ESoftCheck: Removal of Non-vital Checks for Fault Tolerance

Jing Yu
Google Inc.
jingyu@google.com

María Jesús Garzarán, Marc Snir
University of Illinois at Urbana-Champaign
{garzaran, snir}@cs.uiuc.edu

***Abstract*—As semiconductor technology scales into the deep submicron regime the occurrence of transient or soft errors will increase. This will require new approaches to error detection. Software checking approaches are attractive because they require little hardware modification and can be easily adjusted to fit different reliability and performance requirements. Unfortunately, software checking adds a significant performance overhead.**

In this paper we present ESoftCheck, a set of compiler optimization techniques to determine which are the vital checks, that is, the minimum number of checks that are necessary to detect an error and roll back to a correct program state. ESoftCheck identifies the vital checks on platforms where registers are hardware-protected with parity or ECC, when there are redundant checks and when checks appear in loops. ESoftCheck also provides knobs to trade reliability for performance based on the support for recovery and the degree of trustiness of the operations. Our experimental results on a Pentium 4 show that ESoftCheck can obtain 27.1% performance improvement without losing fault coverage.

I. Introduction

Transient errors, also known as soft errors, are due to impacts from high-energy particles or other random external events that change the logic values of latches or logic structures. These errors are temporary, but they caused costly failures in high-end systems in recent years, such as crashes at some of Sun's customer sites, including American Online and eBay [2] and Los Alamos Labs supercomputers [16]. The continued evolution of hardware toward smaller feature size, lower voltage, and higher frequency suggests that fault rates will increase in the future. Thus, error detection mechanisms are necessary to ensure that a soft error does not go undetected and results in an erroneous computation. Once errors are detected, it is often possible to use software schemes for error correction – the performance of error correction schemes is not critical, as long as errors are not too frequent; however, error detection adds an overhead to all computations and has to perform efficiently. For this reason, we focus in this paper on error detection.

Hardware-based error detection is used on modern microprocessors to detect errors in storage and buses: for example, ECC memory and parity bits for caches and various buses, which generally add a low overhead to performance and chip size. On the other hand, it is much harder to detect errors in the pipeline: current solutions include watchdog coprocessors [12], redundant hardware threads [7], [19], [25], [32] or the replication of significant fractions of the CPU logic, as in the high-end IBM mainframes [30], HP NonStop [13] or mission-critical computers [36]. However, it is unclear whether such hardware cost is acceptable for commodity systems. For such systems, software-based error detection may be a preferable solution. Software checking approaches are less costly in terms of hardware and are more flexible: different trade-offs between performance and reliability can be achieved on the same hardware, using different software approaches, while hardware-only solutions cannot offer the same flexibility. Such flexibility can be used, for example, to achieve different levels of reliability for different software components: one may not care about undetected errors that will affect the PC display during a game, but may want to avoid errors that will corrupt the file system metadata. Recently, researchers have also proposed hybrid software-hardware approaches [27].

Our approach is to provide full error detection coverage for the whole program or for a whole section of the program. Thus, our baseline software-based approach is to replicate computing instructions keeping two copies of the data and add checks before stores and other synchronization instructions to compare the copies and ensure that data stored in memory are correct [26], [27], [33]. In this paper we propose EsoftCheck, a set of compiler optimization techniques that reduce the amount of checks by identifying those that are vital – rather than adding checks before all stores and synchronization instructions. Vital checks are the minimum set of checks that are necessary to detect an error and roll back to a correct program state. Thus, ESoftCheck improves performance and at the same time guarantee that an optimized code has the same level of reliability than that of a non-optimized code that contains all the checks [1]. Identifying vital checks is not trivial because all checks in the baseline approach are important: each check is guarding a load, a store, or other synchronization instructions. Removing a check may result in a load returning a wrong value, memory corruption, a system crash, or other problems. Existing compiler optimizations can not help in this situation since these checks are not regular instructions and optimizing them requires a thorough understanding of the purpose of each check and its relationship with the underlying recovery layer. In this paper, we present three classes of optimizations that ESoftCheck can apply: The first optimization identifies

[1]Notice that although we target full coverage some errors cannot be detected when using software approaches. Details are given in Section VII.

the vital checks when the register file is hardware-protected with parity or ECC. Machines with hardware-protected register files include Intel Itanium [14], Sun UltraSPARC [9] and IBM Power4-6 [3]. In these platforms, checks are necessary to verify if the result of a computation is correct. However, contents of registers not involved in logic or arithmetic operations are verified by the hardware and not need to be software checked. This optimization is key for performance improvement in platforms with a limited number of registers (such as x86 where only 8 registers are available) because apart from removing redundant checks registers devoted to keep the replica can be deallocated earlier, reducing register pressure. The other two optimization provided by ESoftCheck can be applied even when the register file is not hardware protected. The first one identifies the vital checks when there are redundant checks. A check of a variable, say `v`, is redundant if it is postdominated by another nearby check of `v` or of a variable whose value is a function of `v` through a data dependence relation; the second optimization hoists out of loop checks of loop-invariant or induction variables.

As a flexible software-based approach, ESoftCheck provides knobs so that the user can trade reliability for performance. With ESoftCheck the user can i) define what are the commit points, and ii) specify the degree of trustiness of each operation. The distance between commit points is important, because the larger the number of instructions between commit points, the more redundant checks can be detected and removed. The degree of trustiness affects the removal of checks on variables that are data dependent through trusted operations.

ESoftCheck compiler techniques are beneficial for both software-only and hybrid (software/hardware) fault tolerant solutions. Software-only solutions applied to single-threaded code [26], [5] benefit from a lower register pressure and a reduction in the number of comparison and branch instructions executed. For multithreaded code [33], ESoftCheck also reduces the number of communication and synchronization instructions executed. In the case of hybrid solutions [27], [23], ESoftCheck reduces the number of hardware checks.

We have implemented ESoftCheck using the LLVM Compiler Infrastructure [10] and run experiments on a Pentium 4 using Spec benchmarks. Our performance results show that ESoftCheck compiler optimizations improve performance by 27.1% compared to a state of the art software-only single thread approach such as SWIFT [26]. As expected, our fault injection experiments using PIN [11] show that ESoftCheck does not produce more Silent Data Corruption than the fully replicated code.

The paper is organized as follows: Section II presents the background, Section III presents an overview of ESoftCheck, Section IV discusses in detail the compiler algorithms that ESoftCheck uses, Section V presents our environmental setup and experimental results, Section VI presents related work, Section VII present some issues that appear in software checking approaches, and finally Section VIII concludes the paper.

II. Background

Our baseline approach is similar to other software techniques for fault tolerance such as SWIFT [26] that assume that data can be corrupted in arbitrary ways within the CPU but that memory and caches are error-free; i.e., that the protection offered by techniques such as ECC and memory scrubbing reduce the frequency of undetected errors to an acceptable level. The software techniques are only reponsible of detecting CPU errors and preventing a faulty value from being written to memory. The base approach for doing so is to keep two copies of each register value, and to execute each operation twice, on different copies of the data; errors are detected by comparing the two copies. Stores, branches, function calls, returns, and loads are considered to be “synchronization” instructions where we need to ensure that certain values are correct; checking instructions are inserted before each synchronization instructions:

- Before a store, checking instructions verify the value and memory address. This ensures that the correct data is stored to the correct memory location.
- After a branch, checking instructions verify that the branch takes the appropriate path.
- Before a function call, checking instructions verify the input parameters.
- Before a function return, checking instructions verify the return value.
- Before a load, checking instructions verify the address of the load. Then, the loaded value is immediately copied to another register [5], [26]. An naive approach to obtain two independent copies and avoid the checking instructions is to replicate the load. However, this approach is not used because two consecutive loads to the same address may not return the same value if the addressed variable is volatile, or is a shared variable in a multithreaded program.

Strictly speaking, software techniques do not prevent faulty values from propagating to memory but they reduce the window of vulnerability where a soft error can cause a faulty value to be written to memory. A detailed description of errors that software techniques cannot handle is done in Section VII.

An example of the original and its corresponding augmented code executing in the same thread is shown in Figure 1-(a) and (b), respectively. The augmented code contains additional instructions that are shown in bold and uses additional registers that are marked with a `'`. Instruction 1 and 5 replicate the additions, instructions 2 and 3 check that the `load` is loading from the correct address, instruction 4 copies the value just loaded in `r3` and instruction 6-9 check that the store writes the correct data to the correct memory address.

III. ESoftCheck

In this Section we present an overview of ESoftCheck. Section III-A discusses the type of checks that ESoftCheck can optimize, Section III-B describes the knobs provided by ESoftCheck, and Section III-C discusses some of the issues that appear.

```
add r6 = r7, 4        add r6 = r7, 4             add r6 = r7, 4             add r6 = r7, 4
                      add r6'=r7', 4   (1)       add r6'=r7', 4  (1)        add r6'=r7', 4  (1)
                      cmp r6, r6'      (2)       cmp r6, r6'     (2)
                      jne faultDet     (3)       jne faultDet    (3)
ld r3=[r6]            ld r3=[r6]                 ld r3=[r6]                 ld r3=[r6]
                      mov r3'=r3       (4)                                  mov r3'=r3      (4)
....                  ....                       ....                       ....
add r4= r3,1          add r4= r3,1               add r4= r3,1               add r4= r3,1
                      add r4'=r3',1    (5)       add r4'=r3,1    (5)        add r4'=r3',1   (5)
                      cmp r4, r4'      (6)       cmp r4, r4'     (6)        cmp r4, r4'     (6)
                      jne faultDet     (7)       jne faultDet    (7)        jne faultDet    (7)
                      cmp r6, r6'      (8)                                  cmp r6, r6'     (8)
                      jne faultDet     (9)                                  jne faultDet    (9)
store [r6]=r4         store [r6]=r4              store [r6]=r4              store [r6]=r4

(a) Original code     (b) Baseline               (c) ESoftCheck             (d) ESoftCheck
                      replicated code              safe registers           non−safe registers
```

Fig. 1. ESoftCheck optimizations, when register file is safe-(c) and when register file is not safe-(d).

A. Optimized Checks

The added instructions in the augmented code of Figure 1-(b) can be classified as either shadow copies of the original instructions (instructions 1, 4 and 5) or error checking instructions (instructions 2, 3 and 6-9). In this baseline approach, checking instructions are inserted before synchronization instructions. The key idea of ESoftCheck is to identify, of all these checks, those that are vital – meaning, the ones that need to be kept so that any error can be detected and the program can roll back to a correct program state. To do that ESoftCheck uses compiler techniques to optimize four types of checks. Next, we describe them (a more formal description is presented in Section IV):

Type 1. Checks when the register file is hardware-protected. The register file of current platforms such as Intel Itanium [14], Sun UltraSPARC [9] and IBM Power4-6 [3] are already hardware-protected by parity or ECC or can be protected with cost-effective hardware mechanism [8], [17]. In these platforms, that we call *register safe platforms*, the hardware will detect faulty data in the registers; however, the hardware cannot determine that a register contains a faulty data as a result of a faulty computation storing the wrong result. Thus, it is still responsibility of the software to check that all the computations return the correct data.

In the register safe platforms, vital checks are only those that check the result of the computation. Registers not involved in arithmetic of logic operations do not need to be checked. Next we show the two cases that ESoftCheck optimizes:

Case a: Checks of registers defined by loads that have not been modified by arithmetic or logic operations are non-vital, and can be removed. In addition, the registers defined by loads do not need to be replicated, saving a copy instruction and reducing register pressure. An example is shown in Figure 2. Figure 2-(a) shows the original code, a pointer chasing code, where the data loaded from memory is used as the address of the next load. ESoftCheck removes instructions 3, 4, 5, and 6 because errors in register `r1` and `r3` are detected by the hardware.

```
                   cmp r2, r2'     (1)      cmp r2, r2'     (1)
                   jne faultDet    (2)      jne faultDet    (2)
ld r1=[r2]         ld r1=[r2]               ld r1=[r2]
                   mov r1'= r1     (3)
                   cmp r1, r1'     (4)
                   jne faultDet    (5)
ld r3=[r1]         ld r3=[r1]               ld r3=[r1]
                   mov r3'= r3     (6)

(a) Original code  (b) Replicated code      (c) ESoftCheck
```

Fig. 2. Check removal when register file is safe.

Case b: Checks of registers holding the result of an arithmetic or logic computations are vital and need to be kept. In this case, there are two situations. In the first situation a check appears right after the computation because of a synchronization operation right after the computation. An example is shown in Figure 1-(c). In this example, `r6` is the result of an `add` instruction and used right after by a `load`. The check before the load (instructions 2 and 3) verifies that the result of the addition is correct; thus, it is vital and needs to be kept. The second check (instruction 8 ad 9) can be removed as `r6` is not modified between the `load` and the `store`. Notice that after the first check register `r6'` can be deallocated, reducing register pressure.

The second situation appears when the synchronization instruction is far from the computation on a register. In this case, rather than keeping the check at the place of the synchronization instruction, ESoftCheck moves the check to appear right after the computation. This does not reduce the number of checks, but reduces register pressure as the replicated register is deallocated after the check. An example where this situation appears is shown in Figure 3, which is extracted from `gzip`. The code is in SSA form (every variable is assigned only once) and has a ϕ(Phi)-function [35] to specify that register `r3` will take the value in `r2` if the control flow comes from block `L1` or the value in `r5` if the control flow comes from block

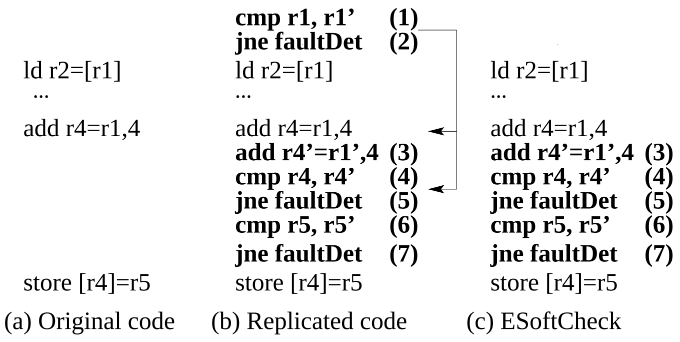
`L7`. In the baseline replicated version (Figure 3-(b)), the check of `r3` (instructions 3 and 4) before the `load` checks register `r2` or `r5`, depending on where the control flow comes from. However, since the check is not in the same basic block where `r2` and `r5` are defined, the replicated register `r2′` and `r5′` need to be kept until register `r3` is checked in the basic block `L3`. ESoftCheck can optimize this code as shown in Figure 3-(c), where `r2` and `r5` are checked right after being defined and before the ϕ(Phi)-function. This optimization results in more static instructions and one extra dynamic check, but reduces register pressure because `r2′` and `r5′` are deallocated after the check.

Notice that for the optimized code to have the same level of reliability than that of the non-optimized code we are assuming that when the register file is hardware-protected, the path to and from the register file, forward paths, and renaming tables are also hardware-protected. If this is not the case, an error in a datapath that would have been detected in the non-optimized code may not be detected in the optimized one. As an example consider the instruction `add r4 = r3 ,1` and its replica (instruction 5) in the optimized code in Figure 1-(c). If the value of register `r3` is corrupted in a datapath, and both instructions receive the same corrupted value, the error cannot be detected. However, if each instruction receives the value from two different datapaths, for instance, one receives it from the forwarding path and the other reads it from the register file, and one value is corrupted and the other not, the error can still be detected. In the non-optimized code in Figuree 1-(b) an error in the datapath will be detected if it occurs after register `r3` has been replicated by the `mov r3′=r3` instruction because in the non-optimized code the original instruction and its replica use different registers (`r3` and `r3′`), so an error in one datapath should only affect one of the instructions.

Type 2. Checks covered by a later check to the same register. A check of a register is redundant if it is always followed (covered) by another check of the same value in the register at the time of the check, and the register has not been modified in between the two checks. In such a case, the last check is the vital one when the register is not hardware-protected. The first check (the redundant one) can be removed since an error will be discovered by the subsequent check (the vital one). The example in Figure 1-(d) shows the optimized code of Figure 1-(b). Instructions 8 and 9 check the same register as instructions 2 and 3, and the value of `r6` does not change between the two checks. Therefore, instructions 2 and 3 can be removed.

Notice that by removing the instructions 2 and 3 in Figure 1-(c), it is possible to load from a wrong address (which will result in a wrong value in both `r3` and `r3′`) or cause a segmentation fault. The error in the load address will be detected when checking `r6` in instructions 8 and 9. Section III-C explains how to handle segmentation faults resulting from transient errors.

Type 3. Checks covered by a later check of a different register. A check of a register is redundant when it is followed (covered) by a check of a different register whose value is a function of the first register. In this case the second check is the vital, and the firt one (the redundant check) can be removed. An example is shown in Figure 4. Figure 4-(a) shows the original code, and Figure 4-(b) shows the replicated code. In this example, it is possible to remove instructions 1 and 2 that check register `r1`, because `r4` is computed by adding a constant to `r1`. An error in `r1` propagates to `r4`, and is detected when `r4` is checked.

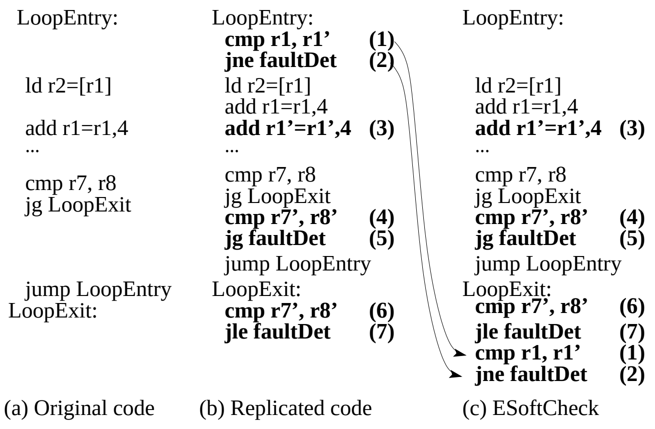


```
                    cmp r1, r1'     (1)
                    jne faultDet    (2)
ld r2=[r1]          ld r2=[r1]                  ld r2=[r1]
 ...                 ...                         ...
add r4=r1,4         add r4=r1,4                 add r4=r1,4
                    add r4'=r1',4   (3)         add r4'=r1',4  (3)
                    cmp r4, r4'     (4)         cmp r4, r4'    (4)
                    jne faultDet    (5)         jne faultDet   (5)
                    cmp r5, r5'     (6)         cmp r5, r5'    (6)
                    jne faultDet    (7)         jne faultDet   (7)
store [r4]=r5       store [r4]=r5               store [r4]=r5
(a) Original code   (b) Replicated code         (c) ESoftCheck
```

Fig. 4. ESoftCheck uses data dependence to remove redundant checks of different registers.

Type 4. Checks of loop induction variables and loop invariants. Checks inside loops can be made redundant and removed by adding covering checks at the loop exits, thus reducing the dynamic check count. Figure 5-(a) shows a loop where register `r1` contains an induction variable. The corresponding replicated code is shown in Figure 5-(b) where checks at the taken and fall through paths of the conditional branch (instructions 4, 5, 6, and 7) verify that the loop executes the correct number of iterations.

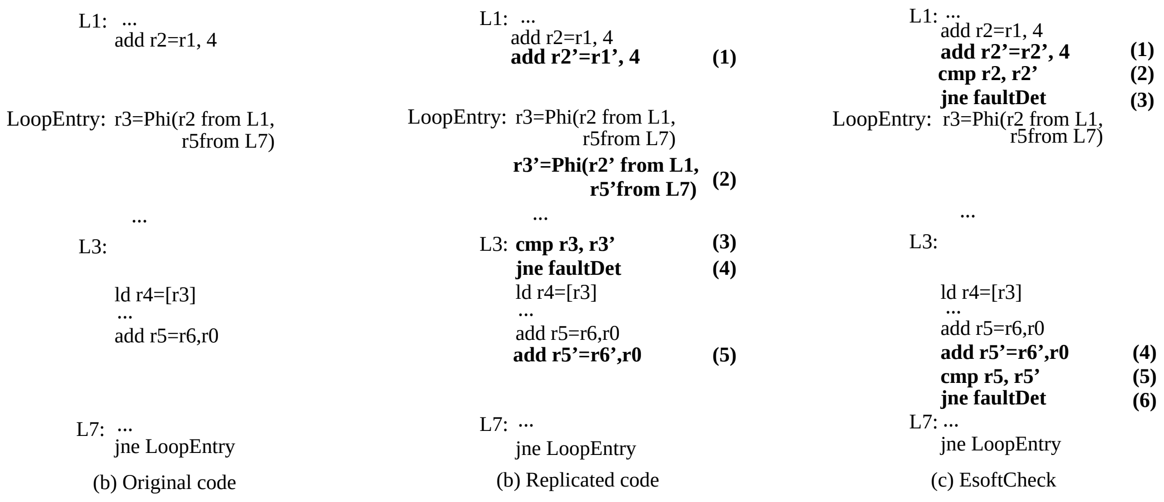
Figure 5-(c) shows the ESoftCheck code, where instructions 1 and 2 that check register `r1` have been moved outside the loop because the compiler has determined that `r1` contains an induction variable, and any error in the loop will also propagate outside, where it will be detected.



```
LoopEntry:          LoopEntry:                   LoopEntry:
                      cmp r1, r1'       (1)
                      jne faultDet      (2)
 ld r2=[r1]           ld r2=[r1]                   ld r2=[r1]
                      add r1=r1,4                  add r1=r1,4
 add r1=r1,4          add r1'=r1',4     (3)        add r1'=r1',4   (3)
 ...                  ...                          ...
 cmp r7, r8           cmp r7, r8                   cmp r7, r8
 jg LoopExit          jg LoopExit                  jg LoopExit
                      cmp r7', r8'      (4)        cmp r7', r8'    (4)
                      jg faultDet       (5)        jg faultDet     (5)
                      jump LoopEntry               jump LoopEntry
 jump LoopEntry     LoopExit:                    LoopExit:
LoopExit:             cmp r7', r8'      (6)        cmp r7', r8'    (6)
                      jle faultDet      (7)        jle faultDet    (7)
                                                   cmp r1, r1'     (1)
                                                   jne faultDet    (2)
(a) Original code   (b) Replicated code          (c) ESoftCheck
```

Fig. 5. Check removal for an induction variable.

Notice that optimizations of checks of types 2, 3, and 4 can



```
L1:  ...
     add r2=r1, 4


LoopEntry: r3=Phi(r2 from L1,
                    r5from L7)


      ...
L3:


     ld r4=[r3]
      ...
     add r5=r6,r0


L7:  ...
     jne LoopEntry
```

(b) Original code

```
L1:  ...
     add r2=r1, 4
     add r2'=r1', 4               (1)


LoopEntry: r3=Phi(r2 from L1,
                    r5from L7)
     r3'=Phi(r2' from L1,
                  r5'from L7)     (2)
      ...
L3:  cmp r3, r3'                  (3)
     jne faultDet                 (4)
     ld r4=[r3]
      ...
     add r5=r6,r0
     add r5'=r6',r0               (5)


L7:  ...
     jne LoopEntry
```

(b) Replicated code

```
L1:  ...
     add r2=r1, 4
     add r2'=r2', 4               (1)
     cmp r2, r2'                  (2)
     jne faultDet                 (3)
LoopEntry: r3=Phi(r2 from L1,
                    r5from L7)


      ...
L3:


     ld r4=[r3]
      ...
     add r5=r6,r0
     add r5'=r6',r0               (4)
     cmp r5, r5'                  (5)
     jne faultDet                 (6)
L7:  ...
     jne LoopEntry
```

(c) EsoftCheck

Fig. 3. Register safe optimizations when the check is far from the computation.

be applied when the register file is not safe. If the register file is safe, checks of type 1 are optimized first and the other types of checks are optimized afterwards. More details are given in Section IV. Finally, notice that to simplify the discussion our examples only contain two checks and simple data dependences, but ESoftCheck can detect more than one redundant check linked by a chain of data dependences.

B. Knobs

ESoftCheck provides two types of knobs so that the user can trade reliability for performance. The user can trade the number of redundant checks that can be removed based on the frequency of checkpoints and the trustiness of operations.

1) Checkpoints: A fault tolerant system requires a checkpoint mechanism that saves snapshots of the application state where to roll back if an error is detected. The *commit points* are the instructions in the application where a new checkpoint is taken so that the space used by the previous checkpoint can be released. To be able to properly recover, a checkpoint must not contain corrupted data. Thus, at commit point we need to make sure that any possible error has been detected. As a result, in order to determine which are the vital checks, the optimization techniques for checks of types 2, 3, and 4 need to know the location of the commit points. For example, given two checks to the same register the first check is a vital check (cannot be removed) if there is a commit point between the two checks. An example illustrating this situation is shown in Figure 6-(a) and (b), where the check operator consists of two instructions: a "comparison" instruction to compare the contents of the register in the original code with the contents of its replica and a "conditional branch" to an error handler if a mismatch is detected. Similarly, when a commit point is inside a loop, checks of loop induction variables and of loop invariants cannot be moved outside the loop. The optimization of checks of type 1 is not affected by the location of the commit points, as after a check verifies that the computation is correct it is the hardware responsibility to detect errors in the registers.

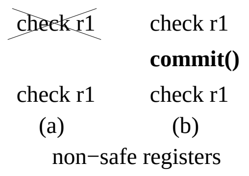


Fig. 6. ESoftcheck in the presence of checkpoints.

ESoftCheck provides knobs so that the user can specify the location of the commit points. Notice that in general the more instructions between commit points, the more likely it will be for ESoftCheck to find redundant checks that can be removed, resulting in a larger reduction of the overheads. For this paper we evaluate two different checkpoint frequencies, described in Section IV-D.

2) Degree of trustiness: As explained when optimizing checks of type 3, a check is non-vital (can be removed) when it operates on a register whose value is a function of an earlier checked register. However, this approach can mask some errors. For instance, if we have `mul r4=r1,r3` and `r3` is zero, by only checking `r4` we will not detect if there is an error in `r1`. Thus, we define *Trusted* operators as those that have a low chance of masking errors. ESoftCheck can provide knobs so that the user can specify which operators are to be considered *trusted* so that variable `a` can be checked through variable `b`, when variable `b` depends on `a` through a chain of dependences that only involves *trusted* operators. In general, arithmetic and shift operators are considered trusted. For logic operators the probability of error propagation will depend on the number of 0's and 1's.

An interesting situation appears with the conditional move operator: `cmov r4, r3, cond` copies register `r3` to `r4` if `cond` is true; otherwise it does not do anything. We consider that it is not safe to check register `r3` by checking the contents of `r4`, because when `cmov` does not perform the copy, an error in `r3` will not be detected by checking `r4`. On the other side, since `cond` is computed as the result of a comparison instruction that executes before the `cmov`, it would be possible to check the operand registers of the comparison through a check of `r4`. However, since the comparison only has two possible outcomes, an error in the comparison operand register has a high probability of being masked. Thus, we consider the `cmov` operator not trusted.

C. Issues

ESoftCheck can increase the number of segmentation faults with respect to the Baseline Fully Replicated codes. The reason is that by removing redundant checks it is possible that some errors will manifest as segmentation faults before the error is detected, that is, before the check that has not been removed is executed (an example was shown in Figure 1-(d), where by removing instructions 2 and 3, a segmentation fault could occur during the load if the contents of `r6` were corrupted). In these circumstances we will not know if the segmentation fault is the result of a programming error or of a soft error. However, since the operating system knows where the segmentation fault occurred and a fault tolerant system must have a mechanism to roll-back to a safe state, we can roll-back and re-execute. If the same error appears the operating system will notify of an error to the user; however, if the error does not appear again, we can consider it was due to a soft error or a heisen software bug. Notice that the number of segmentation faults will not increase when optimizing checks of type 1 on a register safe platform.

This increase of segmentation faults can hardly affect performance. Our experimental results in Section V show that only a very small fraction of the injected errors caused segmentation faults. Thus, the overhead of roll-back and re-execution on those rare situations where a soft error manifests as a segmentation fault pays off for all the executions without the extra checks.

IV. Framework

Our optimizations are implemented as passes on the LLVM intermediate level [10], which is a SSA representation [6]. We present the optimization algorithms for hardware-protected checks (checks of type 1) in Section IV-A, for checks covered by the same or different register in Section IV-B, and for checks before loop induction variables and loop invariants (checks of type 4) in Section IV-C. When the register file is hardware-protected, the algorithm in Section IV-A is applied first.

A. Hardware-Protected Checks

As explained in Section III-A, on register safe platforms there are two cases when checks can be optimized. We unify the two cases by defining that ***register r is safe at point P*** if `r` is defined by a load or there is a check on `r` at point Q, such that Q dominates P and `r` does not change on any path between point Q and point P. If we know register `r` is safe at point P, a check on `r` (at P) can be removed, and the use of `r'` (at P) can be replaced by `r`. The problem of determining which registers are safe at a given point is a forward data-flow problem.

However, there is a situation shown in Figure 3 where the check is far from the block where the register is defined. ESoftCheck implements a second compiler pass that moves the check of a register close to its definition when the distance between the register definition and the check is larger than a certain threshold. Notice that the identification of vital checks in this case does not depend on the location of the commit points (See Section III-B1).

B. Covered Checks

A check `c1` of register `r1` is *covered* by another check `c2` of register `r2` when
1. `c2` postdominates `c1`.
2. Either `r2=r1` or `r2` depends on `r1` through a chain of data dependences that only involve *trusted* operators.
3. There is no update to register `r1` on any path between the two checks.

If `r2=r1` we say `c1` is directly covered by `c2`; otherwise we say it is indirectly covered. A covered check `c1` can be eliminated if there is no commit point on any path between `c1` and `c2`. Examples are shown in Figure 7. On Figure 7-(a) and (b) `c1` is covered by `c2`, but on (b) `c1` cannot be removed because of the commit instruction. On Figure 7-(c), `c1` is not covered because `c2` does not postdominate. On Figure 7-(d), `c1` is covered because the combination of `c2` and `c3` postdominates `c1`.

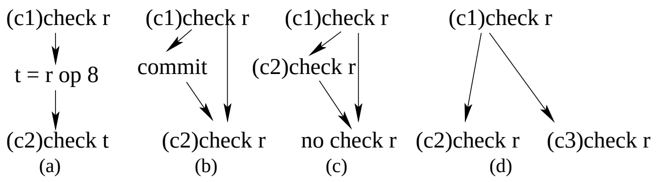


Fig. 7. Examples of covered and non-covered checks. In (a) and (b) check *c1* is covered by *c2*, but check *c1* in (b) cannot be removed because of the commit instruction. In (c), check *c1* is not covered by *c2*. In (d), check *c1* is covered by *c2* and *c3*.

Next, we present the algorithm to detect covered checks that can be removed (Section IV-B1) and then discuss how to apply it based on the support for checkpoint and rollback (Section IV-D).

1) Algorithm To Remove Covered Checks: The algorithm to detect and remove Covered checks is similar but different from classic Common Sub-Expression Elimination(CSE) [18], in that: i) We can optimize the data-dependence case where two checks are checking different registers, while CSE can not optimize if two expressions have different operands. ii) We need to preserve the latest check for catching an error, while CSE preserves the earliest evaluation of an expression. iii) A commit point will kill all available checks in our case, while CSE does not have such a powerful killer.

We define that `check(r)` is *available* at point P if on every path from the program end to P, there is a `check(t)` at point

P' (`t` depends on `r` through a chain of trusted operators, or `t` is `r`), and there is no update to `r` and no commit point in between. To determine if a `check(r)` can be removed ESoftCheck will determine if `check(r)` is available right after it appears in program order using the algorithm described next.

Flow analysis of available checks. The flow analysis will be discussed in two parts. First, we present the intra-basic block (local) analysis, and then the global flow analysis.

Local flow analysis. Let $AC_{AF}(I)$ and $AC_{BF}(I)$ be the set of Available Checks right after and right before instruction I (in program order), respectively. Let $Chk\_Ins()$ be the instruction transfer function which computes AC_{BF} in terms of I and AC_{AF}: $AC_{BF}(I) = Chk\_Ins(I, AC_{AF}(I))$. $Chk\_Ins()$ is defined as follows:

• If I is `check(r)`, $AC_{BF}(I) = AC_{AF}(I) \cup \{check(r)\}$

• If I is `r=OP(t)` and OP is a trusted operator, we will make `check(t)` available if `check(r)` is already available right after I. The reason is that an error in `t` will be determined by `check(r)`. In addition, since `r` is updated we kill the availability of `check(r)`. With this we propagate available checks through chains of data dependence.

$$AC_{BF}(I) = \begin{cases} (AC_{AF}(I) \cup check(t)) - \{check(r)\}\,, \\ if check(r) \in AC_{AF}(I) \\ AC_{AF}(I) - check(r), otherwise \end{cases}$$

• If I is `r=OP(t)` and OP is not a trusted operator, we kill the availability of check(r). $AC_{BF}(I) = AC_{AF}(I) - \{check(r)\}$

• If I is `commit point`, all the available checks are killed, as they cannot propagate across commit points. Thus, $AC_{BF}(I) = \emptyset$

• Otherwise, $AC_{BF}(I) = AC_{AF}(I)$

Global flow analysis.

Let $AC_{IN}(B)$ and $AC_{OUT}(B)$ be the set of available checks on entry to and exit of basic block B, respectively. Let $Chk\_Blk()$ be the basic block transfer function: $AC_{IN}(B) = Chk\_Blk(B, AC_{OUT}(B))$.

Assuming that the basic block B contains the instruction sequence I_1, I_2, ... I_n we define Chk_Blk as $AC_{IN}(B) = AC_{BF}(I_1) = Chk\_Ins(I_1, Chk\_Ins(I_2, \ldots Chk\_Ins(I_n, AC_{OUT}(B)) \ldots))$.

Figure 8 shows an example. For instance, at instruction 3 (`r3=r1+16`), `check (r3)` is killed. In addition, `check(r1)` is made available because `r3` depends on `r1` through a trusted operator and `check(r3)` is available right after instruction 3. Thus, when instruction 1 is processed, we find that `check(r1)` is available right after it. As a result `check(r1)` at instruction 1 can be removed.

To guarantee the postdomination property a check is available at the exit of a basic block only if the check is available on the entries of all the successor basic blocks. Then, the data flow equations are:

(a) $AC_{OUT}(B) = \bigcap AC_{IN}(S)$, over all successors S of B in the data flow graph.

and

(b) $AC_{IN}(S) = Chk\_Blk(S, AC_{OUT}(S))$.

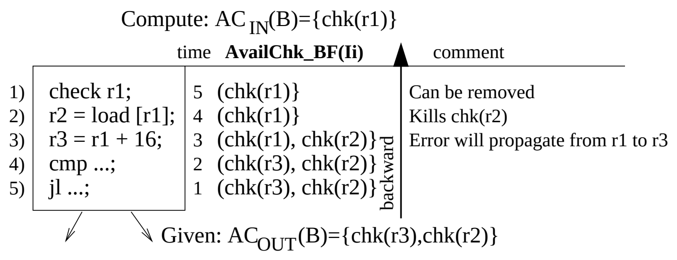


Fig. 8. Available checks for a basic block

Notice that the check removal can be applied while the available check analysis is being applied.

C. Loop Checks

Our algorithm detects loop induction variables and loop invariants and move the checks of these variables outside the loop. Induction variables are variables whose successive values form an arithmetic progression in a loop. In the SSA form, loop induction variables are defined by cycles involving ϕ(Phi)-functions [35]. We scan ϕ(Phi)-functions in the loop header node. Given $r = \phi((pre-header, r0), (backedge, r2))$, if $r2$ is defined as r plus(or minus) a loop constant, we consider `r` as a loop induction variable. This optimization is only applied to loops that do not contain commit points in our implementation.

D. Knobs

Checkpoints: ESoftCheck determines the covered checks that can be eliminated based on the location of the commit points. As explained in Section III-B1 checks cannot be delayed across commit points. In this paper, we evaluate two checkpoint frequencies:

1) **MemUnPolluted**. To compare with previous proposals, we follow their approach: a program is considered correct if its output is correct (assuming memory-mapped I/O), that is, if all the stores have executed correctly [23], [26], [27], [28]. Under this approach stores, function calls and function returns are commit points and checks cannot be delayed across commit points. As a result, checks can only be removed before loads.
2) **MemCheckpoint**. Under this system commit points are function calls and function returns. With MemCheckPoint stores are not commit points and memory can be corrupted with wrong values. Thus, a mechanism for memory checkpointing either in software [4] or in hardware, such as ReVive [24] or SafetyNet [31] is necessary.

Degree of Trustiness: With ESoftCheck the user can specify which are the trusted operators. For the experiments in Section V arithmetic, shift and logic operators are trusted operators, while the conditional move operators are not. Due to space limitations, we do not evaluate the impact on performance or reliability of the degree of trustiness.

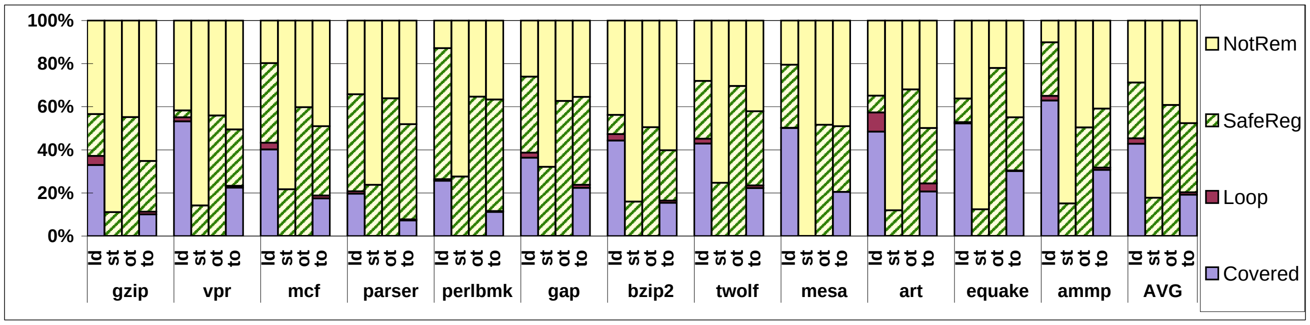


Fig. 9. Characterization of static checks for each type of instruction(MemUnPolluted).

V. Evaluation

In this Section we evaluate our proposed techniques. We first discuss our environmental setup (Section V-A), characterize the number of static checks that can be removed (Section V-B), evaluate performance (Section V-C), and measure reliability (Section V-D).

A. Environmental Setup

We use LLVM [10] as our compiler infrastructure to generate single-threaded codes extended with redundant computations and the check operations. This extension is carried out at the intermediate level, right after all the static optimizations have been done. We replicate all the integer and floating point instructions. To prevent backend optimizations to eliminate the added code we tag the replicated instructions. The backend optimizations are applied separately to the tagged and the untagged instructions. For the evaluation reported here 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.

B. Characterization of Static Checks

In this Section, we characterize the static checks that can be removed. A breakdown is shown in Figure 9. There are four bars for each application. The first three bars characterize the checks based on the type of instruction they guard: load *ld*, store *st*, and function call and return *other*. The last bar *to* corresponds to the sum of all the checks in the three previous bars. The bars are normalized to the total number of checks for each type of instruction. A check is categorized according to the reason why it can be removed: (i) because it is covered by another check to the same or different register (*Covered*), (ii) because it is before a loop induction variable or loop invariant (*Loop*), (iii) because the register file is safe (*SafeReg*). The checks that cannot be removed appear as (*NotRem*). For the characterization we assume the *MemUnPolluted* model described in Section IV-D

Notice that a given check may belong to Covered and RegSafe at the same time. However, in our characterization this check will appears as Covered. The first observation we make is: (1) with the *MemUnPolluted* model only the covered checks before loads can be removed. The removed checks in this case account for 42.8% of the checks before loads and 19.1% of the total number of checks; (2) the fraction of checks that guard loop induction variables and loop invariants is very small; (3) when the register file is safe, an average of 32.1% of the checks can be removed.

C. Performance

Figure 10 shows the performance of different systems where fault tolerance has been implemented in software. Fully Replicated (*FullRep*) corresponds to the baseline system described in Section II and that is similar to SWIFT [26]. Notice that in our implementation of *FullRep* checking instructions are inserted before branches to verify that the branches follow the appropriate path, but checks to verify that the program follows a legal control path and that the PC has not been corrupted have not been inserted, as discussed in Section VII. The rest of the bars in Figure 10 correspond to the different optimizations described in Section IV: removal of checks because the register file is safe (*SafeReg*), removal of checks that are covered by another check to the same or different register (*Covered*), and removal of checks before loop induction variables and invariants (*Loop*). The first four bars of each application correspond to the *MemUnPolluted* model where stores, function calls and function returns are considered commit points. In addition, we also show performance numbers for the MemCheckPoint (*MemChkpt*) model, where commit points are function calls and function returns.

FullRep is on the average 2.16 times as slow as the original code. This large overhead is due to several reasons: i) register pressure, the replicated code needs twice as many registers as the original application, and the x86 ISA only has 8 registers available to the compiler, and ii) the additional instructions. Previous works have published smaller overheads for *FullRep* [5], [26], [27] but in that work the target machines were Itanium or PowerPC platforms that have a larger number of registers.

SafeReg is the optimization that obtains the highest performance benefit. The reason is that apart from the redundant checks removed, it reduces register pressure, which is very important in x86 processors, where only 8 registers are available to the compiler. On the average, *SafeReg* runs 24.6% faster than *FullRep*. Notice that *SafeReg* does not include the second

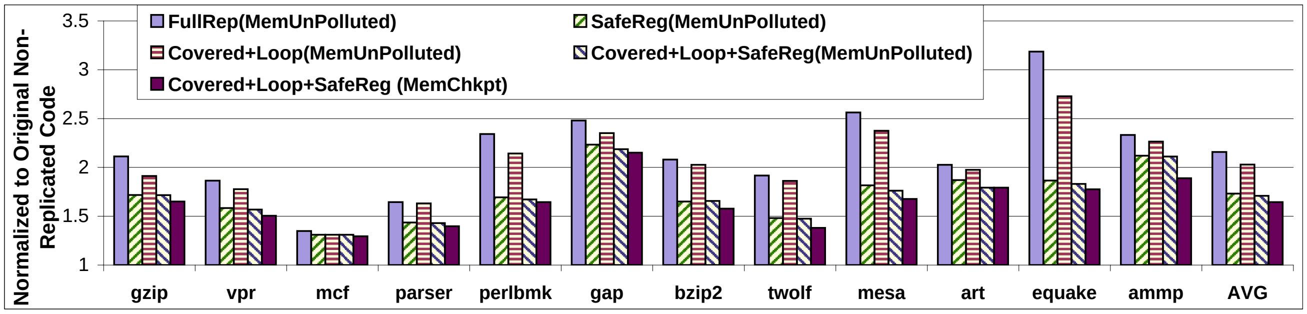


Fig. 10. Performance of the different optimizations normalized against the original non-replicated code.

compiler pass described in Section IV-A where a register is moved close to its definition. Thus, our results do not show the benefit of that optimization, although we do not expect it to have a large impact. Some preliminary experiments that we did show that the number of places where it can be applied is small.

The removal of redundant checks (*Covered*) runs 6.9% faster than *FullRep*, but only checks before loads can be removed under the *MemUnPolluted* model. We have also removed all the checks before loads (not shown in the Figure), and found out that the average normalized execution time is 1.82 (versus 2.02 of *Covered*). Thus, *Covered* reduces 41.2% of the overhead introduced by checking the load address. The optimization before induction variables and loop invariants (*Loop*) results in little performance gain (that is why we do not show results for only *Loop*). The reason is that as shown in Figure 9, the fraction of checks that can be removed is small.

When we combine the three optimizations together *Covered+Loop+RegSafe* we apply the *SafeReg* optimization first and obtain an average 27.1% speedup compared to *FullRep*, resulting in a code that runs 1.70 times as slow as the original code. Under the *MemCheckPoint* model (last bar in Figure 10) where checks before store can also be removed, the combination of all optimizations achieves on average 31.7% speedup compared to *FullRep*.

Notice that *FullRep* corresponds to state of the art approaches such as SWIFT [26] that use only software checking and no special hardware for error detection. Under the *MemUnPolluted* model, when applying all our optimizations *Covered+Loop+SafeReg* the overhead is reduced from 116% to 70%.

D. Reliability

To evaluate the reliability of our optimizations we use Pin [11] and inject faults to the binary file (excluding system libraries) assuming a Single Event Upset (SEU) fault model, that is, a single bit is flipped during the whole execution of the program. Although our detection mechanism will very likely detect multiple bit faults, the probability of multiple faults is much lower than SEU.

Notice that to accurately model soft errors, one should use a HDL simulator and inject faults to buses, latches, combinational logic, and SRAM cells, among others. If this is done, many injected faults would be masked and a few would manifest as errors in the architectural status [29]. Here we report the result of injecting faults into the register and status flags. In effect, we are modeling only those errors that appear in the architectural status. We cannot inject faults that corrupt the program counter, so we cannot model that type of errors. However, notice we did not implement a mechanism to protect the control flow (as explained in Section II) because the target of our optimization techniques is not the program counter. Finally, notice that a similar fault injection mechanism has been previously used by other software checking approaches [5], [26], [33].

In our experiments a total of 2000 faults were injected into each program. When we assume that the register file is not protected in hardware, we mimic the fault distribution by randomly selecting a point in the execution sequence and flipping a random bit in a random register. When we assume that the register file is protected in hardware, we mimic the fault distribution by randomly selecting a dynamic instruction and randomly flipping a bit of its “output”. The output can be in a register or in memory if it has been spilled. Memory load instructions are avoided. We call the first scheme “random fault injection” and the second one “safe register fault injection”. Notice that in practice fault distribution is not uniform, but it is a first order approximation used by previous fault injection approaches [26], [27], [34].

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 program will be categorized as: *unACE*: the bit is unnecessary for Architectural Correct Execution [20]; *Detected*: the error is detected by our checking code; *Self-Detected*: the error is detected by the program assertions; *Seg Fault*: the error manifest 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 avoid *Self-Detected* errors and minimize *Seg Fault*, but these faults can be recovered, so they are less harmful.

Figure 11-(a) and (b) show the experimental results for random fault injection and safe register fault injection, respectively. On the random fault injection scheme (Figure 11-(a)), on the original program (O) on average 72% of the faults appear as *unACE*, 3% as *Self-Detected*, 19% result in *Seg Fault* and 6% are *SDC*. Under the safe register scheme (Figure 11-(b)) more faults result in *SDC* (9% over 6%). The reason is that the random scheme is more likely to select a dead register. It is also interesting to notice that gzip and bzip2 have a large fraction of *Self-Detected* errors (10% and 24%, respectively under the random injection scheme) because the program checks the data consistency after the data is compressed or decompressed.

As expected after the program is replicated (*Fr*), most *Seg Fault*, *Self-Detected* and *SDC* go to the *Detected* category. Also, many *unACE* errors in the original (*O*) program appear as *Detected* because they are now detected by the checks added. *Fr* has 4.7% and 1.1% of *Seg Fault* under the random register injection scheme and the safe register injection scheme, respectively. *SDC* errors appear under the random register injection scheme because some faults are injected before the value is used but after is checked. *SDC* errors do not appear under safe register injection scheme. After our optimization, ESoftCheck does not produce more *SDC* or *Self-Detected* errors than *Fr*. As for *Seg Fault*, under the random fault injection scheme, ESoftCheck generates slightly more than *Fr* (5.5% of *ESoftCheck* versus 4.7% of *Fr*). Under the safe register fault injection scheme, the *Seg Fault* for *ESoftCheck* is 3.0% versus 1.1% of *Fr*. Remember that *Seg Fault* is recoverable by rolling back and re-executing, so these numbers are acceptable.

VI. Related Work

Redundant execution followed by checks is widely used to detect soft errors, either in hardware [25], [19], [32], [7] or in software [26], [27], [23], [33]. Since checks consume time and even communication bandwidth (if the replicated instructions execute in separate threads), previous approaches have also tried to reduce the number of checks. Hardware multi-thread approaches that only detect errors, such as SRT [25] and CRT [19] check stores and uncached loads, assuming that an error will eventually propagate to stores or uncached loads or it will not affect the program result at all. However, when these techniques also consider error recovery [32], [7] the number of checks increases significantly because the trailing thread is used to recover. Thus, every instruction in the trailing thread needs to be checked before it commits. To reduce the number of checks, CRTR [7] proposes the use of “Dependence-based Checking Elimination”, that exploits register dependence chains, so that only the value of the last instruction in a chain is checked. Our optimization to remove the covered checks by a later check to a different register (second type of checks in Section III-A) exploits a similar idea. The difference is that the checks ESoftCheck tries to remove are before loads, stores or other synchronization instructions, so more considerations need to be taken into account when optimizing these checks.

Compared with software-based approaches where a thread checks itself such as [26], [27], [23], [33] ESoftCheck removes additional checks while maintaining the same level of reliability. While doing this, ESoftCheck takes into account the commit points so that recovery is not compromised by the removed checks. Also, even though ESoftCheck optimizations are evaluated using a single thread, they can also be applied to software-based approaches where a separate thread checks the original thread, such as [33]. In this case, since the number of checks is reduced, the synchronization and bandwidth requirements are also reduced.

Compiler techniques have been used in other approaches for fault tolerance. Meixner et al. [15] compute dataflow graphs at compiler time and use special hardware to verify the dataflow when the program is executing. Nakka et al. [21] select ”critical” variables through data dependence analysis, and use a hardware redundant thread to check the slices that contribute to these variables. These two approaches do not provide full fault coverage.

Finally, in [37] we presented a naive algorithm where a check could be removed if the register being checked was defined by a load and the register file was hardware-protected. In that naive approach register pressure was not reduced. In this paper we identify three more types of checks that can be removed (Section III-A).

VII. Issues with Software Checking Approaches

Software-based approaches like ours that replicate instructions can detect errors in the arithmetic and logic units, register file, and varied buses when the errors finally propagate to the resulting value of an instruction. Although they reduce the window of vulnerability to levels that we believe are acceptable for commodity processors that need to balance cost, performance and power consumption, they do not provide 100% fault coverage. Next we discuss the transient errors that our software-based approach cannot detect and outline possible solutions with special hardware support or more expensive software protection mechanisms:

- Errors occurring after the check and before the load/store executes cannot be detected. Similarly, although memory is error-free, it is possible for an error to occur in the path that brings the data from/to memory during a load or store execution. A simple hardware solution to this problem was proposed in [27].
- With the approach described in Section II branches are checked to verify that they follow the appropriate path, but the program can follow an illegal control path if the program counter is corrupted. Thus, additional testing for legal control flow, done either in software or in hardware, is necessary to ensure that the program counter is not corrupted [1], [12], [22].
- For function calls, our baseline replication checks the function arguments before the call. However, the argument could be corrupted in the middle of the transfer. A solution proposed in SWIFT [26] consists of passing two copies of each argument to the callee and checking the arguments right after entering the callee function.

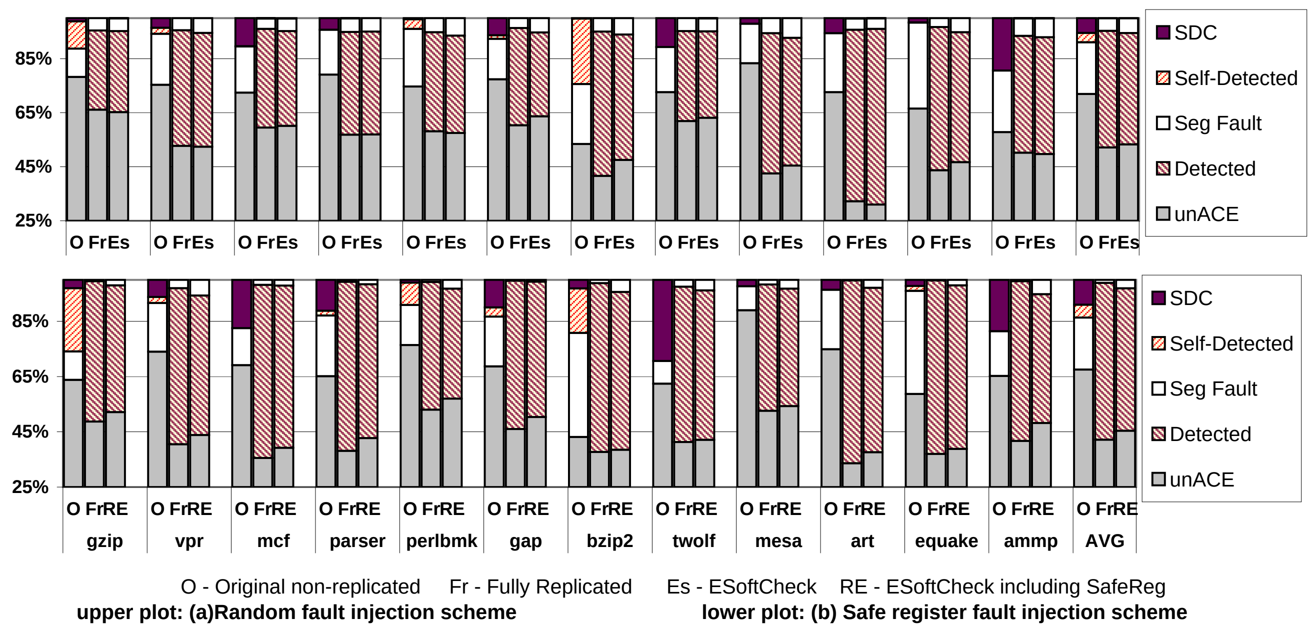


Fig. 11. Fault-detection rates break down (MemUnPolluted)

• The software-based instruction level replication can detect most transient errors of instruction opcode but, if an error changes a nonstore opcode to a store opcode or a store opcode to a null opcode, the error may not be detected. A hardware Store Value Queue proposed in [27] can solve this problem.

• If an error causes an exception that should not occur in correct runs, there are two cases: i) if the exception handler finally returns, the error may be detected by the software inserted checkings; ii) if the exception handler does not return, the error will not be detected unless we also add checking instructions to the exception handler.

• When the software-based checking mechanism is implemented at source or intermediate level, libraries can only be protected if the source code is available. If not, two solutions are possible: if the library call does not make external changes, we can treat this library call as a huge instruction and replicate this instruction. Alternatively, the library can be protected using binary level instrumentation [28].

• An error in the micro-architecture may manifest in multi-bit error or multiple errors in the architectural variables. Though our fault model aims at single error upset (SEU), it is still likely that multiple errors will generate unmatched pairs of variables and thus be detected by the software checkings.

VIII. Conclusions

In this paper we have presented ESoftCheck, a set of compiler techniques that identify vital checks and reduce the overheads of software approaches for fault tolerance. To the best of our knowledge ESoftCheck is the first work that identifies the checks before loads, stores and synchronization instructions that can be removed without sacrificing fault coverage. ESoftCheck also takes into account the location of the commit points where checkpoints are taken, so that recovery is not compromised. Our results show that when the register file is hardware-protected ESoftCheck can not only remove many checks but also deallocate replicated registers, reducing register pressure significantly. As a result, in a system that we call MemUnPolluted (where stores execute correctly and memory is not corrupted with wrong results) our techniques reduce execution time by 27.1% over previous state of the art approaches (overheads are reduced from 116% to 70%). Our techniques are useful for software-only systems and hybrid solutions, both single and multithreaded approaches.

Acknowledgment

This work was supported by the National Science Foundation under the CSR-AES awards 0615273 and 0509432.

References

[1] Z. Alkhalifa, V. S. S. Nair, N. Krishnamurthy, and J. A. Abraham. Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection. *IEEE Trans. Parallel Distrib. Syst.*, 10(6):627–641, 1999.

[2] R. Baumann. Soft Errors in Commercial Semiconductor Technology: Overview and Scaling Trends. *IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals*, pages 121_01.1–121_01.14, April 2002.

[3] D. Bossen, J. Tendler, and K. Reick. Power4 system design for high reliability. *IEEE Micro*, 22(2):16–24, March/April 2002.

[4] G. Bronevetsky, D. Marques, K. Pingali, and R. Rugina. Compiler-enhanced incremental checkpoint. In *Proceedings of Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2007.

[5] J. Chang, G. A. Reis, and D. I. August. Automatic Instruction-Level Software-Only Recovery. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 83–92, 2006.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[7] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-fault Recovery for Chip Multiprocessors. In *Proceedings of International Symposium on Computer Architecture (ISCA)*, pages 98–109, 2003.

[8] J. Hu, S. Wang, and S. G. Ziavras. In-register duplication: Exploiting narrow-width value for improving register file reliability. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 281–290, 2006.

[9] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[10] C. Lattner and V. Adve. The LLVM Compiler Framework and Infrastructure Tutorial. In *LCPC'04 Mini Workshop on Compiler Research Infrastructures*, 2004.

[11] C. 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 Intenational Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[12] A. Mahmood and E. McCluskey. Concurrent Error Detection Using Watchdog Processors - A Survey. *IEEE Transactions on Computers*, 37(2):160–174, 1988.

[13] D. McEvoy. The architecture of tandem's nonstop system. In *ACM 81: Proceedings of the ACM '81 conference*, page 245, 1981.

[14] C. McNairy and R. Bhatia. Montecito: A Dual-core, Dual-thread Itanium Processor. *IEEE Micro*, 25(2):10–20, March-April 2005.

[15] A. Meixner and D. J. Sorin. Error detection using dynamic dataflow verification. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 104–118, 2007.

[16] S. Michalak, K. Harris, N. Hengartner, B. Takala, and S. Wender. Predicting the Number of Fatal Soft Errors in Los Alamos National Laboratory's ASC Q Supercomputer. *IEEE Transactions on Device and Materials Reliability*, 5:329–335, September 2005.

[17] P. Montesinos, W. Liu, and J. Torrellas. Shield: Cost-Effective Soft-Error Protection for Register Files. In *Third IBM TJ Watson Conference on Interaction between Architecture, Circuits and Compilers (PAC206)*, 2006.

[18] S. S. Muchnick. *Advanced Compiler Design and Implementation*, pages 378–396. Morgan Kauffmann, 1997.

[19] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed Design and Evaluation of Redundant Multithreading Alternatives. In *Proceedings of International Symposium on Computer Architecture*, pages 99–110, 2002.

[20] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, page 29, 2003.

[21] N. Nakka, K. Pattabiraman, and R. Iyer. Processor-level selective replication. In *DSN '07: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 544–553, Washington, DC, USA, 2007. IEEE Computer Society.

[22] N. Oh, P. Shirvani, and E. McCluskey. Control-flow checking by software signatures. *Reliability, IEEE Transactions on*, 51(1):111–122, Mar 2002.

[23] F. Perry, L. Mackey, G. A. Reis, J. Ligatti, D. I. August, and D. Walker. Fault-tolerant typed assembly language. *SIGPLAN Not.*, 42(6):42–53, 2007.

[24] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2002.

[25] S. K. Reinhardt and S. S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *Proceedings of International Symposium on Computer Architecture*, pages 25–36, 2000.

[26] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software Implemented Fault Tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2005.

[27] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee. Design and Evaluation of Hybrid Fault-Detection Systems. In *Proceedings of the International International Symposium on Computer Architecture (ISCA)*, 2005.

[28] G. A. Reis, J.Chang, D. I. August, R. Cohn, and S. S. Mukherjee. Configurable Transient Fault Detection via Dynamic Binary Translation. In *Proceedings of the 2nd Workshop on Architectural Reliability (WAR)*, 2006.

[29] G. P. Saggese, N. J. Wang, Z. T. Kalbarczyk, S. J. Patel, and R. K. Iyer. An experimental study of soft errors in microprocessors. *IEEE Micro*, 25(6):30–39, 2005.

[30] T. Slegel, R. Averill, M. Check, B. Giamei, B. Krumm, C. Krygowski, W. Li, J. Liptay, J. MacDougall, T. McPherson, J. Navarro, E. Schwarz, K. Shum, and C. Webb. IBM's S/390 G5 Microprocessor Design. *IEEE Micro*, 19(2):12–23, March-April 1999.

[31] D. Sorin, M. Martin, M. Hill, and D. Wood. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2002.

[32] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault Recovery using Simultaneous Multithreading. In *Proceedings of International Symposium on Computer Architecture (ISCA)*, pages 87–98, 2002.

[33] C. Wang, H. seop Kim, Y. Wu, and V. Ying. Compiler-managed software-based redundant multi-threading for transient fault detection. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, 2007.

[34] N. J. Wang and S. J. Patel. ReStore: Symptom Based Soft Error Detection in Microprocessors. In *Proceedings of the International Conference on Dependable Systems and Network (DSN)*, pages 30–39, 2005.

[35] M. Wolfe. Beyond induction variables. *SIGPLAN Not.*, 27(7):162–174, 1992.

[36] Y. Yeh. Design Considerations in Boeing 777 Fly-by-wire Computers. In *Proceedings of the IEEE International High-Assurance Systems Engineering Symposium.*, pages 64–72, 1998.

[37] J. Yu, M. J. Garzaran, and M. Snir. Techniques for Efficient Software Checking. In *Proceedings of Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2007.