drop much. Under random injection scheme, if we remove checks for load addresses (NAL), comparing to (FR), (SDC) increases from 0.36% to 1.08%, (Seg Fault) increases from 4.47% to 8.05%. If we also remove checks for store addresses (NALS), (SDC) rises to 1.44%, and (Seg Fault) rises to 9.02%. Under safe register injection scheme, removing checks for load addresses increases (SDC) from 0.27% to 0.38%, increase (Seg Fault) from 2.66% to 4.99%. Removing checks for store addresses further results in (SDC) of 1.11%, and (Seg Fault) of 4.99%. In other words, when normalized to the original program, under the safe register scheme removing checks for addresses of load only incurs an extra 1.3% (SDC), while removing checks for all addresses incurs 9.8%(SDC). Given that we almost decrease the performance overhead by half, this loss of fault coverage seems acceptable.

7 Related Work

Previous work on compiler instrumentation for fault tolerance focuses on replication and checking. There have been previous works on software checking optimization. For example, SWIFT [16] merges checks before branches into control flow signature checks, and removes checks for blocks that do not have stores. In this paper, we propose a new area for optimization: when the code structure itself can mask errors and the compiler can determine those programs sections, replication and later checking can be avoided.

Some previous works provide ways to trade reliability for performance. For example, the work by Oh and McCluskey [28] selectively duplicates procedure calls instead of replicating instructions inside them. This way error detection latency is sacrificed for less power consumption. But for each procedure, either all the instructions in the procedure or the call needs to be replicated. PROFiT [29] and Spot [25] divide program into regions and pick up only important regions to do software replication and checking. Spot provides very flexible selection granularity, ranging from a few blocks to a whole procedure. However, in Spot making a good selection requires knowledge of fault mask probability and replication overhead for each region. Jonathan Chang et. al [20] propose to protect a portion of the register file based on a profile of register life time and usage. For different platforms or different programs, the protected portion may be different. In this paper, we provide a fine and simple leverage control: we choose to remove checks for addresses of load or stores. With static compiler analysis, this technique can be applied independently of the target platform. Furthermore, we can combine this technique with previous ones to trade fault coverage with performance.

Previous works on compiler instrumentation for fault-tolerance implement their techniques at the source level [30], compiler backend [16, 24, 31, 29, 32], or runtime binary level [25]. However, our techniques are implemented at the intermediate level, which makes it portable across platforms and friendly to users who are not expert on the target ISA.

8 Conclusion

This paper makes several contributions. First, we identify a code pattern that corresponds to outcome tolerant branches, and develop a compiler algorithm that finds these patterns, avoiding unnecessary replication and checking. Second, we evaluate the removal of address checks for loads and stores, and analyze situations where these checks can be removed with little loss of fault coverage. We also identify the check and replicated registers that can be removed on a register safe platform.

Optimizing outcome tolerant branches obtains 7% performance speedup for 3 benchmarks, and an average of 1.6% for all, while keeping the same level of reliability. We also find that on register safe platforms removing the checks for the addresses of load reduce the replication overhead by 44.9%, and only increases SDC (Silent Data Corruption) rate from 0.27% to 0.38%. Also, if 1.11% SDC rate is acceptable, we can furthermore reduce the replication overhead by 50% by also removing checks for the store addresses.

References

1. Constantinescu, C.: Impact of Deep Submicron Technology on Dependability of VLSI Circuits. In: Proc. of the International Conf. on Dependable Systems and Networks. (2002) 205–209
2. Hazucha, P., Karnik, T., Walstra, S., Bloechel, B., Tschanz, J.W., Maiz, J., Soumyanath, K., Dermer, G., Narendra, S., De, V., Borkar, S.: Measurements and Analysis of SER-tolerant Latch in a 90-nm dual-V/sub T/ CMOS Process. IEEE Journal of Solid-State Circuits **39**(9) (September 2004) 1536–1543
3. Karnik, T., Hazucha, P.: Characterization of Soft Errors Caused by Single Event Upsets in CMOS Processes. IEEE Transactions on Dependable and Secure Computing **1**(2) (April-June 2004) 128–143
4. Shivakumar, P., Kistler, M., Keckler, S., Burger, D., Alvisi, L.: Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. In: Proc. of the International Conf. on Dependable Systems and Networks. (2002) 289–398
5. Slegel, T., Averill, R., Check, M., Giamei, B., Krumm, B., Krygowski, C., Li, W., Liptay, J., MacDougall, J., McPherson, T., Navarro, J., Schwarz, E., Shum, K., Webb, C.: IBM's S/390 G5 Microprocessor Design. IEEE Micro **19**(2) (March-April 1999) 12–23
6. McEvoy, D.: The architecture of tandem's nonstop system. In: ACM 81: Proceedings of the ACM '81 conference, New York, NY, USA, ACM Press (1981) 245
7. Yeh, Y.: Triple-triple Redundant 777 Primary Flight Computer. In: Proc. of the IEEE Aerospace Applications Conference. (1996) 293–307
8. Mukherjee, S., Emer, J., Fossum, T., Reinhardt, S.: Cache Scrubbing in Microprocessors: Myth or Necessity? In: Proc. of the Pacific RIM International Symposium on Dependable Computing. (2004) 37–42
9. Prvulovic, M., Zhang, Z., Torrellas, J.: ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In: Proc. of the International Symposium on Computer Architecture (ISCA). (2002)
10. Sorin, D., Martin, M., Hill, M., Wood, D.: SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In: Proc. of the International Symposium on Computer Architecture (ISCA). (2002)
11. Wang, N.J., Patel, S.J.: ReStore: Symptom Based Soft Error Detection in Microprocessors. In: Proc. of the International Conference on Dependable Systems and Network (DSN). (2005) 30–39
12. McNairy, C., Bhatia, R.: Montecito: A Dual-core, Dual-thread Itanium Processor. IEEE Micro **25**(2) (March-April 2005) 10–20
13. Kongetira, P., Aingaran, K., Olukotun, K.: Niagara: A 32-way multithreaded sparc processor. IEEE Micro **25**(2) (2005) 21–29
14. Bossen, D., Tendler, J., Reick, K.: Power4 system design for high reliability. IEEE Micro **22**(2) (March/April 2002) 16–24
15. Lattner, C., Adve, V.: The LLVM Compiler Framework and Infrastructure Tutorial. In: LCPC'04 Mini Workshop on Compiler Research Infrastructures. (2004)
16. Reis, G.A., Chang, J., Vachharajani, N., Rangan, R., August, D.I.: SWIFT: Software Implemented Fault Tolerance. In: Proc. of the International Symposium on Code Generation and Optimization (CGO). (2005)
17. Mukherjee, S.S., Kontz, M., Reinhardt, S.K.: Detailed Design and Evaluation of Redundant Multithreading Alternatives. In: Proc. of International Symposium on Computer Architecture, Washington, DC, USA, IEEE Computer Society (2002) 99–110
18. Reinhardt, S.K., Mukherjee, S.S.: Transient Fault Detection via Simultaneous Multithreading. In: Proc. of International Symposium on Computer Architecture, New York, NY, USA, ACM Press (2000) 25–36

19. Wang, N., Fertig, M., Patel, S.: Y-Branches: When You Come to a Fork in the Road, Take It. In: Proc. of the International Conference on Parallel Architectures and Compilation Techniques (PACT). (2003)
20. Chang, J., Reis, G.A., Vachharajani, N., Rangan, R., August, D.: Non-uniform fault tolerance. In: Proceedings of the 2nd Workshop on Architectural Reliability (WAR). (2006)
21. Gaisler, J.: Evaluation of a 32-bit microprocessor with built-in concurrent error-detection. In: FTCS '97: Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97), Washington, DC, USA, IEEE Computer Society (1997) 42
22. Montesinos, P., Liu, W., Torrellas, J.: Shield: Cost-Effective Soft-Error Protection for Register Files. In: Third IBM TJ Watson Conference on Interaction between Architecture, Circuits and Compilers (PAC206). (2006)
23. Hu, J., Wang, S., Ziavras, S.G.: In-register duplication: Exploiting narrow-width value for improving register file reliability. In: DSN '06: Proceedings of the International Conference on Dependable Systems and Networks (DSN'06), Washington, DC, USA, IEEE Computer Society (2006) 281–290
24. Reis, G.A., Chang, J., Vachharajani, N., Rangan, R., August, D.I., Mukherjee, S.S.: Design and Evaluation of Hybrid Fault-Detection Systems. In: Proc. of the International International Symposium on Computer Architecture (ISCA). (2005)
25. Reis, G.A., J.Chang, August, D.I., Cohn, R., Mukherjee, S.S.: Configurable Transient Fault Detection via Dynamic Binary Translation. In: Proceedings of the 2nd Workshop on Architectural Reliability (WAR). (2006)
26. Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In: Proc. of the Intenational Conference on Programming Language Design and Implementation (PLDI). (2005)
27. Mukherjee, S.S., Weaver, C., Emer, J., Reinhardt, S.K., Austin, T.: A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In: MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture, Washington, DC, USA, IEEE Computer Society (2003) 29
28. Oh, N., McCluskey, E.J.: Low Energy Error Detection Technique Using Procedure Call Duplication. In: Proc. of the International Conference on Dependable Systems and Network (DSN). (2001)
29. Reis, G.A., Chang, J., Vachharajani, N., Rangan, R., August, D.I., Mukherjee, S.S.: Software-controlled fault tolerance. ACM Trans. Archit. Code Optim. **2**(4) (2005) 366–396
30. M.Rebaudengo, Reorda, M.S., Violante, M., Torchiano, M.: A Source-to-Source Compiler for Generating Dependable Software. In: IEEE International Workshop on Source Code Analysis and Manipulation (SCAM). (2001) 35–44
31. Oh, N., Shirvani, P., McCluskey, E.J.: Error Detection by Duplicated Instructions in Super-scalar Processors. IEEE Transactions on Reliability **51**(1) (March 2002) 63–75
32. Chang, J., Reis, G.A., August, D.I.: Automatic Instruction-Level Software-Only Recovery. In: DSN '06: Proceedings of the International Conference on Dependable Systems and Networks (DSN'06), Washington, DC, USA, IEEE Computer Society (2006) 83–92