A Compiler Optimization to Reduce Soft Errors in Register Files

Jongeun Lee Aviral Shrivastava

Department of Computer Science and Engineering
Arizona State University, Tempe, AZ 85281, USA
{Jongeun.Lee, Aviral.Shrivastava}@asu.edu

Abstract

Register file (RF) is extremely vulnerable to soft errors, and traditional redundancy based schemes to protect the RF are prohibitive not only because RF is often in the timing critical path of the processor, but also since it is one of the hottest blocks on the chip, and therefore adding any extra circuitry to it is not desirable. Pure software approaches would be ideal in this case, but previous approaches that are based on program duplication have very significant runtime overheads, and others based on instruction scheduling are only moderately effective due to local scope. We show that the problem of protecting registers inherently requires inter-procedural analysis, and intra-procedural optimization are ineffective. This paper presents a pure compiler approach, based on inter-procedural code analysis to reduce the vulnerability of registers by temporarily writing live variables to protected memory. We formulate the problem as an integer linear programming problem and also present a very efficient heuristic algorithm. Our experiments demonstrate that our proposed technique can reduce the vulnerability of the RF by 33 $\sim$ 37% on average and up to 66%, with a small 2% increase in runtime. In addition, our overhead reduction optimizations can effectively reduce the code size overhead, by more than 40% on average, to a mere 5 $\sim$ 6%, as compared to highly optimized binaries.

Categories and Subject Descriptors D.3.4 [*Software*]: Programming languages, Processors—Code generation, Compilers, Optimization; B.8.1 [*Hardware*]: Performance and Reliability—Reliability, Testing, and Fault-Tolerance; C.3 [*Computer Systems Organization*]: Special-purpose and application-based systems—Real-time and embedded systems

General Terms Algorithms, Reliability, Performance

Keywords Embedded system, Soft error, Register file, Architectural vulnerability factor, Static analysis, Compilation, Link-time optimization

1. Introduction

Due to continuous technology scaling, soft errors—transient faults mainly caused by energetic particles—are becoming an important design concern for earthbound applications in addition to space applications [ITRS]. Traditionally, due to their large size, only memory structures like the main memory and caches were considered important for protection against soft errors. However, recently, Blome et al. [2006] observed that the majority of the faults both in combinational and sequential logic that affect the architectural state of a processor come from the register file. Since register files are accessed very frequently, corrupted data in the register file can quickly spread to other parts of the system, increasing chances of an error. While memory structures, like caches and the main memory are routinely protected using parity or Error Correcting Codes (ECC) [Mitra et al., 2005], protecting the Register File (RF) using such schemes is prohibitive not only because the RF is often in the timing-critical path of the processor [Shrivastava et al., 2004], but also because RF is one of the hottest blocks on the chip [Skadron et al., 2003], and adding any additional circuitry only exacerbates the situation.

In a bid to reduce the additional hardware circuitry, previous techniques have proposed to protect only a part of the register file [Montesinos et al., 2007]. While it is possible to select which variables to map onto the protected registers in hardware, power-efficient solution is for the compiler to map variables to the protected registers at a higher priority during register allocation [Lee and Shrivastava, 2009a]. Nevertheless, this still includes additional hardware overhead.

Complete software solutions come in the form of code duplication [Oh et al., 2002b, Reis et al., 2005] and control flow checking [Oh et al., 2002a], either partially or fully duplicate the program code in order to detect errors in the original program. Full duplication of the code will be able to detect any error including those in RF, but the overhead is generally high in terms of both code size and performance, whereas partial duplication or control flow checking is able to detect only a subset of errors in the RF. The only pure software approach at the compiler level is by Yan and Zhang [2005], in which they reduce the distance between loads and stores to protect the RF, but it is not very effective owing to the local nature of instruction scheduling.

Like Yan and Zhang [2005], we also rely on certain protected components in the processor (namely, the memory) to protect the contents of registers, but we take this one step further. We insert explicit load/store instructions to temporarily write live registers in memory in order to protect them. The idea of using the memory to protect registers is counter-intuitive, since whenever a variable is used it must be brought to the RF, so keeping it in the memory to protect it may result in significant performance degradation, which could make the variables stay longer in the register file on average, exposed to soft errors for a longer duration. The fundamental conflict is, that while performance is maximized by keeping live variables in the RF, protection is maximized by evicting live variables from the RF. Despite this conflict, our initial investigation into the scope of pure-compiler approaches shows promise.

LCTES'09, June 19–20, 2009, Dublin, Ireland.

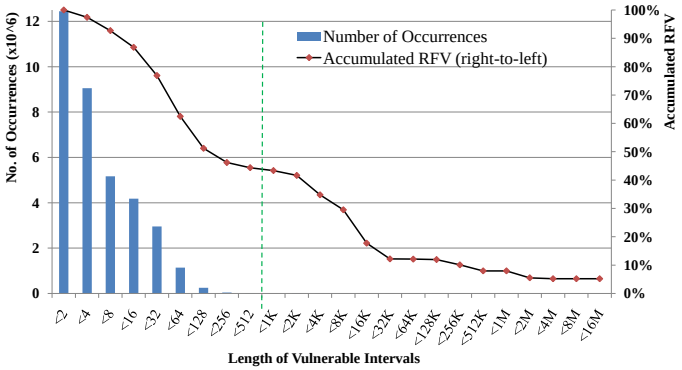


Figure 1. Histogram of vulnerable intervals showing the scope of compiler approaches. Vulnerable intervals that are 512 cycles or longer contribute about 43% of the RFV.

1.1 Motivation

On the lines of Architecture Vulnerability Factor proposed by Mukherjee et al. [2003], we define Register File Vulnerability (RFV) as a metric for soft error susceptibility of the Register File (RF) at the microarchitecture-level. A register is vulnerable at any moment of time, if the next access to the register will be a read by the processor, or if it will be stored into the memory which may be used later by the processor. A register is not vulnerable if it will be simply overwritten. An interval between consecutive accesses to the same register during which the register is vulnerable is defined as a vulnerable interval. During the execution of a program, a register may have multiple vulnerable intervals. The vulnerability of a register in the program is the amount of time during which the register was vulnerable, and can be calculated as the sum of the lengths of all its vulnerable intervals. Finally the vulnerability of the register file, or RFV is simply defined as the sum of vulnerabilities of all the registers.

We first observe that the vulnerability of program variables vary greatly depending on the type of registers they get assigned to. Variables assigned to caller-saved registers, or t-registers in the MIPS architecture [Yeager, 1996], have very short live ranges, since they have to be saved and restored before/after every function call. Contrarily, variables assigned to callee-saved registers, or s-registers, may have very long live ranges if the registers are not used in the callee function, since save and restore operations are not required when they are not used. The opportunity for compiler optimizations on reducing RFV is in those variables with long live ranges.

Second, the real opportunity for low-overhead RFV reduction comes when there is a long interval during which some registers are not accessed at all. To see how often such intervals would appear in real embedded applications, we profiled several applications from the MiBench benchmark suite [Guthaus et al., 2001] collecting the lengths of vulnerable intervals of all the registers. The applications were compiled to MIPS binary and executed on SimpleScalar simulator [Austin]. The vertical bars in Figure 1 plot the histogram of the occurrences of vulnerable intervals in the jpeg application. It shows that most vulnerable intervals are small. More than 99% of vulnerable intervals are less than 512 processor cycles long. While that is true, the more useful observation is that, even though the number of long vulnerable intervals is less, they contribute considerably to the total vulnerability of the RF. This is depicted by the continuous curve which plots the cumulative vulnerability, accumulated from right to left. It shows that the long vulnerable intervals ($>$ 512 cycles) contribute about 43% to the total RFV. If a compiler can identify all such long vulnerable intervals, it will be possible to significantly reduce the RFV at minimal power and performance overhead. Clearly there is significant scope for RFV reduction, and the effectiveness of the compiler techniques rests on how many long vulnerable intervals can the compiler discover, and if it will be able to protect those registers with minimal code overhead, while maintaining the program correctness.

1.2 This Work

A simple way to reduce RFV is to find heavily executed loops, identify unused registers in them, and save/restore the registers before/after the loops. However, with such an ad-hoc method, it is not only hard to achieve optimal results but also very cumbersome to handle complex control flows, function calls, and even recursive functions. Moreover, an intra-procedural optimization has a fundamental weakness that it can protect even unnecessary intervals, which significantly lowers the efficiency of ad-hoc methods. We approach this as an optimization problem: *given a performance bound, what is the set of program points in which to insert save/restore operations so that the transformed program will achieve the minimum RFV with minimal code size overhead?* This is inherently an inter-procedural problem, since register save/restore operations can easily affect other functions along the program paths, not only in terms of functionality but more in terms of RFV, and also because identifying long vulnerable intervals will necessarily demand considering more than one functions. Other challenges include devising simple yet effective save/restore operations, inserting them not overly but just enough to guarantee the program correctness, and accurately estimating their effect on performance, code size, and RFV, in the midst of complex control flows and function calls.

After discussing the limitations of intra-procedural approaches, we first formulate the problem as a large ILP (Integer Linear Programming) problem. This ILP involves far too many variables and constraints that it is practically impossible to solve. Hence we propose a scalable solution based on the concept of *access-free region*, or a connected subgraph in a control flow graph with no access to a particular register (it is analogous to vulnerable interval but in the static domain). Using access-free regions, our technique tries to find the best save/restore points by first discovering all the maximal access-free regions through an inter-procedural analysis, and then selecting the most profitable ones after cost-benefit evaluation. Additionally, we perform overhead reduction optimizations to reduce the code size overhead due to our transformation.

Our experimental results on a number of applications from MiBench benchmark suite [Guthaus et al., 2001] demonstrate that our compiler techniques can effectively reduce RFV, by up to 66%, or 33 $\sim$ 37% on average, with a small 2% increase in runtime. The RFV reduction by our technique is much higher compared to an average 9.4% reduction by an intra-procedural method mimicking ad-hoc optimization, and close to the potential maximum RFV reduction of 47% (on average). Also, our overhead reduction optimizations can effectively reduce the code size overhead, cutting it by more than 40%, to a mere 5 $\sim$ 6% compared to the original, highly optimized binaries.

2. Limitations of Intra-procedural Approach

Intuitively, most RFV is very likely to be generated by heavily executed loops and functions. Thus if we can reduce the RFV in those kernel loops, which account for, say, more than 95% of the execution time, we seem to be able to effectively reduce the total RFV, just like reducing the number of cache misses. However, there is an important difference between RFV and performance optimization. Whereas performance can be very much considered in isolation without considering other parts of the program, and still

```
function-main() {
   save register s1, s2;
   use register s1, s2;
   function-foo();
   s2 = function-bar();     // writing to s2
   s1 = s1 + s2;
   restore register s1, s2;
}

function-foo() {
   loop1 {
      use register t1;
   }
   use register t1, t2;
}

function-bar() {
   save register s1;
   loop2 {
      use register s1, t1, t2;
   }
   restore register s1;
}
```

Figure 2. Example program to illustrate limitations of intra-procedural approaches. For instance, whether `s2` is vulnerable in loop1 cannot be determined from function `foo`.

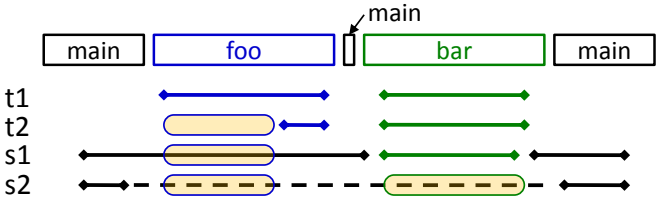


Figure 3. Vulnerable intervals of the example, represented by solid bars. The dashed bar in the last row is not vulnerable. Among the four round boxes indicating the intervals that would be protected by ad-hoc method, only one (`s1`) is vulnerable.

be quite accurate, RFV can be sometimes completely dependent on other parts than the part in question, which thwarts any local approach to optimizing RFV.

To illustrate the point, let us examine an ad-hoc method to reduce RFV, that is to first identify unused registers in most heavily executed loops and then protect them. We show through an example that it can lead to a very inefficient solution. Here we assume that there are only four registers available, and two of them, namely `t1` and `t2`, are caller-saved registers that callee functions can use without first saving them, while the others, namely `s1` and `s2`, are callee-saved registers that callee functions must save before use. Ignoring other details, the program from the register access point of view can look like Figure 2. The program consists of three functions. Function `main` uses s-registers only as it expects to call other functions. Function `foo` uses t-registers only, which can be used without saving. Lastly, function `bar` uses all but `s2` registers. Note that in function `bar`, `s1` must be saved before use whereas `t1` and `t2` can be used right away. There are only two types of operations on a register: read and write. From this point of view, save is a read (from a register), restore is a write (to a register), and use may be any.

Now, straightforward application of the ad-hoc method will identify `t2`, `s1`, and `s2` as unused registers in loop1, and `s2` as the unused register in loop2, all of which will be protected. However, only one of them is really necessary while the others just add to the overhead. To see this, let us consider the vulnerable intervals of the program, which is illustrated in Figure 3. The top row represents the function execution sequence, and the next rows show the vulnerable intervals for each register. Note that live range of a t-register is always contained in one function whereas live range of an s-register may span multiple functions. The live ranges of `s2` are particularly interesting. Register `s2` is live in the first and last segments of `main` but not inbetween because the interval (dashed line) ends with a write (see line 5 of Figure 2). Thus only the solid lines are vulnerable intervals and their total length is equal to the RFV of the program. Now if we locate in the diagram the intervals that would be protected by the ad-hoc method (marked with round boxes), we can see that only one of them is vulnerable. All the others are unnecessarily protected. This "false protection effect" that an optimization tries to protect register intervals that are never or rarely vulnerable can significantly lower the efficiency of the optimization.

To check whether an interval is live at a certain point requires in general an inter-procedural analysis. In our example, whether `t2` is live at the end of loop1 can be known simply by looking at the first access after the loop (which is also in the same function). In this example, the first access is, and must be, a write because `t2` does not appear before the loop, but it could be a read if `t2` were used before the loop. In either case, the liveness of a t-register can be known by an intra-procedural analysis. However, s-registers are more tricky. If an s-register is not used in a function, e.g., `s1` and `s2` in `foo` or `s2` in `bar`, other functions must be consulted to determine its liveness. And since there is no bound on how many caller/callees have to be searched in the call graph before finding the first access, which can be either read or write, and since call graphs may contain cycles, our problem requires a nontrivial inter-procedural analysis.

3. Problem Formulation

The problem is to find the set of program locations to insert save and restore operations that will maximize RFV reduction, with minimal code size overhead, under a given performance bound. Specifically,

- Input: τ (performance tolerance), optimized binary of the program
- Output: $\mathcal{S}_r$ (set of program points to insert save operation for register r), $\mathcal{R}_r$ (set of program points to insert restore operation for register r)
- Objective: Maximize RFV reduction and minimize code size overhead
- Constraints: Runtime overhead should be less than τ; program behavior must remain the same.

We can consider the save and restore operations as *mode* changing operations. Hence, our system has two modes, which we call A ("unprotected") and B ("protected"). Initially the program starts in mode A. The save and restore operations change the mode from A to B and from B to A, respectively. There are two modes for each architectural register in the processor, but since our technique deals with each register individually, we will often refer to the modes as if there are only two of them.

Mode change operations affect the mode of the program execution until the next mode change operation is performed. Therefore the mode is determined at runtime by the *path* of the program execution and not by the program location. In other words, an instruction at a certain program location could be executed in either mode at runtime depending on the program execution path leading

up to the program location. However, this dependence of mode resolution on execution paths makes static analysis very hard and it becomes even more challenging to guarantee program correctness. Thus we require that each program point be associated with only one of the two modes with respect to each register.

This small assumption transforms our problem into that of partitioning, where we need to partition all the program points into two groups. To further simplify our discussion let us consider the problem at the basic block granularity (thus we only need to partition basic blocks). Then to preserve the program semantics requires that we map any basic block accessing a register to mode A with respect to the register. For the rest of the basic blocks, which are called *access-free blocks*, we need to find the optimal mapping so that the mode change overhead is minimized and the RFV reduction is maximized.

This leads to an ILP formulation as follows. We require branch probabilities, which can be obtained from either static analysis [Wu and Larus, 1994] or profiling, to compute all the execution counts of basic blocks and edges between them. For this ILP formulation only, we further require register liveness information of each basic block, or the probability of a register being live at the end of a basic block, which may be obtained from static analysis [Lee and Shrivastava, 2009b] or profiling.

Input:

- τ: runtime tolerance in dynamic instruction count
- $G = (V, E)$: Inter-procedural Control Flow Graph (ICFG) of the program, where V is the set of basic blocks and E is the set of edges representing the control flows
- n_i: number of instructions of basic block $i \in V$
- b_i: execution count of basic block $i \in V$
- $f_{i,j}$: execution count of edge $(i, j) \in E$
- R: the set of architectural registers
- l_i^r: liveness of register $r \in R$ at the end of block $i \in V$

Output:

- x_i^r: binary variables denoting the mode of basic block $i \in V$ with respect to register $r \in R$ (1 if mode B).

Let Acc_r be the set of basic blocks in which register r is accessed at all. Then the following is necessary to preserve program semantics.

$$\forall r \in R, \forall i \in Acc_r : \quad x_i^r = 0 \tag{1}$$

Also, on every edge $(i, j) \in E$ where blocks i and j are mapped to different modes we must insert a mode change operation, unless the register is statically known to be not alive at the end of i. This condition is modeled by binary variables $y_{i,j}^r = (x_i^r \neq x_j^r) \wedge (l_i^r \neq 0)$, with 1 denoting that a mode change operation is required. The above equation can be linearized using auxiliary binary variables $t_{i,j}^r, u_{i,j}^r$. The term $(l_i^r \neq 0)$ is a constant evaluating to either 0 or 1.

$$\begin{aligned} &\forall r \in R, \forall (i,j) \in E : \\ &\quad x_i^r + x_j^r = t_{i,j}^r + 2u_{i,j}^r, \\ &\quad y_{i,j}^r \leq t_{i,j}^r,\ y_{i,j}^r \leq (l_i^r \neq 0), \\ &\quad y_{i,j}^r \geq t_{i,j}^r + (l_i^r \neq 0) - 1 \end{aligned} \tag{2}$$

Then the code size increase C and runtime increase R can be easily expressed.

$$C = \sum_r \sum_{(i,j) \in E} y_{i,j}^r$$

$$R = \sum_r \sum_{(i,j) \in E} f_{i,j} y_{i,j}^r$$

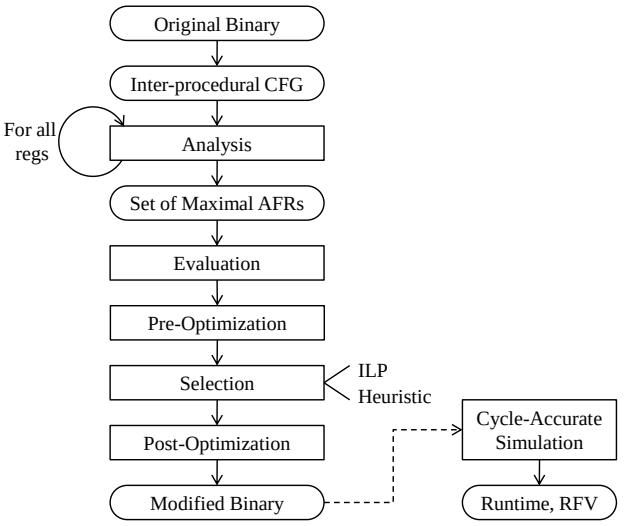


Figure 4. Overall flow of our technique.

The RFV reduction is equal to the total register vulnerability in the basic blocks mapped to mode B. Approximating the time spent in basic block i to the number of instructions n_i, the RFV reduction can be calculated as follows [Lee and Shrivastava, 2009b]:

$$V = \sum_r \sum_{i \in V} n_i b_i l_i^r x_i^r$$

Now the ILP is to maximize $V - \alpha C$ for some weighting parameter α while satisfying (1), (2), and $R < \tau$. As the number of basic blocks can be large even for a modest-size program, solving this ILP may require prohibitive amount of resources. Therefore we need a more scalable solution, which we present in the next section.

We realize that our problem bears some resemblance with the min-cut graph partitioning problem [Garey and Johnson, 1979], which is NP-complete and has many heuristic algorithms including [Kernighan and Lin, 1970], in that our problem also favors partitioning with smaller cut cost. However, several complications make direct application of existing heuristics very difficult. First, the objective, RFV reduction, is determined not only by the characteristics of the blocks mapped to mode B, but also by what comes after the blocks in the program flow. Although to avoid this issue we assumed register liveness information (l_i^r) in this section, in the next section we develop an algorithm that do not require such information as an input. Second, register liveness also influences the cost of a cut, possibly eliminating it, as represented by variables $y_{i,j}^r$. Third, our goal is to reduce the code size as well as the runtime. Straightforward application of min-cut partitioning algorithms would try to minimize only one of them. Finally, the cost and objective functions are defined collectively considering all the registers. Solving a min-cut partitioning problem for each register could result in extremely poor solutions, although we could also merge the graphs at the cost of much longer solving time.

4. Proposed Solution

4.1 Overview

Our solution is based on an intuitive idea that the largest RFV reduction with the smallest overhead results when we map to mode B an entire loop or function containing no access to some registers. To exploit this idea, we define *Access-Free Region* (AFR) as a connected subgraph of an Inter-procedural Control Flow Graph (ICFG) [Harrold et al., 1998] containing access-free blocks only.

(Considering AFRs instead of individual blocks also makes it much easier to reason about register liveness, which influences both cost (runtime, code size) and benefit (RFV reduction) in our problem definition.) Then, a *maximal* AFR, which is an AFR contained by no other AFR but itself, can generate the greatest RFV reduction, and closely matches our notion of access-free loops and functions. Our method is essentially to discover all the maximal AFRs in the program and select the best ones through cost and benefit analysis (see Figure 4). As will be discussed next, mode change operations originally required by our method have relatively large overhead. Pre- and post-optimizations in the flow reduce such overhead by moving mode change operations around.

4.2 Mode Change Operation

Mode change operations can be implemented using load/store instructions. The memory addresses used by the load/store instructions can be either stack-relative or absolute (we need as many memory locations as the number of registers). However, using stack-relative addresses requires that all the mode change operations for a selected AFR exist in the same function,[1] which is quite restrictive, whereas using absolute addresses allows for AFRs whose boundaries may be distributed over multiple functions. Absolute addresses may be generated using the global pointer (`gp`) or, if any, constant register (e.g., `r0` in the MIPS architecture). The register used as the base address register in the load/store instructions (stack pointer, global pointer, etc.) can no longer be protected by our optimization, which is more consequential for global pointer than stack pointer, as stack pointer is usually more frequently accessed, and has less opportunity for RFV reduction, than the other.

Changing modes at edges of an ICFG, as opposed to doing it in basic blocks, achieves the minimum number of mode changes at runtime. However, implementing mode changes on edges requires one more instruction—an unconditional jump—in addition to a store/load instruction. Although unconditional jumps can be accurately predicted by modern processors and may not cause a significant performance penalty especially in out-of-order execution processors, the code size effect is more difficult to mitigate. One way to remove the unconditional jumps is to convert edge insertion points into node insertion points, which we do in pre- and post-optimizations.

4.3 Inter-procedural Analysis

The purpose of our analysis is to find all the maximal AFRs in an ICFG. Finding maximal AFRs, or finding maximally connected subgraphs containing only access-free blocks, can be done very efficiently using the algorithm listed in Alg. 1. To avoid recursion the algorithm uses a "work queue" implemented as a set ($nodeset$). The algorithm iterates over all the nodes once, checking if they are already processed. If not, a node and all the connected nodes are labeled with a new region ID, eventually partitioning all the access-free blocks into maximal regions. The mapping from access-free blocks to region numbers is stored in AFR. The complexity of this algorithm is $O(|E|)$, where $|E|$ is the number of edges in the ICFG.

4.4 Evaluating Maximal Access-Free Regions

Mode change operations must be inserted at the boundaries of selected AFRs except for the locations where the register is known to be not live. Finding out whether a register is live at an outgoing edge is easy because maximal AFRs must neighbor non-access-free blocks, which directly give the first access (the register is live only if it is first read after following the edge). For an incoming edge, we do not have to check the register liveness; if it is not live, it must be not live at every outgoing edge too, and the RFV of the AFR must also be zero. If the RFV is zero, the AFR will not be selected anyway. Once we have found all the boundary edges where the register may be live, the code size overhead is simply twice the number of edges (load/store + unconditional jump), and the runtime overhead is the combined execution counts of the edges multiplied by two. The execution counts of basic blocks and control flow edges can be easily computed from branch probabilities, which can be obtained from either static analysis [Wu and Larus, 1994] or profiling.

[1] To be exact, between two stack manipulation instructions, which are at the beginning and at the end of a function.

Algorithm 1 Find all maximal access-free regions in ICFG

```
input: ICFG = (V, E)
output: AFR := V → {region_id} : initialized to zero
for all n ∈ V do
    if AFR[n] ≠ 0 or n is not an access-free-block then
        continue to the next iteration
    end if
    nodeset ← {n}
    id ← id + 1
    repeat
        m ← take one from nodeset
        AFR[m] ← id
        for all k that is a successor or predecessor of m do
            if AFR[k] = 0 and k is access-free-block then
                nodeset ← nodeset ∪ {k}
            end if
        end for
    until nodeset is empty
end for
```

The benefit, or RFV reduction, of selecting an AFR is the RFV of the AFR, or $\sum_i n_i b_i l_i$, a summation over all the basic blocks included in the AFR (n_i, b_i, l_i are the number of instructions, the execution frequency, and the liveness of basic block i; here, execution time is approximated with dynamic instruction count). To avoid using l_i, we approximate the RFV with $\mu \sum_i n_i b_i$, where μ is the probability of first seeing a read access after exiting the AFR. This probability can be computed rather accurately, since maximal AFR must neighbor non-access-free blocks. Note that finding μ is very similar to finding live outgoing edges, except that we weigh the edges according to their execution frequencies to obtain a single number μ.

4.5 Selection Problem

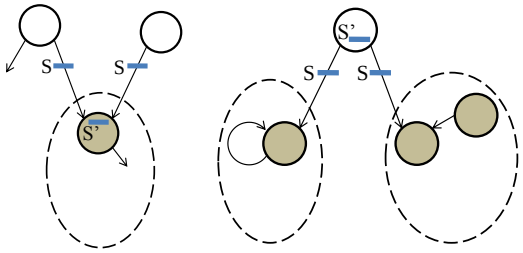
Having found all the maximal AFRs for all the registers that may be protected, the next step is to find the best ones that collectively maximize the RFV reduction subject to the cost constraint. Let v_k, c_k, t_k be the RFV reduction, code size increase, and runtime increase, respectively, of AFR k. Let x_k be the binary variables denoting that AFR k is selected (1 if selected). Then the selection problem is to maximize $\sum_k (v_k x_k - \alpha c_k x_k)$ while satisfying $\sum_k t_k x_k < \tau$, which is a knapsack problem. We can use an ILP solver to solve this problem.

Alternatively, we can use this very simple heuristic:

1. sort the AFRs in the order of decreasing $(v_k - \alpha c_k)/t_k$
2. select from the top of the sorted list until their combined runtime overhead reaches τ

4.6 Pre- and Post-optimizations

Since adding a mode change operation to an edge requires one more instruction than adding it to a node, node insertion points are preferred to edge insertion points. Also, if N mode change oper-



(a) Inward move (b) Outward move

Figure 5. Moving mode change operations from edge to node for save operations. Solid circles represent nodes, or basic blocks (white: mode A, gray: mode B), and dashed ovals represent selected maximal AFRs. Thick small bars represent save operations (S: before move, S': after move).

ations on N different edges, sharing the same node, are replaced by a single mode change operation on the node, we can reduce the number of instructions to $1/2N$. Thus the goal of pre- and post-optimizations is to minimize the code size overhead while not increasing the runtime or RFV of application code.

Without changing the semantics we can move mode change operations from an edge to a node if all the incoming (or outgoing) edges have the same mode change operation. Figure 5 illustrates examples for save operations. Such a move does not affect the RFV or runtime overhead of selected AFRs, but can reduce the code size overhead significantly. As a result of a move, mode change operations move either inside or outside an AFR, called an *inward* or *outward* move. Inward moves can be performed for each maximal AFR even before we make a selection, whereas outward moves can be performed only on the selected ones. Thus we perform inward moves before selection (pre-optimization) and outward moves after selection (post-optimization).

5. Experiments

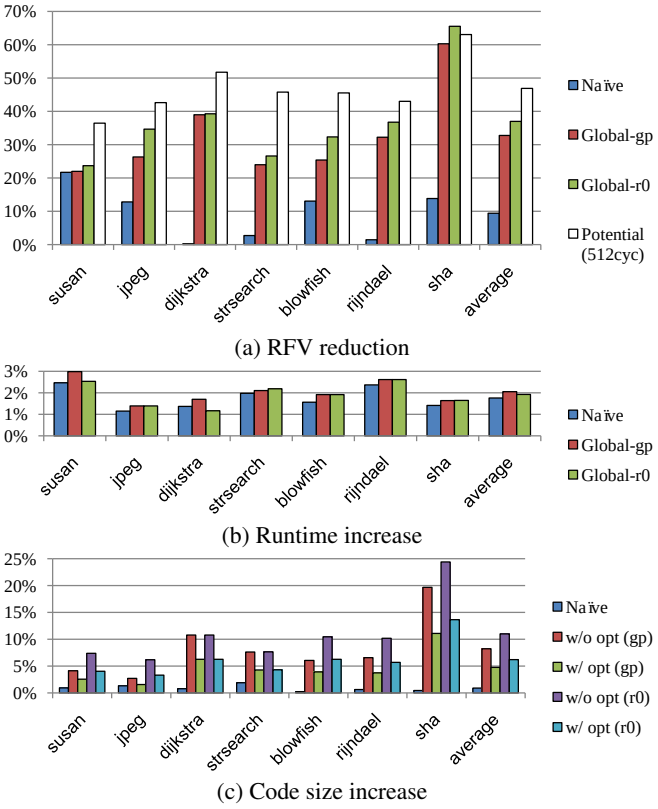
To evaluate the effectiveness of our compiler technique we use applications from MiBench benchmark suite [Guthaus et al., 2001]. We compile applications using GCC 2.7.2.3[2] with the optimization level specified in the benchmark suite, and simulation is done using the SimpleScalar cycle-accurate simulator [Austin]. The proposed compiler optimization is implemented as a post-link optimizer to be able to handle library functions, which may play a crucial role in determining RFV. The execution counts of basic blocks and control flow edges are computed using a linear equation method similar to [Chen et al., 2001] from branch probabilities obtained from initial simulation. For the tolerance parameter we use either 1% or 2%, which is determined on an application basis.[3] For weighting parameter α, we use $0.5V_o/C_o$, where V_o and C_o are the RFV and code size of the original program, respectively.

5.1 Naïve Approach

For a comparison we also implement an intra-procedural optimization based on our proposed technique, which we call naïve approach. We use exactly the same flow as shown in Figure 4 except that we do an intra-procedural analysis on a set of Control Flow Graphs (CFGs). Finding maximal AFRs in a function can be done very efficiently using Alg. 1, provided that function calls are removed. We resolve a function call by replacing it with a node, which is considered access-free only if the replaced function is

[2] This is one of the latest versions supporting the SimpleScalar target.

[3] The 1% tolerance was used for jpeg, dijkstra, and sha only.



(a) RFV reduction

(b) Runtime increase

(c) Code size increase

Figure 6. Comparing different approaches in terms of RFV, runtime, and code size.

access-free. This means that we should analyze callee functions before caller functions, or in the depth-first order of the call graph. For recursive functions, it is easy to see that either all the functions in a cycle are access-free or none at all; checking which is the case is not a difficult problem. Exact cost and benefit estimation is rather difficult for maximal AFRs derived from CFGs. Even though we use ICFG for the evaluation step, outgoing edges may neighbor access-free blocks in another function. In such a case, it is not straightforward to find the exact type of the first access, which may involve following many blocks down the control flow; instead, we conservatively assume no-access as a read. For the addressing mode of the mode change operations we use `r0`, which is expected to be better than global pointer or stack pointer. For the selection step our heuristic algorithm is used.

5.2 Effectiveness of Our Compiler Technique

First we compare our compiler technique with the naïve approach in terms of RFV reduction as well as runtime and code size overhead, which is summarized in Figure 6. Note that the RFV reduction and runtime overhead are measured through cycle-accurate simulation after applying optimizations to the original application binaries. Thus the effect of extra memory accesses through increased cache misses, for instance, is all accounted for. We verified that the transformed applications have the same functionality as the originals, which is also evidenced by the small increase in runtime. In the first graph, we also show the potential RFV reductions, which are obtained through profiling by accumulating the live vulnerable intervals that are at least 512 cycles long. In the graphs, Global-gp and Global-r0 represent our proposed technique using `gp` and `r0`, respectively, as the base address register in mode change operations. For the results in this subsection we use our heuristic algo-

Table 1. Comparing ILP (I) vs. Heuristic (H)

	RFV reduction	Runtime increase	Code size overhead		Time (sec)
			w/o opt	w/ opt	
I	32.3%	2.11%	8.6%	6.4%	< 1
H	32.7%	2.05%	8.2%	4.8%	< 1

rithm in the selection step; the effect of using ILP is compared in the next subsection. Our code size reduction optimization has very little effect on RFV or runtime; thus we show its effect only in the third graph.

From the first graph, we see that the potential RFV reduction is usually high, on average 47%, and up to 63% in the case of sha. Our technique can realize most of the potential, achieving significant RFV reduction of 33 $\sim$ 37% on average, and up to 66% if we use `r0` as the base register. Note that our compiler technique achieving greater RFV reduction than the potential is not a contradiction, since the potential we use is computed at the threshold of 512 cycles but can be higher if the threshold is lower. Between our methods, Global-r0 achieves higher RFV reduction, since it can perform optimization even on the `gp` register. Sometimes, the missed opportunity translates into a lower RFV reduction as is the case with jpeg and blowfish, but other times, for instance, in susan and dijkstra, Global-gp fills the gap with opportunities from other registers. This is evidenced by the noticeable increase in runtime in those two applications (from Global-r0 to Global-gp), which means that Global-gp musters more smaller access-free regions to substitute for the missing larger access-free regions from `gp`.

The RFV reduction by the naïve approach varies greatly with an average of 9.4%. In most applications it achieves far lower RFV reduction than the proposed technique, with susan being the only aberration. Application susan is special in that it is the only application with a function that is called only once, but which accounts for 95% of the total runtime. For such a simple call scenario, interprocedural analysis would not be necessary. On the other hand, in some applications such as dijkstra, strsearch, and rijndael the RFV reduction is almost insignificant, which is primarily because of the "false protection effect" that the optimization tries to protect a region where the register is rarely or never vulnerable.[4]

The overhead of our optimization is not high, about 2% runtime increase and 5 $\sim$ 6% code size increase on average. The third graph shows that our technique adds far more instructions than the naïve approach, which also suggests the extensiveness of our technique. It also demonstrates that our overhead reduction optimizations are very effective in reducing the code size overhead; the code size overhead is reduced consistently in all the cases, and by over 40% on average.

5.3 ILP vs. Heuristic

Table 1 compares the two selection methods, ILP and our heuristic algorithm. For this comparison we use `gp` as the base address register. Again, the RFV and runtime numbers are from actual simulations, averaged for all the applications. Overall, the quality of solutions found by our heuristic is as good as that of ILP. The minute differences between them may seem to suggest that our heuristic is slightly better than ILP, which cannot be true. This is an artifact of RFV dependence on runtime, ignored in our optimization framework. In a bid to maximize the RFV reduction, ILP chooses the maximum possible runtime and code size, which, however, has an adverse effect on RFV (RFV is increased roughly by the amount of the runtime increase), which is why ILP does not necessarily achieve the maximum RFV reduction in reality. The only significant difference in this comparison is on the code size overhead after applying our overhead reduction optimizations, which tend to work better with our heuristic algorithm. The processing time for the selection step is less than one second per application in either case, on a PC with 2GHz Xeon (single threaded execution) with 4GB memory.

[4] The false protection effect also creates the "false RFV increase" effect, making the interval up to the store operation appear vulnerable. We have taken care not to count such false RFV increases in all our RFV measurements.

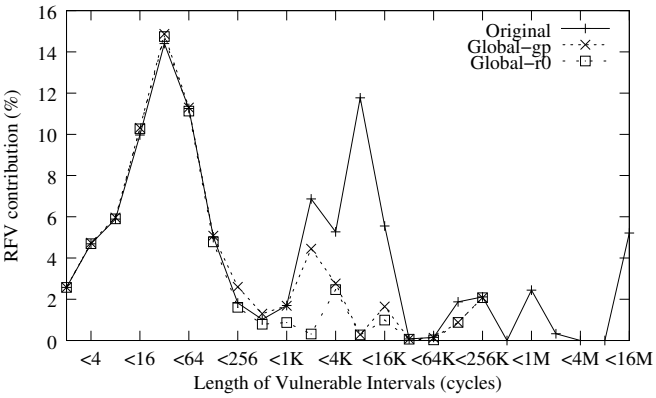


Figure 7. Comparing RFV distributions before/after the proposed optimization.

5.4 RFV Distribution

Figure 7 shows the RFV contributions made by vulnerable intervals of different sizes; for instance, the first point in the graph represents that the vulnerable intervals of length 1 generates about 2.5% of the total RFV. Thus the area under the graph represents the total RFV. This information is collected during the simulation of the original and modified binaries of jpeg. The initial goal stated in Introduction was to remove all the RFV components of 512 cycles or more. The graph shows that our methods can achieve the goal, successfully filtering out most of the long-interval RFV components while leaving intact the short-interval RFV components. However there are some intervals especially in the 64K $\sim$ 256K range, which could not be identified or protected for some reason. We leave finding a low-cost solution to protect even such intervals to future work.

6. Related Work

Many techniques exist that share the same goal with our work, of protecting register files from soft errors. First, there are hardware techniques [Blome et al., 2006, Kandala et al., 2007, Memik et al., 2005, Montesinos et al., 2007, Slegel et al., 1999], which protect registers using ECC, parity, or duplication, without necessary support from compiler or other software. Since providing full hardware protection for the entire RF has extremely high overhead, many cost-effective solutions [Blome et al., 2006, Memik et al., 2005, Montesinos et al., 2007] protect only part of the RF, with the decision of which variables to protect being made in hardware. Second, there are compiler-hardware hybrids [Lee and Shrivastava, 2009a, Yan and Zhang, 2005], which use compiler for the decision of what-to-protect, thus eliminating the overhead of hardware decision and/or improving the quality of decisions using compile-time analysis. The third category is pure-compiler solutions, of which we are aware of only one previous technique. Yan and Zhang [2005] proposes an instruction scheduling that tries to reduce the distance between loads and stores in a bid to lower RFV. The reduced distance between a load (which is a write) and a store (which is a read) translates into reduced RFV, which is, however, bounded

by the size of the block in which instruction scheduling is done. Thus, not only instruction scheduling is only a local optimization, the scope of it is also limited by the size of the block, which cannot be as large as a whole function or multiple functions unlike in our approach. As a result, their instruction scheduling generates mixed results, *increasing* RFV in 50% of the cases. But in the other cases where the instruction scheduling can reduce the RFV, our technique can be applied on top of the instruction scheduling, since they work at different granularities. Finally, there are software techniques such as code duplication [Oh et al., 2002b, Reis et al., 2005] and control flow checking [Oh et al., 2002a], which are different from the above approaches in that they achieve reliability through software redundancy [Koren and Krishna, 2007] rather than space redundancy (e.g., register duplication) or information redundancy (e.g., ECC and parity).

The concept of vulnerability was first introduced as Architectural Vulnerability Factor (AVF) by Mukherjee et al. [2003], which is a quantitative measure of the amount of "live" information that needs protection of each microarchitectural component. The vulnerability measure is used in nearly all cost-effective soft error mitigation techniques at the architecture level and above to approximate the probability of soft errors. Techniques have been proposed to compute AVF during simulation [Mukherjee et al., 2003], or estimate it at runtime [Li et al., 2008, Walcott et al., 2007] or at compile-time [Lee and Shrivastava, 2009b].

This work shares some insights with partially protected RF techniques though they are different in many aspects including the constraints and the mechanisms to achieve vulnerability reduction. With partially protected RFs, it proved to be key for cost-effectiveness, to treat register variables differently depending on their live lengths, in both hardware approaches [Montesinos et al., 2007] and our earlier work on compiler-hardware hybrids [Lee and Shrivastava, 2009a]. In this approach, however, since a live range can consist of multiple vulnerable intervals, we look more finely at the vulnerable intervals, treating them differently to maximize the efficiency of our compiler optimization.

For a more complete comparison, it is worthwhile to consider reducing the number of registers used during compilation. Reducing the number of registers used may reduce the overall liveness of registers, and thus the total RFV. And it can be done rather easily, with gcc for instance, using compiler switches specifying what registers must not be used. However, recompiling applications with fewer registers has several limitations. First only certain registers can be disallowed such as t-registers and s-registers. Special registers such as global/stack/frame pointers, and argument and return value registers, are outside the scope. Second there are only a certain number of choices to choose from, and there is no control or estimation of exactly how much impact the choice will make on vulnerability, code size, and performace. This is very different from our approach, which can control the performance penalty with maximum RFV reduction.

7. Conclusion

This paper presents a case for a pure compiler approach to reducing soft errors in processor register files. Unlike in performance optimization, optimizing for reliability necessitates a global approach—optimizing only kernels without the knowledge of global consequences may result in extremely poor results. To provide a quantitative answer to the question of how effective a pure compiler approach can be, we formulated an optimization problem, turned it into a graph partitioning one, and proposed an efficient heuristic solution based on access-free regions. Our experiments on a number of embedded applications demonstrate that our technique can reduce RFV very effectively, by $33 \sim 37\%$ on average and up to 66%, at a nominal 2% performance overhead, which is far better than an intra-procedural approach, and approaches the potential maximum. Also, our overhead reduction optimizations can successfully reduce the code size overhead, cutting it by more than 40%, to a mere $5 \sim 6\%$ compared to the original, highly optimized binaries. One limitation of our technique is that it uses only the maximal access-free regions as candidates for protection. While exploring all the subsets of maximal access-free regions is clearly intractable, we intend to apply heuristics from graph partitioning approaches to this problem in the near future.

Acknowledgments

This work was partially supported by Raytheon, Stardust Foundation, and the Korea Research Foundation Grant (KRF-2007-357-D00225) funded by the Korean Government (MOEHRD). The authors also would like to thank members of Compilers and Microarchitecture Lab (CML) for their valuable support in this work.

References

Todd Austin. SimpleScalar LLC. URL `http://www.simplescalar.com/`.

Jason A. Blome, Shantanu Gupta, Shuguang Feng, and Scott Mahlke. Cost-efficient soft error protection for embedded microprocessors. In *CASES '06*, pages 421–431, 2006.

Kaiyu Chen, Sharad Malik, and David I. August. Retargetable static timing analysis for embedded software. In *ISSS '01*, pages 39–44, 2001.

M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

M. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IWWC*, pages 3–14, 2001.

M. Harrold, G. Rothermel, and S. Sinha. Computation of interprocedural control dependence. In *Proc. ACM SIGSOFT Int'l Symp. on Software Testing and Analysis*, pages 11–20, 1998.

ITRS. International technology roadmap for semiconductors 2007 executive summary. URL `http://www.itrs.net/`.

M. Kandala, W. Zhang, and L. Yang. An area-efficient approach to improving register file reliability against transient errors. In *Proc. Int'l Conf. on Advanced Information Networking and Applications Workshops*, pages 798–803, 2007.

B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Tech. Journal*, 49:291–307, February 1970.

Israel Koren and C. Mani Krishna. *Fault-Tolerant Systems*. Morgan Kaufmann Publishers, 2007.

Jongeun Lee and Aviral Shrivastava. Compiler-managed register file protection for energy-efficient soft error reduction. In *Proceedings of Asia and South Pacific Design Automation Conference*, pages 618–623, 2009a.

Jongeun Lee and Aviral Shrivastava. Static analysis to mitigate soft errors in register files. In *Proc. Int'l Conf. Design Automation and Test in Europe (DATE)*, 2009b.

Xiaodong Li, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. Online estimation of architectural vulnerability factor for soft errors. *SIGARCH Comput. Archit. News*, 36(3):341–352, 2008.

G. Memik, M. H. Chowdhury, A. Mallik, and Y. I. Ismail. Engineering over-clocking: reliability-performance trade-offs for high-performance register files. *Proc. Int'l Conf. on Dependable Systems and Networks*, 2005.

Subhasish Mitra, Norbert Seifert, Ming Zhang, Quan Shi, and Kee Sup Kim. Robust system design with built-in soft-error resilience. *IEEE Computer*, 38(2):43–52, 2005.

Pablo Montesinos, Wei Liu, and Josep Torrellas. Using register lifetime predictions to protect register files against soft errors. In *DSN '07*, pages 286–296, 2007.

Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proc. International Symposium on Microarchitecture*, Dec 2003.

Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. Control-flow checking by software signatures. *IEEE Transactions on Reliability*, 51:111–122, 2002a.

Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, 51:63–75, 2002b.

George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. SWIFT: Software implemented fault tolerance. In *Int'l Symp. Code Generation and Optimization*, pages 243–254, 2005.

Aviral Shrivastava, Eugene Earlie, Nikil D. Dutt, and Alexandru Nicolau. Operation tables for scheduling in the presence of incomplete bypassing. In *CODES+ISSS*, pages 194–199, 2004.

K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. In *Proc. Int'l Symp. on Computer Architecture*, pages 2–13, 2003.

Timothy J. Slegel, Robert M. III Averill, Mark A. Check, Bruce C. Giamei, Barry W. Krumm, Christopher A. Krygowski, Wen H. Li, John S. Liptay, John D. MacDougall, Thomas J. McPherson, Jennifer A. Navarro, Eric M. Schwarz, Kevin Shum, and Charles F. Webb. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19(2):12–23, 1999.

Kristen R. Walcott, Greg Humphreys, and Sudhanva Gurumurthi. Dynamic prediction of architectural vulnerability from microarchitectural state. *SIGARCH CA News*, (2):516–527, 2007.

Youfeng Wu and James Larus. Static branch frequency and program profile analysis. In *MICRO 27*, pages 1–11, 1994.

Jun Yan and Wei Zhang. Compiler-guided register reliability improvement against soft errors. In *EMSOFT '05*, pages 203–209, 2005.

K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 1996.