Static Analysis to Mitigate Soft Errors in Register Files

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

***Abstract*—With continuous technology scaling, soft errors are becoming an increasingly important design concern even for earth-bound applications. While compiler approaches have the potential to mitigate the effect of soft errors with minimal runtime overheads, static vulnerability estimation—an essential part of compiler approaches—is lacking due to its inherent complexity. This paper presents a static analysis approach for Register File (RF) vulnerability estimation. We decompose the vulnerability of a register into intrinsic and conditional basic-block vulnerabilities. This decomposition allows us to develop a fast, yet reasonably accurate, linear equation-based RF vulnerability estimation mechanism. We demonstrate its practical application to compiler optimizations. Our experimental results on benchmarks from MiBench suite indicate that not only our static RF vulnerability estimation is fast and accurate, but also compiler optimizations enabled by our static estimation can achieve very cost-effective protection of register files against soft errors.**

I. Introduction

Due to continuous technology scaling, soft errors—transient faults mainly caused by high-energy particles—are becoming an important design concern for earth-bound embedded applications in addition to space applications [1]. Traditionally, due to their large size, only large memory structures like the main memory and caches were considered important for protection against soft errors. However, recently, Blome et al. [2] observed that majority of the faults both in combinational and sequential logic that affect the architectural state of a processor comes 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 main memory are routinely protected using parity or Error Correcting Codes (ECC) [3], protecting the Register File (RF) using such schemes is prohibitive, because RF is often i) in the timing-critical path [4] of the processor, and ii) is one of the hottest blocks on the chip [5]. Protecting RF is challenging, since to minimize impact on performance, RF protection scheme should operate in parallel to the normal execution, and at minimal power overhead. Keeping the RF cool is important not only since reliability is negatively affected by high temperature [6], [7] but also to avoid performance loss due to thermal degradation [8], [9]. Consequently, protecting RFs is a topic of significant research interest.

Several approaches for protecting RFs have been proposed, but most of them are microarchitectural solutions [2], [10]–[12]. Of them, cost effective techniques implement some form of *Partial Protection* of the RF. They take advantage of the fact that not all registers hold useful data at all times, therefore protecting only a part of the RF may result in high protection at low power overheads. However, since the microarchitectural techniques make the decision of which register to protect in the hardware at runtime, they necessarily incur high power overheads.

Potentially more interesting is the compiler approach [13], [14]. Compilers can mitigate the effect of soft errors in RF without any hardware support, e.g., shortening the average live range of variables through memory spills may reduce the soft error rate of a register file if the L1 data cache is already protected, as transient faults occurring to a register containing no live variable cannot cause an error. In addition, compilers can be extremely valuable in enhancing the effectiveness of hardware support. For instance, the decision of which variables to protect in a partially protected RF scheme can be made by compiler, completely eliminating the power overhead in decision. In addition, since the compiler has the whole program information, it can make better decisions.

Essential to all such compiler techniques is a method to estimate the Register File Vulnerability (RFV) *of a program to soft errors.* The concept of RFV comes from Architectural Vulnerability Factor (AVF) [15]. A register is vulnerable only if it will be read by the processor, and is not vulnerable if its value will be overwritten. The RF vulnerability of a program is the sum of vulnerability of all registers during the program execution. The challenge lies in the fact that there are no known methods to statically estimate the vulnerability of programs. Existing techniques can compute RFV only through simulation, which is clearly not useful for compiler optimizations.

This paper proposes a static analysis approach to estimate RFV. Static estimation of vulnerability is more challenging than performance estimation. This is because while performance is dependent on branch probabilities, the vulnerability of a register is dependent on the execution path. The central idea in this paper is to use a linear function representation for the vulnerability of a basic block, which is otherwise very difficult to capture accurately. Our approach breaks the problem into i) Computing the vulnerability of a register as a function of *block post-condition*, and ii) Estimating block post-condition from branch probabilities. Block post-condition of a register is the probability of the next access to the register after this block being a read, and can be estimated from branch probabilities with reasonable accuracy. Moreover, computing vulnerability from block post-conditions can be done exactly and very efficiently. Thus our static

This work was partially supported by grants from Raytheon, Science Foundation of Arizona (SFAz), and Stardust foundation.

978-3-9810801-5-5/DATE09 © 2009 EDAA

vulnerability estimation can be fast while being reasonably accurate. Our experimental results on a number of embedded applications indicate that not only our static RF vulnerability estimation is fast and relatively accurate but also compiler optimizations enabled by our static estimation can achieve very cost-effective RF vulnerability reduction. When used to compile for partially protected RF, simplest of compiler-management schemes can be much more energy efficient than hardware schemes, and that explicit optimization can further reduce the energy overhead by up to 50% to 75% while not sacrificing vulnerability.

II. Related Work

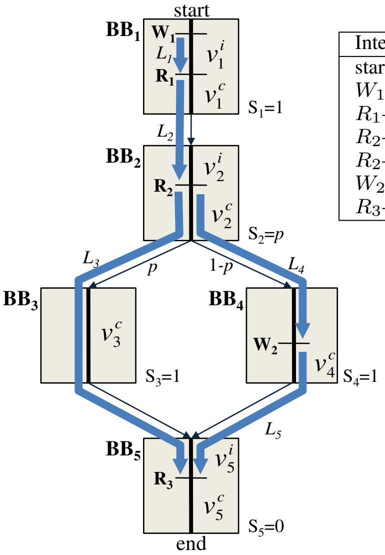
Existing RF protection mechanisms can be broadly classified into hardware and software techniques. Several hardware techniques exist, and are based on using ECC, parity checking, and duplication. The IBM G5 enterprise server has every register and latch protected by either ECC or parity [16]. Since protecting the entire RF has extremely high overheads, cost-effective solutions [2], [10], [11] protect only part of the RF. This is based on the observation that not all register values are always vulnerable; consequently, cost-efficiency can be achieved by protecting only part of the RF and dynamically choosing register variables to be protected. However, all these microarchitectural techniques necessarily incur power overhead associated with the dynamic decision logic. In this paper, we develop compiler techniques to enhance the effectiveness of hardware techniques for RFV optimization, including elimination of the power overhead of runtime prediction.

We are aware of only two compiler techniques [13], [14], both of which are based on partially protected register files and try to reduce RF vulnerability by different register allocation. While effective, those techniques are profile based, and is therefore extremely limited in application. In addition to the long time required by profile based methods, there are challenges in even obtaining a representative profile. While compilers can reduce the RF vulnerability with or without hardware support, any such compiler technique requires static estimation of RF vulnerability, and to date there is no method to do it; and that is the topic of this paper.

The concept of vulnerability was first introduced as Architectural Vulnerability Factor (AVF) by Mukherjee et al. [15], which is a quantitative measure of the amount of "live" information that needs protection of each microarchitectural component. Techniques for runtime estimation of AVF [17], [18] were proposed, which was then used to adjust the protection level of thread-level duplication techniques to the varying workload vulnerability. RF has very high AVF, and RF vulnerability computation is typically done in simulation, and there is no static method to estimate vulnerability of register files.

III. RF Vulnerability Computation

Vulnerability of a register is defined as the total time for which it is vulnerable, or holds useful data. The Register File Vulnerability (RFV) is then just the sum of the vulnerability of all the registers in the RF. Figure 1 illustrates a Control Flow Graph (CFG), where nodes are basic blocks and edges represent control flow. The accesses to a register are marked by W_i or R_i depending on whether it is written or read, respectively. The reads and writes divide the execution flow into intervals, which are denoted by L_i's and bold arrows in the figure. The embedded table lists all the intervals and their vulnerability. The total vulnerability is then their summation: $V = L_1 + L_2 + pL_3 + (1-p)L_5$, where p is the probability that BB$_3$ is executed after BB$_2$. However, in general this problem becomes difficult, as not only the number of intervals grow exponentially with the number of branch nodes, but it also becomes difficult to compute the execution probability of an interval, as it span several basic blocks. The presence of loops in the program makes this problem intractable.



Interval	Freq.	Vulnerab.
start–W_1	1	0
W_1–R_1	1	L_1
R_1–R_2	1	L_2
R_2–R_3	p	pL_3
R_2–W_2	$1-p$	0
W_2–R_3	$1-p$	$(1-p)L_5$
R_3–end	1	0

Fig. 1. Computing RF vulnerability from intervals.

One way to break away from this path-dependence of vulnerability calculation is to attribute vulnerabilities to basic blocks, instead of intervals, so that the total RFV can be computed simply by adding up the basic block vulnerabilities. Similar approach is also used in, for instance, static estimation of performance; the runtime of each basic block is first estimated, and the total runtime is then simply the summation of the cycle count of each basic block, weighted by the execution frequency of each basic block.

Unfortunately the similarity with performance estimation ends here. While the runtime of a basic block is estimated as a constant, and not dependent on other basic blocks, the vulnerability of a variable in the basic block necessarily depends on what happens with the register in the following basic blocks. Consider the basic block BB$_2$ for example. The interval from the start of BB$_2$ to R_2 is definitely vulnerable, since this interval ends in a read. However, whether the interval from R_2 to the end of BB$_2$ is vulnerable may be determined immediately in the next basic block, as is the case with the path BB$_2$–BB$_4$, or may be determined several basic blocks down the flow, as is the case with the path BB$_2$–BB$_3$–BB$_5$. This inherent context dependence of vulnerability computation prevents us from assigning one number to basic block vulnerability.

IV. New Vulnerability Representation

To break this context dependence, we represent the basic block vulnerability of a register as a linear function $v^i + v^c s$, where v^i and v^c are constants derived from the basic block itself while variable s, called *post-condition*, is the probability summarizing the contextual information. This is essentially decomposing the vulnerability into *intrinsic vulnerability*, v^i, which definitely contributes to vulnerability, and *conditional vulnerability*, v^c, which does so conditionally depending on the following blocks. v^i is computed as the average combined length of read-finished intervals within the basic block (here an interval is between register accesses or basic block boundary), and v^c is the average length of the last interval. These lengths are in time or in cycles, and therefore exact computation of these quantities, even at the basic block level, requires microarchitectural knowledge, for which static estimation techniques such as [19], [20] may be used.

Unlike v^i and v^s, which are determined from their own basic blocks, post-condition s is determined from other basic blocks. Once s is known either through profiling or static analysis, the total RFV can be easily computed as $\sum_j f_j V_j = \sum_j f_j (v^i_j + v^c_j s_j)$, where f_j and V_j are the execution frequency and vulnerability of j-th basic block, respectively, and v^i_j, v^c_j, and s_j are also of the j-th basic block. In our example the vulnerabilities of the five basic blocks are: $V_1 = v^i_1 + v^c_1 \cdot 1$, $V_2 = v^i_2 + v^c_2 \cdot p$, $V_3 = 0 + v^c_3 \cdot 1$, $V_4 = 0 + v^c_4 \cdot 1$, and $V_5 = v^i_5 + v^c_5 \cdot 0$. And the total vulnerability is given as below, which is equal to the earlier formula after replacing L_i's.

$$V = (v^i_1 + v^c_1) + (v^i_2 + v^c_2 p) + p\, v^c_3 + (1-p) v^c_4 + v^i_5$$

The linear function representation of basic block vulnerability is exact for any value of s, and the computed total vulnerability is accurate as long as the post-conditions are accurate.[1] Now the problem is reduced to finding the probability s, which is the topic of the next section.

V. Estimating Post-conditions

Post-condition is the probability of the next access being a read. Consider a CFG in Figure 2, which includes a loop. Each node is annotated with *first-access-type* attribute (inside small boxes), which is either read ('r'), write ('w'), or no-access ('–'). Assume that the CFG is repeated twice and the branch probabilities are as shown in the figure. Interestingly, while the execution frequencies of nodes and edges (numbers in parentheses) can be easily found out from branch probabilities, post-conditions cannot be determined from branch probabilities alone. The figure lists two execution paths that result in the same branch probabilities. In both the cases, execution frequencies of the nodes and edges are exactly the same. However, post-conditions of some basic blocks are different, as shown in Table I. This is because post-condition depends on execution paths rather than mere execution counts.

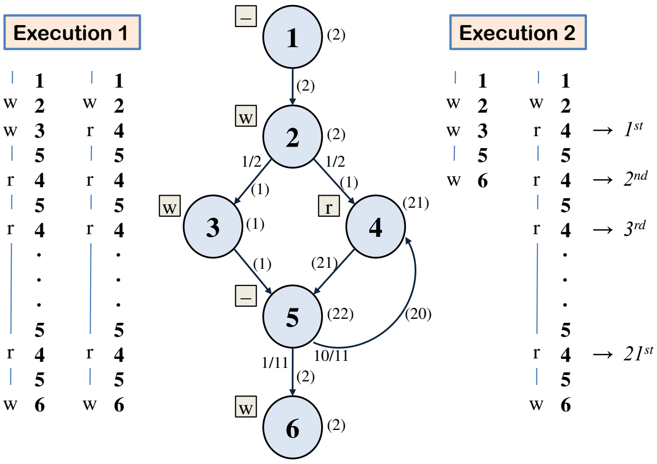

Execution 1
| 1 w 2 w 3 | 5 r 4 | 5 r 4 · · · 5 r 4 | 5 w 6
| 1 w 2 r 4 | 5 r 4 | 5 r 4 · · · 5 r 4 | 5 w 6
–
1
(2)
(2)
w
2
(2)
1/2
1/2
(1)
(1)
w
3
(1)
r
4
(21)
(1)
(21)
–
5
(22)
(20)
1/11
10/11
(2)
w
6
(2)
Execution 2
| 1 w 2 w 3 | 5 w 6
| 1 w 2 r 4 | 5 r 4 | 5 r 4 · · · 5 r 4 | 5 w 6
→ 1st
→ 2nd
→ 3rd
→ 21st

Fig. 2. Two possible executions with the same execution frequency but different post-conditions (see Table I).

Post-condition is the probability that a node with the read first-access-type will be visited prior to any node with the write first-access-type. Therefore, using the conditional probability $P(a|q)$, which is the probability of visiting node a right after path q, one can write $s_3 = P(5|3)P(4|3,5)$ and $s_4 = P(5|4)P(4|4,5)$. Since $P(5|3) = P(5|4) = 1$, we have $s_3 = P(4|3,5)$ and $s_4 = P(4|4,5)$. Now the branch probability at BB$_5$ specifies $P(4|5) = 10/11$ only, but not $P(4|3,5)$ nor $P(4|4,5)$, which is why we can have different post-conditions from the same branch probabilities, namely, lack of enough information. To obtain these probabilities from profiling, for instance, we need to find the frequencies of length-2 paths such as $(3,5,4)$ and $(4,5,4)$, since $P(4|3,5) = P(3,5,4)/P(3,5) = N(3,5,4)/N(3,5)$, where $P(q)$ and $N(q)$ are the probability and the frequency of path q, respectively. In the most general case, however, the paths we need to consider can extend to the earliest join node, and the number of different paths will be exponential in the number of join nodes, even without loops. This is clearly unscalable. As a practical solution we can approximate length-n conditional probabilities with length-m conditional probabilities, where $m < n$, or simply with length-1 conditional probabilities, that is, branch probabilities. Approximation with branch probabilities gives $P(4|3,5) \approx P(4|4,5) \approx P(4|5)$. We generalize this and later present our linear equation-based method.

It is worth noting that in this particular example we can find the two illustrated solutions entirely from the branch probabilities. Though length-2 conditional probabilities are not specified, there is a constraint on them, i.e., $N(3,5,4) + N(4,5,4) = N(5,4)$, since BB$_5$ has only two immediate

TABLE I
Different post-conditions

Node	Execution 1	Execution 2	Linear Equations
1	0/2	0/2	0/2
2	1/2	1/2	1/2
3	1/1	0/1	20/22
4	19/21	20/21	20/22
5	20/22	20/22	20/22
6	0/2	0/2	0/2

[1] It is exact under the assumption that the length of the last register access interval of a basic block is the same in every execution. Then, the last interval v^c is vulnerable fs times, the number of times the basic block is followed first by a read; therefore, its vulnerability contribution is fsv^c.

predecessors BB$_3$ and BB$_4$. Again using the definition of conditional probability, $N(3,5)P(4|3,5) + N(4,5)P(4|4,5) = N_5P(4|5)$, where N_i is the execution frequency of BB$_i$. Since $N(3,5) = N_3$ and $N(4,5) = N_4$ (having only one immediate successor), we have $N_3s_3 + N_4s_4 = N_5s_5$. In our example, $N_3 = 1$. Therefore N_3s_3 can be either 0 or 1. Since we know that $s_5 = P(4|5) = 10/11$, we have $s_4 = 20/21$ or $s_4 = 19/21$, which are the two cases illustrated.

Linear Equation Method: Once we approximate length-n conditional probabilities with branch probabilities, we can use the linear equation method to compute post-conditions. Let s_i^* be a variable associated with BB$_i$, which is defined to be 1, 0, or s_i if the first-access-type of BB$_i$ is read, write, or no access, respectively. Then between every node BB$_i$ and its immediate successors BB$_j$'s this relationship holds: $s_i = \sum_j P(j|i)s_j^*$. Finally as an initial condition we require $s_{sink}^* = 0$ for the sink node of an inter-procedural CFG. This procedure has to be repeated for each register in the RF, since post-conditions can be different for different registers. This set of linear equations has a unique solution and can be solved very efficiently. The complexity of generating the equations for one register is $O(E)$, where E is the number of edges in the inter-procedural control flow graph, and in practice dominated by the linear equation solver runtime. Solving the equations does not take too long as shown in our experiments, since most of them are trivial involving only two or less variables and there is no inequality. For the example of Figure 2, the linear equation method yields the solution listed in the last column of Table I, which is in between the two solutions illustrated.

VI. Guiding Compiler Optimizations

Since vulnerability is dependent on when a register is read and written, and compiler optimizations, e.g., loop transformations, instruction scheduling, register allocation, directly affect that, compilers can greatly affect vulnerability of programs. Static technique to estimate RFV opens door for a whole range of compiler RFV optimizations. While compilers can reduce program vulnerability without hardware support, synergistic hardware-software techniques can be even more effective.

We use a popular microarchitectural technique, namely, partially protected RF (PPRF) as a vehicle to demonstrate the effectiveness of compiler techniques. The main idea behind PPRF is that full protection has very high overheads in terms of speed, area, and power, and that only a fraction of register variables contribute to majority of RFV. Therefore cost-effective RFV reduction can be achieved by protecting only a part of RF. There are several flavors of this technique [2], [10], [11], but all are complete hardware solutions. They make the decision of which register variable to protect at runtime, necessitating significant power overheads. Compiler approaches can enhance this by making these decisions at compile-time.

To allow compile-time decision of protected registers, we assume that partially protected RF defines a set of architectural registers that are protected with certain hardware mechanisms such as duplication, parity, or ECC. All existing approaches have attempted to minimize RFV by predicting which register variables should be mapped to the protected RF at runtime. But for the compiler, the problem of RFV minimization is exactly the same as the traditional problem of register allocation, with preferential allocation to the protected registers—the RFV will be minimized as long as we keep all the protected registers busy all the time. However the problem of power-efficient RF vulnerability reduction is more challenging for the compiler, as it requires finding out variables that will have long lifetimes, but will be accessed rarely (assuming that RF power is proportional to the number of accesses). Furthermore, compilers are better suited to solve this problem, since it requires "whole program information." Therefore the goal of our compiler optimization is to find the register swapping that will minimize RFV (V) as well as the RF energy (E), or the weighted sum of the two as in $V + \beta E$. This is our objective function.

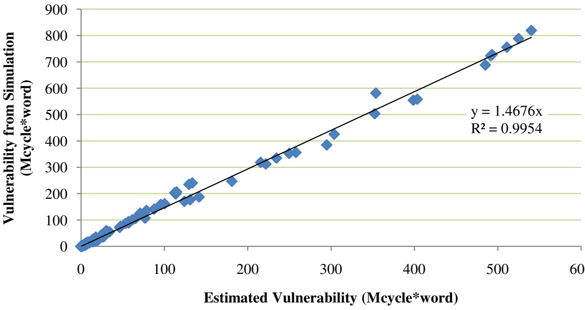


Fig. 3. Simulation vs. Static Estimation.

Register swapping or reassignment can be performed on a per-function or per-program basis without sacrificing application performance [13]. On a per-program basis, Application-level Register Swapping (ARS) swaps registers globally throughout the entire program, including library routines, and therefore can work on all registers, except for those architecturally distinguished or reserved for system calls. An example of an architecturally distinguished register is the link register `r31` in the MIPS architecture [21], since it is implicitly accessed by instruction `jal`. On the other hand, Function-level Register Swapping (FRS) may swap registers differently in each function, and can work on and within certain classes of registers such as callee-saved registers (also called s-registers in MIPS) and caller-saved registers (called t-registers in MIPS). ARS and FRS may be combined for greater flexibility.

Determining the optimal register reassignment (RR) for ARS is easy. Since protecting a register decreases V but increases E, with the energy increase proportional to the number of accesses, energy efficiency can be maximized by protecting the registers with highest vulnerability and fewest accesses. It is shown [13] that sorting the registers according to their $V_r - \beta E_r$ values gives the optimal RR that minimizes the objective function. In FRS, however, changing the register assignment in one function may affect the vulnerability in other functions, particularly for the callee saved registers, or the s-registers in MIPS convention. For example, an s-register that is written right after a function call and return will have different vulnerability depending on whether the register is accessed in the callee function. Therefore, we can either do the global optimization, considering all the functions simultaneously, which is computationally prohibitive, or determine each function's RR iteratively, recomputing vulnerability after deciding the RR of each function. In the latter case, the solution may not be the global optimum and its quality depends on the order of functions visited. We use the order of growing importance, as measured by the number of execution cycles of each function. The complexity of this algorithm is $O(FRL)$, where F is the number of functions, R is the number of registers, and L is the linear equation solver runtime. It can be reduced to $O(cRL)$ by recomputing RF vulnerability only before visiting the c most important functions, where c is a design parameter. Note that recomputing RF vulnerability as we determine each function's RR is not an option in profile based approaches such as [13].

VII. Experiments

We evaluate the effectiveness of our compiler approach using several applications[2] from an embedded benchmark suite [22]. For the target architecture we use the SimpleScalar-PISA [23], which is based on the MIPS instruction set. To emulate an embedded processor the cycle-accurate SimpleScalar simulator is configured for in-order execution. For the other parameters we use the simulator's default settings. Applications are compiled using GCC 2.7.2.3[3] with the benchmark-specified optimization level. From the executable we construct interprocedural CFG, from which we generate linear equations for post-conditions. Branch probabilities are obtained from an initial profiling, though they can be also statically estimated [24]. The linear equations are solved using the `lp_solve` software package [25] which can handle not only linear equations but also mixed-integer linear optimizations. The solutions are post-conditions, from which to compute vulnerability we need execution frequency and basic block vulnerability components (v^i, v^c). Execution frequency is computed from branch probabilities using a method similar to [19], and basic block vulnerability components are approximated with instruction counts.

A. Validation of Static RFV Estimation

First we compare the accuracy of our static RFV estimation. Figure 3 compares the vulnerability generated from simulation (on the y-axis) and from our static estimation (on the x-axis). Each dot represents one register. Due to the approximations, viz. basic block vulnerability components to instruction counts and path probabilities to branch probabilities, some degree of inaccuracy is expected. However, despite those approximations we see the dots placed near the $y = ax$ line, indicating that our static RFV estimation can closely follow the measured vulnerability most of the time. The accuracy can be further improved if the approximations are removed.

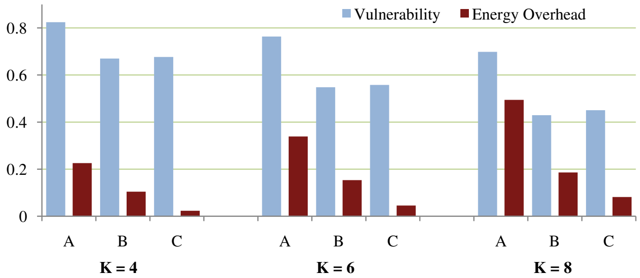


Fig. 4. Normalized V and E (on y-axis) of the three cases (A, B, C).

B. Effectiveness of Compiler Approach

For a fair comparison of and demonstration of the need and usefulness of compiler RFV optimizations, we consider three flavors of partially protected register file (PPRF) approach.

- **[Case A: PPRF using Runtime Prediction]** K registers are protected, and the runtime prediction logic in [11] is used to map register variables to the protected registers in hardware at runtime.
- **[Case B: Hardwired PPRF]** We profile all the applications, and compute each register vulnerability. The top K registers with the highest vulnerability are then hardwired for protection.
- **[Case C: Hardwired PPRF with Compiler Optimization]** K registers are hardwired for protection. We apply our compiler optimization using static RFV estimation, and statically rename the top K registers with the highest cost metric to be mapped onto the protected registers. For compiler optimization we set c to 5 and β to the ratio of RFV to the total access count.

Figure 4 compares the RFV and RF energy overhead of the seven applications (aggregated by taking geometric mean) for different values of K. The RFV is normalized to that of the original unprotected RF. The energy overhead is normalized to the energy consumption of the original RF, assuming that the energy overhead (i.e., ECC generation and checking) of a protected register access is equal to the energy consumption of an unprotected register access, which enables technology independent comparison. We do not include prediction power in Case A.

Vulnerability Aspect: We observe that in Case A, the runtime prediction can achieve 20% to 30% reduction in RFV, which is consistent with earlier results [11]. Compared to that, Cases B and C can achieve almost twice the vulnerability reduction with the same number of protected registers. This means that the runtime prediction algorithm in A is not as effective or optimal as static (C) or offline (B) decisions. This is because vulnerability estimation requires post-condition, which is not available to dynamic schemes. On the other hand, our compiler technique can achieve almost the same vulnerability reduction as Case B, which reinforces the accuracy of our static RFV estimation.

Energy Aspect: The energy overhead difference is more dramatic. The graph shows that Case A has 20% to 50% accesses to the protected registers, as normalized to the total number of RF accesses. This implies that even without prediction power, which may also be considerable, the runtime scheme can consume 20% to 50% more energy for RF protection, if the ECC generation/checking takes about the same power as one RF access. The energy overhead goes up rapidly as the number of protected registers increases. Hardwired PPRF approach has much lower energy overhead.

[2] They are JPEG, patricia, ispell, rsynth, string search, PGP, and FFT.
[3] This is one of the latest versions supporting the SimpleScalar target.

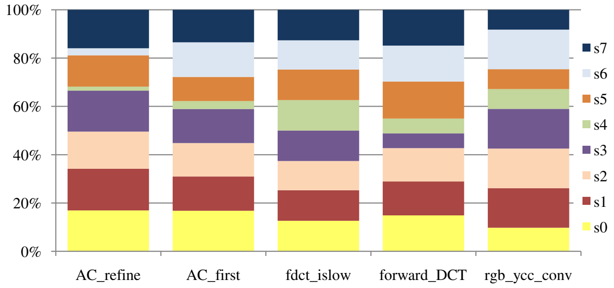


Fig. 5. Distribution of optimization cost metric across s-registers (on y-axis).

The primary reason is that the runtime prediction may evict existing variables from protected registers to accommodate new variables that appear to be longer-lived (this involves only updating ECC table), in a bid to maximize the vulnerability reduction. The compiler approach, Case C offers the most energy-efficient RF vulnerability reduction. Compared to Case B, compiler optimizations can bring down the energy overhead to 1/4, for K=4, and 1/2 for K=8. Although our compiler optimization involves solving multiple sets of linear equations, the optimization time is modest; the entire vulnerability estimation and optimization for each application took at most one minute on a 2GHz Xeon PC, except for one application, PGP, which took about 7 minutes.

While it is clear from a comparison between case A and B that there is no need for runtime prediction, and even a hardwired scheme of register protection provides energy efficient protection, to demonstrate the need/scope of compiler optimization, we plot the optimization cost metric for the top five functions of JPEG application for each register (Figure 5). The cost metric represents the energy efficiency of protecting a register; thus registers with higher metric are better to protect in term of energy efficiency. Only s-registers are shown since they contribute the most to vulnerability and need careful selection. (This is because they are live across function calls, in contrast to the t-registers, which are live only within a function.) From the graph we see that the contribution of each register varies greatly across different functions, as is the case with different applications (not shown); consequently, no fixed ordering will be optimal, which indicates significant need and scope for compiler techniques to analyze and find out the registers that need to be protected.

VIII. Summary

This paper proposes a static analysis approach to estimate RFV. Static estimation of vulnerability is more challenging than performance estimation due to the path-dependent nature of vulnerability computation. This paper makes several fundamental contributions. First, we analyzed this dependence at the basic block level, which allows us to decompose the basic block vulnerability into intrinsic and conditional vulnerabilities. Combining the two with post-condition gives the exact and efficient way to represent the RFV. Second, since the exact computation of post-condition requires path frequency information of an interprocedural CFG, which is not practical, we presented a fast, reasonably accurate, linear equation-based method to estimate post-condition from branch probabilities. Third, we demonstrated practical application of our static estimation technique through compiler optimizations for partially protected RF. Our experimental results on a number of embedded applications indicate that not only our static RFV estimation is fast and relatively accurate but also compiler optimizations enabled by our static estimation can achieve very cost-effective RF vulnerability reduction. When used to compile for partially protected RF, simplest of compiler-management schemes can be much more energy efficient than hardware schemes, and explicit optimization can further reduce the energy overhead by up to 50% to 75% while not sacrificing vulnerability.

This work is seminal in that static RFV analysis opens doors to develop compiler approaches for optimizing RF reliability. Future work includes studying register allocation and instruction scheduling for power and performance efficient RF protection.

References

[1] “International technology roadmap for semiconductors 2007 executive summary.” [Online]. Available: http://www.itrs.net/
[2] J. A. Blome *et al.*, “Cost-efficient soft error protection for embedded microprocessors,” in *CASES ’06*, 2006, pp. 421–431.
[3] S. Mitra *et al.*, “Robust system design with built-in soft-error resilience,” *IEEE Computer*, vol. 38, pp. 43–52, 2005.
[4] A. Shrivastava *et al.*, “Operation tables for scheduling in the presence of incomplete bypassing.” in *CODES+ISSS*, 2004, pp. 194–199.
[5] K. Skadron *et al.*, “Temperature-aware microarchitecture,” in *Proc. Int’l Symp. on Computer Architecture*, 2003, pp. 2–13.
[6] P. Dodd and L. Massengill, “Basic mechanisms and modeling of single-event upset in digital microelectronics,” *IEEE Trans. on Nuclear Science*, vol. 50, pp. 583–602, June 2003.
[7] P. Dodd *et al.*, “Neutron-induced latchup in SRAMs at ground level,” in *41st Annual Int’l Reliability Physics Symposium*, 2003, pp. 51–55.
[8] J. Hasan *et al.*, “Heat stroke: Power-density-based denial of service in SMT,” in *HPCA*, 2005.
[9] S. Park *et al.*, “Bypass aware instruction scheduling for register file power reduction,” in *LCTES*, 2006, pp. 173–181.
[10] G. Memik *et al.*, “Engineering over-clocking: reliability-performance trade-offs for high-performance register files,” *DSN ’05*, 2005.
[11] P. Montesinos *et al.*, “Using register lifetime predictions to protect register files against soft errors,” in *DSN ’07*, 2007, pp. 286–296.
[12] M. Kandala *et al.*, “An area-efficient approach to improving register file reliability against transient errors,” in *ISEC*, 2007.
[13] *Compiler-Managed Register File Protection for Energy-Efficient Soft Error Reduction*, 2009.
[14] J. Yan and W. Zhang, “Compiler-guided register reliability improvement against soft errors,” in *EMSOFT ’05*, 2005, pp. 203–209.
[15] S. S. Mukherjee *et al.*, “A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor,” in *Proc. International Symposium on Microarchitecture*, Dec 2003.
[16] T. J. Slegel *et al.*, “IBM’s S/390 G5 microprocessor design,” *IEEE Micro*, vol. 19, pp. 12–23, 1999.
[17] X. Li *et al.*, “Online estimation of architectural vulnerability factor for soft errors,” *SIGARCH Comput. Archit. News*, vol. 36, pp. 341–352, 2008.
[18] K. R. Walcott *et al.*, “Dynamic prediction of architectural vulnerability from microarchitectural state,” *SIGARCH CA News*, pp. 516–527, 2007.
[19] K. Chen *et al.*, “Retargetable static timing analysis for embedded software,” in *ISSS ’01*, 2001, pp. 39–44.
[20] S. Chatterjee *et al.*, “Exact analysis of the cache behavior of nested loops,” in *PLDI ’01*, 2001, pp. 286–297.
[21] D. Patterson and J. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers, 2004.
[22] M. Guthaus *et al.*, “Mibench: A free, commercially representative embedded benchmark suite,” in *IWWC*, 2001.
[23] T. Austin, “SimpleScalar LLC.”
[24] Y. Wu and J. Larus, “Static branch frequency and program profile analysis,” in *MICRO 27*, 1994, pp. 1–11.
[25] “Open-source mixed-integer linear programming system.” [Online]. Available: http://lpsolve.sourceforge.net/5.5/