Software-Based Register File Vulnerability Reduction for Embedded Processors

JONGEUN LEE, UNIST
AVIRAL SHRIVASTAVA, Arizona State University

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. Software approaches would be ideal in this case, but previous approaches based on instruction scheduling are only moderately effective due to local scope. In this article we present a compiler approach, based on interprocedural program 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. Further we present an iterative optimization method based on Kernighan-Lin's graph partitioning algorithm. Our experiments demonstrate that our proposed techniques can reduce the vulnerability of a RF by $33 \sim 37\%$ on average and up to 66%, with a small 2% increase in runtime. In addition, our overhead reduction optimization can effectively reduce the code size overhead, by more than 40% on average, to a mere $5 \sim 6\%$, compared to highly optimized binaries.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Code generation, compilers, optimization*; B.8.1 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance

General Terms: Algorithms, Reliability, Performance

Additional Key Words and Phrases: Soft error, register file, vulnerability, static analysis, embedded system, compilation, link-time optimization

ACM Reference Format:
Lee, J. and Shrivastava, A. 2013. Software-based register file vulnerability reduction for embedded processors. ACM Trans. Embedd. Comput. Syst. 13, 1s, Article 38 (November 2013), 20 pages.
DOI: http://dx.doi.org/10.1145/2536747.2536760

1. INTRODUCTION

Soft errors, or transient faults mainly caused by energetic particles, are growing in importance as the technology scaling continues. Traditionally only large memory structures such as DRAM memories and caches were considered important to protect against soft errors. However, for embedded processors, Register Files (RFs) can be responsible for the majority of faults affecting the architectural state of the processor [Blome et al. 2006]. Moreover, since RFs are accessed very frequently, corrupted data in RFs can quickly spread to other parts of the system, adding to the importance of protecting RFs. While memory structures are routinely protected using parity or Error Correcting

This work was partially supported by funding from National Science Foundation grants CCF 1055094 (CAREER), and by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology, under grant 2010-0011534.
Author's addresses: J. Lee, School of ECE, Ulsan National Institute of Science and Technology (UNIST), Ulsan 689-798, Korea; email: jlee@unist.ac.kr; A. Shrivastava, Department of CSE, Arizona State University, Tempe, AZ 85281.

DOI: http://dx.doi.org/10.1145/2536747.2536760

Codes (ECCs) [Mitra et al. 2005; Fazeli et al. 2008], protecting RFs poses unique challenges because a RF is typically in the timing-critical path of a processor, and is one of the hottest blocks of a chip. Keeping a RF cool is important not only to save power but also because higher temperature rapidly reduces the reliability of circuits [Dodd and Massengill 2003]. Consequently, protecting a RF is a topic of significant research interest.

Previous work on protecting RFs can be classified into hardware approaches, software approaches, and hardware-software hybrids. Hardware approaches [Montesinos et al. 2007; Kandala et al. 2007; Blome et al. 2006; Naseer et al. 2006] protect RFs either fully or partially using various mechanisms such as ECC, parity, and duplication, in a way that is completely oblivious to the program running on the processor. Thus software compatibility (i.e., software need not be modified at all in order to protect a RF from soft errors) is a big advantage of the hardware approaches. On the other hand, the additional cost due to the extra circuitry, which is also permanent, can be high. Some techniques in this category (e.g., Montesinos et al. [2007]) advocate the concept of *partially protected register file*, which protects only some of the registers to minimize the hardware overhead. For hardware-based partially protected register files, the decision of which variables to map to the protected registers must be made at runtime using another piece of hardware, which can dissipate a significant amount of power [Lee and Shrivastava 2010]. Hybrid approaches [Yan and Zhang 2005; Lee and Shrivastava 2010], alternatively, expose the hardware protection mechanism to software, so that the decision of which variables to map to protected registers can be made at compile-time, possibly by the compiler. Consequently the program binary must be modified. The hybrid approaches can be quite effective, and more power-efficient than their hardware-only versions. However, they still rely on special hardware such as partially protected RF, which must be supported by the architecture.

Software approaches do not require such special hardware. One way to achieve this is to use some components in the architecture that are already protected. For instance, in an architecture where the L1 cache memory is already protected against soft errors (e.g., via ECC), moving a variable from the RF to the cache will reduce the chance of the variable being corrupted by soft errors. Thus the soft error rate in a register depends on the duration of time in which the register holds a "live" variable, and not just the mere runtime. Based on this intuition, Yan and Zhang [2005] proposes a new instruction scheduling method, which can reduce the soft error rate in RFs by minimizing distances between loads and stores. However, the effectiveness of this method is limited because instruction scheduling cannot see beyond a procedure, where there are greater opportunities as we demonstrate in this article.

To maximize the effectiveness of software-based RF protection, we go one step further. We insert explicit load/store instructions to temporarily write live registers in memory in order to protect them. However, since these explicit memory instructions did not exist in the original program, their existence will increase the application runtime. If the runtime is increased, it can make variables, on average, stay longer in the RF, exposing them to soft errors for a longer duration. Therefore, although we need to evict live variables from the RF for protection, we must not do so overly, to prevent the increased runtime due to extra memory instructions from canceling out the soft error reduction we gain from eviction. Despite these conflicting requirements, our initial investigation into the scope of our software approach shows promise.

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

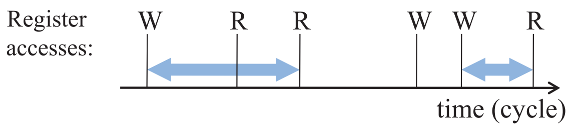


Fig. 1. Vulnerable intervals shown in a register access timing diagram (R is a read, W is a write). Vulnerable intervals are indicated with thick arrows.

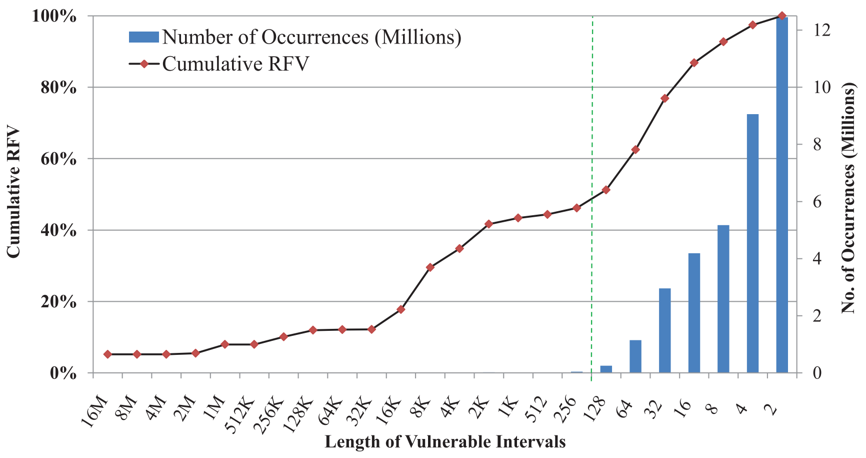


Fig. 2. Histogram of vulnerable intervals showing the scope of compiler approaches. Vulnerable intervals that are 128 cycles or longer contribute about 46% of the total RFV.

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 (see Figure 1). An interval between consecutive accesses to the same register during which the register is vulnerable is defined as *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 temporary 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. In contrast, variables assigned to callee-saved 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 2 plot the histogram of the occurrences of vulnerable intervals in the jpeg application; for instance, the rightmost bar represents the number of vulnerable intervals that are just 1 cycle long. From the graph it is evident that most vulnerable intervals are small. More than 99% of vulnerable intervals are less than 128 processor cycles long. While that is true, a 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 left to right. It shows that the long vulnerable intervals that are 128 cycles or longer contribute about 46% of the RFV in the case of jpeg. 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 a compiler can discover, and if it will be able to protect those registers with minimal code overhead, while maintaining the program correctness.

1.2. Problem and Contributions

One way to reduce RFV is to identify unused registers in heavily executed loops and save/restore them before/after the loops. However, with such an ad-hoc method, it is not only hard to achieve optimal results but it also becomes very cumbersome to handle complex control flow, function calls, and even recursive functions. In this article we approach it 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 expected soft error rate 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 also 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, minimizing their uses, and accurately estimating their effect on performance, code size, and RFV, in the midst of complex control flows and function calls.

In this article we first formulate the entire 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. Using access-free regions, our technique tries to find the best save/restore points by first discovering all the maximal access-free regions through interprocedural analysis, and then selecting the most profitable ones through cost-benefit analysis. Additionally, we perform overhead reduction optimizations to reduce the code size overhead due to our transformation. Furthermore, we present an iterative optimization method based on the Kernighan-Lin's partitioning algorithm [Kernighan and Lin 1970], which can give further reduction in RFV and runtime, as well as help evaluate the effectiveness of our region-based heuristic.

Our experimental results on a number of applications from MiBench benchmark suite [Guthaus et al. 2001] demonstrate that our compiler techniques can effectively reduce the soft error rates in RFs, by up to 66%, or $33 \sim 37\%$ on average, with a small 2% increase in runtime. The soft error rate 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 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. Our iterative optimization method, when applied to the results of our region-based heuristic, can sometimes give additional RFV and runtime reduction, but the improvement is marginal, confirming the effectiveness of our region-based heuristic.

The rest of the article is organized as follows. We discuss related work in Section 2. After formulating the problem in Section 3, we present our region-based heuristic in

Section 4 and our iterative improvement optimization in Section 5. We present our experimental results in Section 6 and conclude the article in Section 7.

2. 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 [Naseer et al. 2006; Blome et al. 2006; Memik et al. 2005; Montesinos et al. 2007; Kandala et al. 2007], which protect registers through 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, several 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 hardware-software hybrids [Lee and Shrivastava 2010; 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 software approaches. 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. There are other software techniques such as code duplication [Oh et al. 2002b; Reis et al. 2005] and control flow checking [Oh et al. 2002a], which either fully or partially duplicate the program code to detect any hardware fault, transient or permanent alike, that may occur at runtime. Thus they are not specifically designed for RF protection nor for soft errors only, but full duplication of the code will be able to detect any error in the RF as well as in other parts of the system, but the overhead is generally high in terms of both code size and performance. On the other hand, partial duplication and control flow checking are able to detect only a subset of errors in the RF.

The concept of vulnerability was first introduced by Mukherjee et al. [2003] as Architectural Vulnerability Factor (AVF), 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 2011].

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 was enough for cost-effectiveness, to treat each register variable differently depending on the length of the *live range* as opposed to the individual vulnerable intervals that make up the live range, in both hardware approaches [Montesinos et al. 2007] and our earlier work on compiler-architecture hybrids [Lee and Shrivastava 2010]. In this article, however, we have to look more finely at each vulnerable interval, possibly treating them differently to maximize the efficiency of our compiler optimization. Our work in this article is based on the RFV representation proposed by Lee and Shrivastava [2011], which decomposes the vulnerability contribution of a basic block into an intrinsic part, which can be determined by the basic block itself, and a conditional part, which depends on other blocks that may follow in the program flow.

3. PROBLEM FORMULATION

3.1. Partitioning Problem

In this section we present an ILP formulation[1] of the entire problem. Specifically we want to find the set of program locations in which to insert save and restore operations so that the RFV reduction is maximized, with minimal code size overhead, under a given performance bound.

—Input: τ (performance tolerance), an optimized program binary
—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.[2] Mode change operations affect the mode of the program execution until the next mode change operation is encountered. Therefore the mode is determined at runtime by the *path* of the program execution and not by the program location. However, this dependence of mode resolution on the execution path makes static analysis as well as guaranteeing program correctness very hard. Thus we require that each program point be associated with only one of the two modes with respect to each register.

This simplification transforms our problem into that of partitioning, where we need to partition all program points into two disjoint groups. To further simplify the problem, let us consider it at the basic block granularity; then we need to partition only basic blocks. As a boundary condition any basic block accessing a register must be mapped to mode A with respect to the register. That way we can ensure that registers are restored to their original state before they are accessed. The other basic blocks, called *access-free blocks*, may be mapped to either mode. Thus we need to find a mapping for access-free blocks, that maximizes the RFV reduction while minimizing the mode change overhead in terms of code size and runtime. Once we find out how to quantify the RFV reduction, the rest is straightforward.

3.2. Quantifying RFV

To perform any compiler optimization for RFV reduction, it is crucial to obtain accurate RFV values, not only for the entire program but also for a part of the code. Following Lee and Shrivastava [2011], the RFV contribution, v_n, of a basic block n with respect to a certain register, can be represented as a simple linear expression, $v_n = v_n^i + v_n^c p_n$, where v_n^i and v_n^c are constants determined by the basic block itself whereas p_n is a variable determined by external factors such as what follows the block n during the execution of the program.

The first term, v_n^i, is called *intrinsic vulnerability* because this vulnerability exists regardless of what may follow the block. For instance, if a basic block contains two

[1]This ILP is not to be confused with a much smaller ILP given in Section 4.5 defined for a selection problem.
[2]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.

accesses to a register, one write followed by one read, the duration of time between the two accesses (i.e., v_n^i) is a vulnerable interval regardless of other blocks, contributing to the vulnerability of the register. However, the duration of time between the read access and the end of the block, which is represented by v_n^c, may or may not contribute to the vulnerability depending on what follows the block, or specifically, on the type of the next access for that register. If it is a read, this interval will become part of a vulnerable interval, contributing to v_n; otherwise, it will not contribute to the vulnerability at all. Thus the second term is called *conditional vulnerability*, and the factor p_n, which is between zero and one, captures how much of it is realized.

Given an execution trace of a program, the factor p_n, called *liveness*, can be calculated for every block. One only needs to count the number of times that the access immediately following the block—however later as it may be—turns out to be a read, and the liveness is the ratio between the read count and the execution count of the block. Very usefully, the RFV calculated thus using the above linear expression is exact, in the sense that it matches the RFV calculated directly from the execution trace. However, without the luxury of an execution trace, which is the case with compiler optimizations, liveness has to be estimated from available information, and can be estimated rather accurately from branch probabilities, either by solving a set of simple linear equations or via a data flow analysis [Lee and Shrivastava 2011].

3.3. ILP Problem

In this formulation we assume the availability of 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. We also use register liveness information, which can be obtained similarly from static analysis or profiling. For runtime overhead and RFV estimation we approximate the number of cycles to the number of executed instructions.

Input:

—τ: runtime tolerance in dynamic instruction count
—$G = (V, E)$: Interprocedural 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 (binary variables):

—x_i^r: denotes the mode of basic block $i \in V$ with respect to register $r \in R$ (1 if mode B)
—$y_{i,j}^r$: if 1, it means that mode change is necessary for register r en route from block i to block j

Objective:

—Code size overhead: $\Delta C = \sum_r \sum_{(i,j)\in E} y_{i,j}^r$
—RFV reduction: $\Delta V = \sum_r \sum_{i\in V} n_i b_i l_i^r x_i^r$
—Objective: maximize $\Delta V - \alpha \Delta C$ for some weighting parameter α

Constraints:

—Runtime overhead: $\Delta T = \sum_r \sum_{(i,j)\in E} m f_{i,j} y_{i,j}^r < \tau$,
where m is a parameter equaling the number of instructions required for one mode change operation,

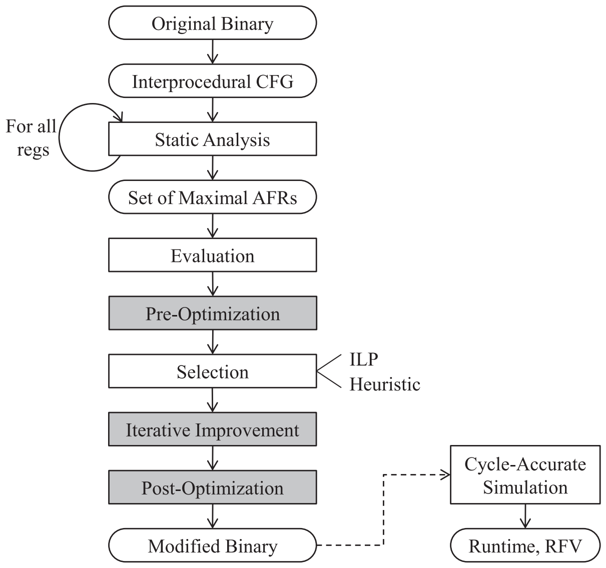


Fig. 3. Overall flow of our technique. Steps in gray are optional.

—Boundary condition: $x_i^r = 0$ for $\forall r \in R, \forall i \in Acc_r$,
where Acc_r is the set of all the basic blocks in which register r is accessed at all
—Mode change condition[3]: $y_{i,j}^r = (x_i^r \neq x_j^r) \wedge (l_i^r \neq 0)$,
where the last term is a constant evaluating to either 0 or 1

This problem formulation can be linearized using a few auxiliary variables. However, solving this ILP may require prohibitive amount of resources as the number of basic blocks is large even for a modest-sized program. Hence we present more scalable solutions in next sections.

4. REGION-BASED HEURISTIC

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 Interprocedural 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 3). 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.

[3]The mode change condition is written so because we must insert a mode change operation on every edge $(i, j) \in E$ where blocks i and j are mapped to different modes, unless the register is statically known to be not alive at the end of block i.

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,[4] 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 within basic blocks, achieves the minimum number of mode changes at runtime. However, implementing a mode change on an edge requires an extra instruction—an unconditional jump—in addition to a store/load instruction, which only is needed if it is implemented within a basic block. 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 partially do in pre- and postoptimizations.

4.3. Interprocedural 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 Algorithm 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

ALGORITHM 1: Find all maximal access-free regions in ICFG

```
1:  input: $ICFG = (V, E)$
2:  output: $AFR := V \rightarrow \{region\_id\}$ : initialized to zero
3:  for all $n \in V$ do
4:     if $AFR[n] \neq 0$ or $n$ is not an access-free-block then
5:        continue to the next iteration
6:     end if
7:     $nodeset \leftarrow \{n\}$
8:     $id \leftarrow id + 1$
9:     repeat
10:       $m \leftarrow$ take one from $nodeset$
11:       $AFR[m] \leftarrow id$
12:       for all $k$ that is a successor or predecessor of $m$ do
13:          if $AFR[k] = 0$ and $k$ is access-free-block then
14:             $nodeset \leftarrow nodeset \cup \{k\}$
15:          end if
16:       end for
17:    until $nodeset$ is empty
18: end for
```

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

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, one can use static RFV analysis [Lee and Shrivastava 2011] to find the register liveness.[5] 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.

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, where n_i, b_i, l_i are the number of instructions, the execution frequency, and the liveness of basic block i, and execution time is approximated to dynamic instruction count. For register liveness l_i, one can use static RFV estimation, which gives a fairly accurate liveness value for each register variable. Alternatively, one can approximate the RFV to $\mu \sum_i n_i b_i$, where μ is the probability of first seeing a read access on exiting the AFR. This probability μ can be computed very accurately, since maximal AFR must neighbor non-access-free blocks. 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 variable 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 formulated as an ILP problem. This ILP problem is much smaller compared to the earlier one in Section 3.3, and can be solved with an ILP solver.

Alternatively, one can use this very simple heuristic algorithm.

(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 Postoptimizations

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, for example, a node has N outgoing edges, and all those edges require mode change operations, we can replace them with a single mode change operation on the node, thereby reducing the number of mode change instructions from $2N$ to 1. Thus the goal of pre- and postoptimizations is to minimize the code size overhead while not increasing the runtime or RFV of application code.

[5]One could avoid static RFV analysis by conservatively assuming that an incoming edge is live unless all the outgoing edges are not live.

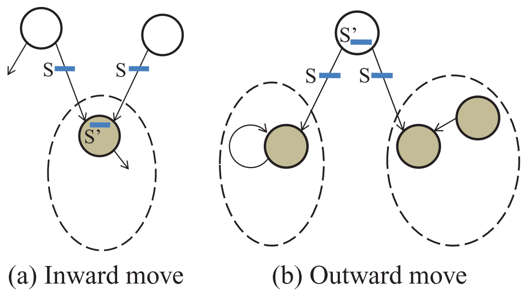


Fig. 4. Moving mode change operations from edge to node for save operations. Solid circles represent nodes, or basic blocks (white: mode A, gray: mode B), and dashed ovals represent selected maximal AFRs. Thick small bars represent save operations (S: before move, S': after move).

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 4 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 (postoptimization).

5. ITERATIVE IMPROVEMENT

5.1. Purpose

While our region-based heuristic is able to find good partitions for a given ICFG, its granularity is limited to *regions* and thus cannot explore opportunities that may come from local exchanges of nodes along the region boundaries. On the other hand, 2-set partitioning problems have long been studied in the VLSI domain, where the most well-known algorithm is the Kernighan and Lin's (KL) partitioning heuristic [Kernighan and Lin 1970]. Thus we apply the KL heuristic to improve the solutions found by our region-based heuristic. Our purpose in this is two-fold: (i) the KL heuristic is fast, and therefore can be added to our optimization flow as an *iterative* improvement pass (see Figure 3), and (ii) by trying an optimization at a finer granularity we can assess the validity of the assumption behind our region-based heuristic.

5.2. Adaptation of KL Algorithm

5.2.1. KL Algorithm. As illustrated in Figure 5, the main steps of the KL algorithm consist of N-pair selection and K-pair selection, where N is the size of each of the two equal-sized partitions given as input, and K is the output of the K-pair selection step. The N-pair selection step determines the sequence of locally-best-looking exchanges between the two partitions, also calculating the additional gain by each exchange. The K-pair selection determines a partial sequence, defined by the *first* K exchanges from the sequence, that maximizes the cumulative gain, thus avoiding the trap of local optima.

5.2.2. Challenges and Solution. While applying the KL algorithm directly to our original problem poses several difficulties (i.e., unequal partition sizes, extra constraints such as runtime overhead (ΔT) constraint and some blocks being fixed to one partition), we can avoid most of them by applying the KL algorithm only to improve the results of our AFR heuristic. However, there is still one more challenge—evaluating the optimization

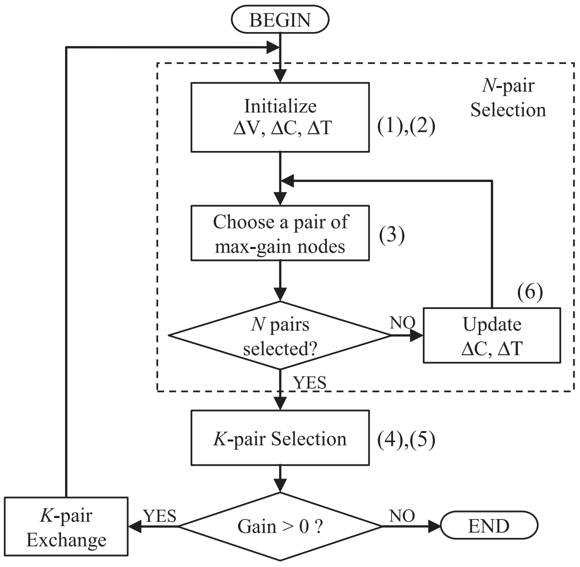


Fig. 5. Our partitioning algorithm based on Kernighan and Lin [1970]. Numbers next to boxes are the corresponding equations.

objective is much more complicated than in the case of the KL algorithm. In the KL algorithm the objective, which is the sum of the weights of all edges between two partitions, can be evaluated very easily, and calculating the gain of any single move is trivial. In our problem, on the other hand, the objective, which is a weighted sum of RFV reduction and code size increase, is by itself much more complicated, and moreover calculating the gain of a move is even more puzzling, since it is determined not only by the characteristics of the nodes involved but also by those of their neighbors and arbitrarily many descendants that may follow. We solve this issue by precalculating the register liveness values of all blocks using the static RFV estimation [Lee and Shrivastava 2011] and by keeping track of the gain of every potential move as follows.

With every node, a, of an ICFG we associate the gain of moving the node to the other set. To do this efficiently we maintain three arrays, ΔV, ΔC, ΔT, representing the changes in RFV, code size, and runtime, respectively. The size of each array is equal to the number of nodes. Initialization of these arrays can be done as (1) and (2) (computing ΔT is very similar to computing ΔC, and not shown here), provided that node a initially belongs to set A. If the node belongs to set B, all three terms have to be negated.

$$\Delta V_a = n_a b_a l_a^r \tag{1}$$

$$\Delta C_a = \sum_{i \in A} Y_{a,i} + Y_{i,a} - \sum_{j \in B} Y_{a,j} + Y_{j,a}, \tag{2}$$

where $Y_{i,j}$ is either m (the number of instructions for one mode change operation) if there exists an edge $(i, j) \in E$ and l_i^r is nonzero, or zero otherwise. Then the gain of a move and that of an exchange can be computed as follows.

$$G_a = \Delta V_a - \alpha \Delta C_a \tag{3}$$

$$G_{a,b} = G_a + G_b - \epsilon_{a,b} \tag{4}$$

$$\epsilon_{a,b} = 2\alpha(Y_{a,b} + Y_{b,a}), \tag{5}$$

where $\epsilon_{a,b}$ is the common portion between G_a and G_b.

Now whenever we choose a move with the greatest gain during the N-pair selection, we update the gains of all the affected nodes. This way we can reduce the complexity of the algorithm greatly, compared to evaluating the gains of all the nodes in every iteration of the N-pair selection. The RFV part of the gain need not be changed, however[6]; only ΔC and ΔT parts do.

$$\Delta C_i^{new} - \Delta C_i = -2(Y_{i,a} + Y_{a,i}) + 2(Y_{i,b} + Y_{b,i}) \tag{6}$$

where $a \in A$ and $b \in B$ are the pair of nodes chosen to be exchanged. This update must be applied to every neighbor of a in set A. For neighbors in set B, the right-hand side has to be negated.

5.2.3. Complexity. The time complexity of the N-pair selection step in Figure 5 is $O(N^2 + |E|)$, where N is the number of access-free blocks in a set and E is the set of edges in the ICFG. This is because the inner loop is repeated N times, and choosing a pair of max-gain nodes requires $O(N)$.[7] Since we expect the initial solution to be close to optimal, the K value will be small, and therefore we limit N to 1000 pairs in our experiments.

5.3. Implementation Issues

The KL algorithm deals with equal-sized partitions, only swapping a pair of nodes from both partitions to minimize the cut. However, our problem does not require maintaining equal-sized partitions, nor is it desirable to do so. In fact, it can be beneficial to allow unequal-sized partitions because the vulnerability reduction in our problem can be higher if we allow more blocks in the protected partition (set B). Thus in addition to swapping nodes, which is called *symmetric*, we allow the algorithm to use *asymmetric* moves.

In the asymmetric variant of the algorithm we choose a single node that maximizes the gain as per (3). Since there are two directions (from A to B, and from B to A), we select the maximum-gain node for each direction, and finally choose the one with the higher gain.

6. EXPERIMENTS

6.1. Experimental Setup

To evaluate the effectiveness of our proposed techniques we use applications from MiBench benchmark suite [Guthaus et al. 2001]. We compile applications using GCC 2.7.2.3, which is one of the latest versions supporting the SimpleScalar target, with the optimization level specified in the benchmark suite (typically `-O3` but `-O4` in the case of susan). For runtime and RFV values we run simulation using the SimpleScalar cycle-accurate simulator [Austin], which models an in-order single-issue processor with separate 4-way set-associative 16 KB L1 caches and a unified 4-way 256 KB L2 cache. The proposed compiler optimization is implemented as a postlink 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

[6]The RFV reduction (ΔV) of a node should be changed too if the node moves to the other set. However, we do not need to update the change during the N-pair selection, since the same node cannot be selected again, and one node moving to the other set cannot affect the RFV of another node, as indicated by (1), and also because the ΔV array will be initialized again at the beginning of the next N-pair selection loop (see Figure 5).
[7]The time complexity of the N-pair step could be lowered to $O(|E|logN)$ if one uses a priority queue instead of a vector to store gains.

an application basis.[8] For weighting parameter α, we use $0.5V_o/C_o$, where V_o and C_o are the RFV (in cycle-words) and code size (in words) of the original program, respectively.

Our main comparison is between three cases: *Global-gp*, *Global-r0*, and *Naive*. The first two are our proposed techniques based on AFR (Access-Free Region) heuristic, which is applied to interprocedural CFGs. To implement mode change operations, global-gp uses the gp register as the base address register. The global-r0 scheme is the same as the global-gp, except that the r0 register is used instead of r0 register, which allows even gp to be protected. In both cases the r0 register is assumed to be hard-wired to zeros, therefore not needing protection.[9] For comparison we also implement an intra-procedural optimization based on our proposed technique, which is called naive approach. We use exactly the same flow as shown in Figure 3 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 Algorithm 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 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.

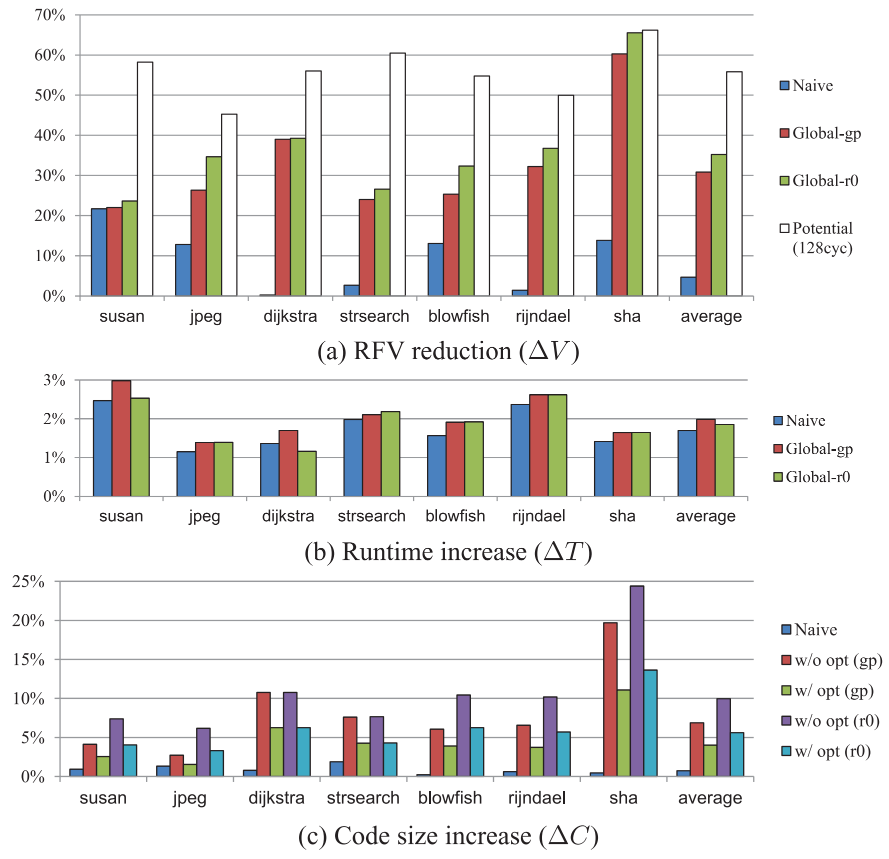
6.2. Effectiveness of Interprocedural Analysis

First we compare our proposed techniques with the naive 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. 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 128 cycles long. For the selection algorithm we use our simple heuristic rather than the ILP formulation, but the effect of using ILP is discussed 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 56%, and up to 66% in the case of sha. Our technique can realize a considerable part of it, achieving significant RFV reduction of $33 \sim 37\%$ on average, and up to 66% if we use r0 as the base register. 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 suggests that

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

[9]Hard-wired registers such as r0 are not universally available among processors. While the global-r0 scheme shows greater RFV reduction as can be seen in our experimental results, the global-gp scheme should be applied for architectures without such registers.



(a) RFV reduction (ΔV)

(b) Runtime increase (ΔT)

(c) Code size increase (ΔC)

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

Global-gp musters more smaller access-free regions to substitute for the missing larger access-free regions from gp.

The RFV reduction by the naive 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.[10]

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 naive 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.

[10]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.

Table I.
Comparing ILP vs. Heuristic for solving the selection problem (Global-gp, geometric mean for all applications).

	RFV reduction	Runtime increase	Code size overhead		Time (sec)
			w/o opt	w/ opt	
ILP	30.5%	2.05%	7.19%	5.32%	<1
Heuristic	30.9%	1.99%	6.87%	4.03%	<1

Table II. Number of AFRs Generated vs. Selected (Global-gp case)

Application	#Blocks	#AFRs	#Selected	#Regs
susan	3737	4425	8	8
jpeg	6448	9465	45	19
dijkstra	3337	3404	8	8
strsearch	2611	3094	10	9
blowfish	2574	3216	8	7
rijndael	2743	3123	11	10
sha	2564	2787	24	12

6.3. ILP vs. Heuristic

In Section 4.5 we present two methods to select the best set of maximal AFRs, a small ILP formulation (not to be confused with the ILP formulation of Section 3.3) and a simple heuristic algorithm, which are compared in Table I. For this comparison we use gp as the base address register. Again, the RFV and runtime numbers are from actual simulations, geometrically[11] 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 workstation with 2GHz Intel Xeon processor (single threaded execution) with 4GB DRAM.

6.4. Access-Free Regions

Table II compares the number of AFRs generated vs. selected for each application, in the case of Global-gp. The second column (#Blocks) lists the number of basic blocks in each application, which roughly matches application sizes. The number of AFRs, shown in the third column, can be greater than that of basic blocks, since AFRs are generated separately for each register. The fourth column is the number of selected AFRs, which is much less than that of AFRs generated. In addition the number of unique registers selected for protection by our technique is shown in the last column. Since only a few dozens of AFRs can be selected for protection in each application, deciding which register to protect becomes an important issue. Our proposed technique can handle it elegantly by first creating a pool of AFRs for all registers and then selecting the most

[11]In an earlier version [Lee and Shrivastava 2009] we used arithmetic mean, which gives slightly different numbers.

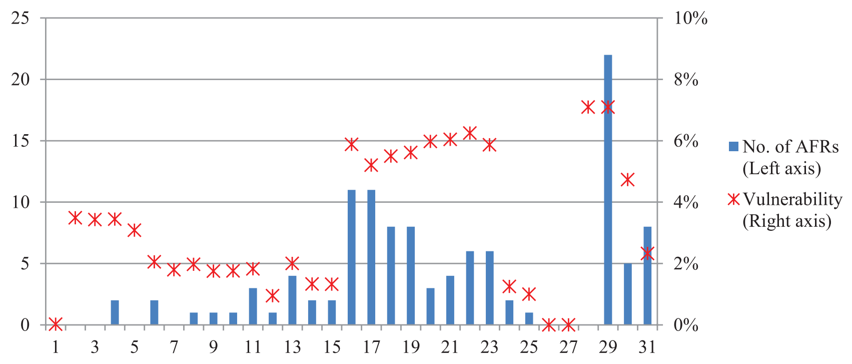


Fig. 7. Number of selected AFRs for each register (Global-gp case, accumulated for all applications), overlaid with the vulnerability distribution across registers.

profitable ones. Contrastingly, in the partitioning approach, selection decisions have to be made individually for each register, which can be a big disadvantage.

Figure 7 shows which registers are most often protected, along with the vulnerability distribution across registers. In the graph, bars represent the number of selected AFRs associated with each register (the left axis), asterisks the vulnerability of each register (the right axis), and the x-axis the register number. The number of selected AFRs is cumulative for all the applications, taken from the Global-gp results. Thus the sum of heights of all the bars in the graph matches the sum of all the numbers in the fourth column of Table II. We first observe that there is some correlation between the vulnerability and the number of times a register is protected. However, other than the global pointer (`r28`), which obviously cannot be protected by the Global-gp scheme, there are some registers (e.g., r2, r3, r5) that are not protected despite their significant vulnerability. This is due to their frequent use in the program as carriers of argument values and return values, which are typically short-lived. On the other hand, the stack pointer (`r29`) and return address register (`r31`) are among the most frequently protected relative to the vulnerability.

6.5. Sensitivity on Weighting Parameter

In our experiments we set the weighting parameter α to $0.5V_o/C_o$, where V_o and C_o are the RFV and code size of the original program, respectively. To see the effect of α we vary it from $10V_o/C_o$ to $0.001V_o/C_o$. Figure 8 shows the results of this exploration, averaged for all applications. As expected, the greater the value of α is, the lower the code size overhead, but at the same time the reduction in RFV is also diminished. On the other hand, if we decrease the value of α, both the RFV reduction and the code size overhead increase, but after a certain point it reaches a plateau. We believe that this is because our constraint on runtime overhead also constrains the code size overhead. Therefore even if one sets α to a very small number, the code size overhead is still constrained. However, since there is a limit to reducing vulnerability as well, too small a value of α seems to only increase the code size with very little benefit in RFV. From the graph the range of usable values for α seems to be between $0.5V_o/C_o$ and $2V_o/C_o$.

6.6. RFV Distribution

Figure 9 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

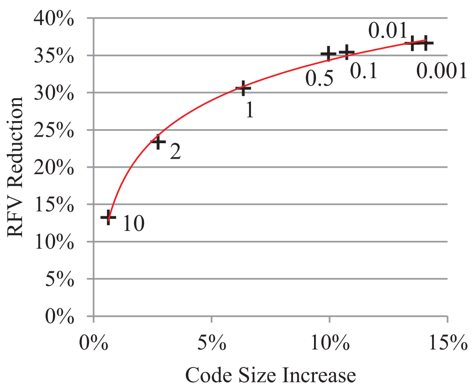


Fig. 8. Effect of α, varied from $10V_o/C_o$ to $0.001V_o/C_o$ (Global-r0 case, geometric mean of all applications).

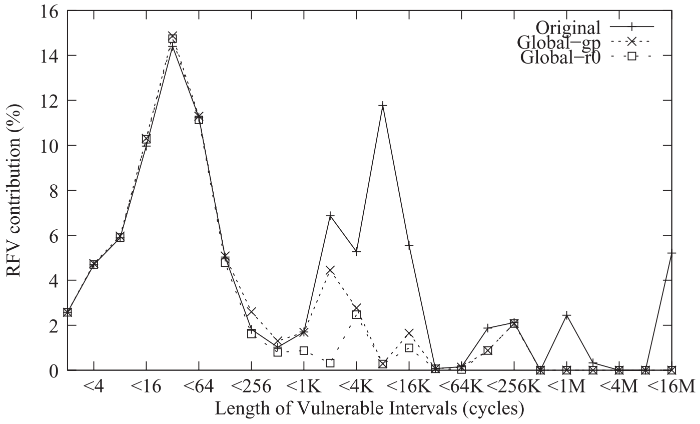


Fig. 9. Comparing RFV distributions before/after the proposed optimization.

and modified binaries of jpeg. The graph shows that our methods can successfully filter 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.

6.7. Partitioning Approach

Though our experimental results so far demonstrate that our selection heuristic is as good as the ILP method, they only address the problem of what to select from a given set of access-free regions. Whether the given AFRs are good candidates for protection to begin with is unanswered. On the other hand, the KL partitioning approach has the potential to improve the quality of partitioning by directly selecting basic blocks to maximize a given objective function. To assess the effectiveness of our AFR heuristic, we implement the KL partitioning algorithm as described in Section 5, and compare the two methods using the same setting. For the KL algorithm we evaluate two variants, one that allows asymmetric moves (referred to as *Asymmetric KL*) as well as one that always exchanges two basic blocks from different partitions (*Symmetric KL*). Asymmetric moves may be beneficial, since the more blocks in the protected partition, the higher the vulnerability reduction will be. In all the cases, register 0 (r0) is used as the base address register for mode change operations.

Table III. Comparing AFR-based vs. KL-based heuristics

Application	ΔV (%)			ΔT (%)		
	AFR only	Symmetric KL	Asymmetric KL	AFR only	Symmetric KL	Asymmetric KL
susan	23.7	24.2	24.2	2.53	2.55	2.56
jpeg	34.7	35.6	35.2	1.39	1.39	1.39
dijkstra	39.3	39.3	39.3	1.16	1.13	1.13
strsearch	26.6	26.8	26.7	2.18	1.93	2.08
blowfish	32.3	32.3	32.3	1.92	1.92	1.92
rijndael	36.8	37.1	36.8	2.62	2.65	2.62
sha	65.6	65.5	65.6	1.65	1.65	1.65
Mean	35.2	35.5	35.4	1.85	1.82	1.83

Table III shows the results in terms of RFV reduction (ΔV) and runtime increase (ΔT). Here the numbers are the differences (in percent) from the baseline, which is for the original program. We first note that the results of the KL partitioning methods should not be worse than those of the AFR method (which is the case with our data), since the partitioning methods take the results of the AFR method as their initial solution. A close look at the data reveals that while there are differences among the applications, on average the KL methods generate slightly better results in terms of both RFV reduction (larger) and runtime increase (smaller). The differences between Symmetric and Asymmetric variants of the KL method are, on average, negligible.

Nevertheless the results of the AFR method on the whole are very similar to those of the KL methods. To be fair, the KL methods we implement are very quick, finishing within a few seconds at most, and unlike simulated annealing, the KL methods accept a change only if the cumulative gain is positive. However, given that the KL methods too are designed to avoid the trap of local optima, and that they are not based on the notion of AFR at all (therefore they are free to explore partitions involving non-AFRs), the fact that numerous node exchanges can bring in so small an improvement suggests that our AFR method is indeed quite good, at least as compared to the KL algorithm.

7. CONCLUSION

In this article we presented a case for a compiler approach to reducing soft errors in register files of embedded processors. 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 (AFRs). 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. In addition we presented an iterative improvement optimization based on the Kernighan-Lin partitioning algorithm, which confirms the effectiveness of our AFR heuristic.

REFERENCES

Blome, J. A., Gupta, S., Feng, S., and Mahlke, S. 2006. Cost-efficient soft error protection for embedded microprocessors. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '06)*. 421–431.

Chen, K., Malik, S., and August, D. I. 2001. Retargetable static timing analysis for embedded software. In *Proceedings of the 14th International Symposium on System Synthesis (ISSS '01)*. 39–44.

DODD, P. AND MASSENGILL, L. 2003. Basic mechanisms and modeling of single-event upset in digital microelectronics. *IEEE Trans. Nucl. Sci. 50*, 583–602.

FAZELI, M., AHMADIAN, S. N., AND MIREMADI, S. G. 2008. A low energy soft error-tolerant register file architecture for embedded processors. In *Proceedings of the 11th IEEE High Assurance Systems Engineering Symposium*. IEEE, 109–116.

GUTHAUS, M., RINGENBERG, J. S., ERNST, D., AUSTIN, T. M., MUDGE, T., AND BROWN, R. B. 2001. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the International Workshop on Workload Characterization*. 3–14.

HARROLD, M., ROTHERMEL, G., AND SINHA, S. 1998. Computation of interprocedural control dependence. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*. 11–20.

KANDALA, M., ZHANG, W., AND YANG, L. 2007. An area-efficient approach to improving register file reliability against transient errors. In *Proceedings of the International Conference on Advanced Information Networking and Applications Workshops*. 798–803.

KERNIGHAN, B. AND LIN, S. 1970. An efficient heuristic procedure for partitioning graphs. *Bell Syst. Tech. J. 49*, 291–307.

LEE, J. AND SHRIVASTAVA, A. 2009. A compiler optimization to reduce soft errors in register files. *ACM SIGPLAN Noti. 44*, 7, 41–49.

LEE, J. AND SHRIVASTAVA, A. 2010. A compiler-microarchitecture hybrid approach to soft error reduction for register files. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 29*, 7, 1018–1027.

LEE, J. AND SHRIVASTAVA, A. 2011. Static analysis of register file vulnerability. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 30*, 607–616.

LI, X., ADVE, S. V., BOSE, P., AND RIVERS, J. A. 2008. Online estimation of architectural vulnerability factor for soft errors. *SIGARCH Comput. Archit. News 36*, 3, 341–352.

MEMIK, G., CHOWDHURY, M. H., MALLIK, A., AND ISMAIL, Y. I. 2005. Engineering over-clocking: reliability-performance trade-offs for high-performance register files. In *Proceedings of the International Conference on Dependable Systems and Networks*.

MITRA, S., SEIFERT, N., ZHANG, M., SHI, Q., AND KIM, K. S. 2005. Robust system design with built-in soft-error resilience. *IEEE Computer 38*, 2, 43–52.

MONTESINOS, P., LIU, W., AND TORRELLAS, J. 2007. Using register lifetime predictions to protect register files against soft errors. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '07)*. 286–296.

MUKHERJEE, S. S., WEAVER, C., EMER, J., REINHARDT, S. K., AND AUSTIN, T. 2003. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the International Symposium on Microarchitecture*.

NASEER, R., BHATTI, R. Z., AND DRAPER, J. 2006. Analysis of soft error mitigation techniques for register files in IBM cu-08 90nm technology. In *Proceedings of the 49th IEEE International Midwest Symposium on Circuits and Systems (MWSCAS '06)*. 515–519.

OH, N., SHIRVANI, P. P., AND MCCLUSKEY, E. J. 2002a. Control-flow checking by software signatures. *IEEE Trans. Reliab. 51*, 111–122.

OH, N., SHIRVANI, P. P., AND MCCLUSKEY, E. J. 2002b. Error detection by duplicated instructions in super-scalar processors. *IEEE Trans. Reliab. 51,* 63–75.

REIS, G. A., CHANG, J., VACHHARAJANI, N., RANGAN, R., AND AUGUST, D. I. 2005. SWIFT: Software implemented fault tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization*. 243–254.

WALCOTT, K. R., HUMPHREYS, G., AND GURUMURTHI, S. 2007. Dynamic prediction of architectural vulnerability from microarchitectural state. *SIGARCH CA News 2*, 516–527.

WU, Y. AND LARUS, J. 1994. Static branch frequency and program profile analysis. In *Proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture*. 1–11.

YAN, J. AND ZHANG, W. 2005. Compiler-guided register reliability improvement against soft errors. In *Proceedings of the International Conference on Embedded Software (EMSOFT '05)*. 203–209.

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

Received February 2012; revised October 2012; accepted January 2013