Static Analysis of Register File Vulnerability

Jongeun Lee *Member, IEEE,* and Aviral Shrivastava

***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 RF vulnerability estimation mechanism. We validate and compare a linear equation based method and an iterative method. Also we demonstrate a practical application of RF vulnerability estimation 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.**

***Index Terms*—Architectural vulnerability factor, compilers, embedded systems, partially protected register file, soft error, static analysis.**

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 applications as well as 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, for an embedded processor, majority of the faults both in combinational and sequential logic that affect the architectural state come from the register file. Since register files are accessed very frequently, corrupted data in the register file can quickly spread to other parts of the system, increasing chances of an error. While memory structures, like caches and main memory are routinely protected using parity or error correcting codes (ECC) [3], protecting the register file (RF) using such hardware schemes is prohibitive, especially for systems that are constrained by power such as embedded systems. Protecting RF is challenging because RF: 1) is often in the timing-critical path [4] of the processor, and 2) is one of the hottest blocks on the chip [5]. Keeping the RF cool is important, not only to save power for embedded systems and avoid performance loss due to thermal degradation [6], [7], but also because higher temperature rapidly reduces reliability of circuits [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 by the hardware at runtime, they necessarily incur high power overheads.

Potentially more interesting is the compiler approach, for which we propose a static analysis in this paper. Compilers can mitigate the effect of soft errors in RF with or without hardware support. For instance, shortening the average live range of variables through memory spills by a compiler may reduce the soft error rate of the RF if the L1 data cache is already protected [13]. Also for a partially protected RF where only some entries are protected, a compiler may be able to make a better decision of which variables to place in the protected registers for maximum energy efficiency by utilizing the global program information than without such information. *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) [14]. 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 [15], which may not be very useful for compiler optimizations.

This paper proposes a static analysis to accurately estimate RFV, which is very useful for compiler-based soft error reduction. Static estimation of vulnerability is more challenging than performance estimation. This is because while performance such as dynamic instruction count is dependent only on

Manuscript received June 7, 2010; revised September 19, 2010; accepted October 26, 2010. Date of current version March 18, 2011. This work was supported in part by the funding from National Science Foundation (NSF), under Grants CCF-0916652, CCF-1055094, and IIP-0856090, in part by NSF Industry/University Cooperative Research Center for Embedded Systems, Arizona State University, Tempe, Raytheon Company, Aurora, CO, Science Foundation Arizona, Phoenix, and Stardust Foundation, Auburn, NY, and in part 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. This paper was recommended by Associate Editor L. Pozzi.

J. Lee is with the School of Electrical and Computer Engineering, Ulsan National Institute of Science and Technology, Ulsan 689-798, Korea (e-mail: jlee@unist.ac.kr).

A. Shrivastava is with the Department of Computer Science and Engineering, Arizona State University, Tempe, AZ 85281 USA (e-mail: aviral.shrivastava@asu.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2010.2095630

attributes of program points (e.g., execution count of each basic block), and therefore can be efficiently computed from branch probabilities, the vulnerability of a register is dependent on the execution path of a program, which requires much more information than branch probabilities. To efficiently compute RFV, we first distribute the RFV over the basic blocks, leading to the concept of *basic block vulnerability* or simply *block vulnerability*, which is the portion of RFV attributed to each basic block. While the basic block vulnerability allows the RFV of the program to be easily computed by simply summing up all the basic block vulnerabilities weighted by the execution frequency of each basic block, the burden is now on finding out the exact basic block vulnerability. The novelty of our method is that we use a linear function representation ($ax + b$) for the basic block vulnerability, which is otherwise very difficult to capture accurately. For a given register, the coefficients (a, b) of the linear function can be directly determined from each basic block and the independent variable (x) is what we call the *liveness* of the register. The liveness of a register for a given program point is the probability of the register being first read along the paths from the program point to the exit, and can be estimated from branch probabilities with reasonable accuracy though exact computation of it requires a path-sensitive analysis.

Thus, our approach breaks the problem of estimating RFV into: 1) computing the vulnerability of a register as a function of *register liveness*, and 2) estimating the register liveness from branch probabilities. The first step can be done very efficiently and accurately. For the second step we present two methods: a linear equation method and an iterative method based on data-flow analysis. Both are efficient and highly accurate. Thus, our static vulnerability estimation can be fast while being accurate.

To illustrate the use of our static analysis for compiler-based soft error reduction, we consider optimizing the register assignment of a program for a partially protected RF (PPRF). A PPRF [16] has a set of registers that are protected, but the protected registers require more energy to access than unprotected ones; therefore, to provide the maximal level of protection with minimal energy overhead, one needs to consider access count as well as vulnerability of each variable. This becomes easier if it is done as a post-link optimization, which estimates vulnerability of each register (from the compiled binary program) and swaps some registers if profitable, both on a procedure and a global level.

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, the 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 66% to 75% while not increasing vulnerability significantly.

II. Related Work

Existing RF protection mechanisms can be broadly classified into hardware and software techniques. While full-hardware techniques, such as protecting every register and latch with either ECC or parity as in the case of the IBM G5 enterprise server [17], can provide a very high level of protection that is also transparent to software systems, protecting the whole register file in this way has very significant power and performance overheads, which may be prohibitive for embedded systems. Consequently, Montesinos *et al.* [11] and Blome *et al.* [2] proposed partial protection schemes that protect only a subset of the RF, where the decision of which variables to protect is made in hardware. At the other end of the spectrum are full-software schemes such as code duplication [18], [19] and control flow checking [20], which can cover not only the RF but the entire processor, albeit with a high overhead in code size and performance. Lee *et al.* [21] proposed a low-overhead full-software scheme that specifically targets RF vulnerability reduction. There are also hybrid approaches [13], [16] that rely on hardware protection mechanisms such as ECC, but the management of the hardware is governed by software, often at compile or link-time. These so-called compiler approaches as well as software approaches have one distinct advantage over hardware approaches, that the level of protection can be easily changed to tradeoff between reliability and power in a varying environment. This kind of flexibility simply lacks in hardware approaches.

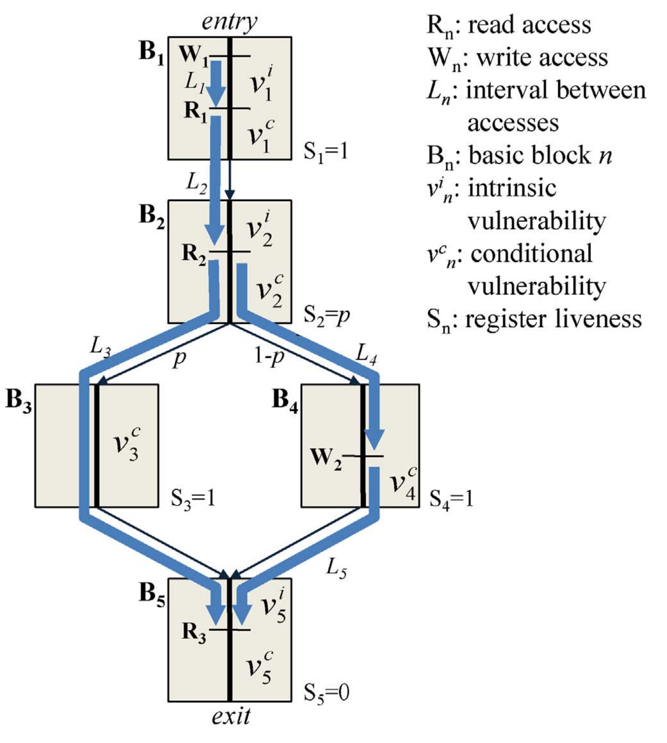
We are aware of only two compiler techniques [13], [16], both of which are based on partially protected register files and try to reduce RF vulnerability by different register allocation. While effective, those techniques are profile based, and are therefore extremely limited in application. In addition 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 AVF by Mukherjee *et al.* [14], which is a quantitative measure of the amount of "live" information that needs protection of each microarchitectural component. Techniques for runtime estimation of AVF [22], [23] were proposed, mainly to be used by operating systems to adjust the protection level. To guide compiler optimizations for soft error reduction, architectural simulation as opposed to microarchitectural simulation can be used [15]. An analytic method to estimate data cache vulnerability has been proposed [24]. However, there is no static method to estimate vulnerability of register files, except our own [25], which is extended in this paper to include an iterative method based on data-flow analysis.

III. RF Vulnerability Computation

Vulnerability of a register[1] is defined as the total time for which it is vulnerable, or holds useful data. The RFV is then just the sum of the vulnerability of all the registers in the RF (we analyze each register separately for its vulnerability).

[1]Registers mentioned in this paper mean the actual hardware registers, which may be regarded as variables from the compiler's point of view on an architecture without hardware register renaming.



Interval	Length	Frequency	Vulnerability
entry–W_1	L_0	1	0
W_1–R_1	L_1	1	L_1
R_1–R_2	L_2	1	L_2
R_2–R_3	L_3	p	pL_3
R_2–W_2	L_4	$1-p$	0
W_2–R_3	L_5	$1-p$	$(1-p)L_5$
R_3–exit	L_6	1	0

$$\text{RFV} = L_1 + L_2 + pL_3 + (1-p)L_5$$

Fig. 1. Computing the vulnerability of a register given the execution frequency of every path. Register vulnerability is the combined length of vulnerable intervals, multiplied by their execution frequencies, as shown in the embedded table.

Fig. 1 illustrates a control flow graph (CFG), where nodes are basic blocks and edges represent control dependence. The accesses to a register are marked by R_i or W_i depending on whether they are read or write, 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 B$_3$ is executed after B$_2$. 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 spans several basic blocks. The presence of loops in the program makes this problem intractable.

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 B$_2$ for example. The interval from the beginning of B$_2$ to R_2 is definitely vulnerable, since this interval ends in a read. However, whether the interval from R_2 to the end of B$_2$ is vulnerable may be determined immediately in the next basic block, as is the case with the path B$_2$–B$_4$, or may be determined several basic blocks down the flow, as is the case with the path B$_2$–B$_3$–B$_5$. This inherent path dependence of vulnerability computation prevents us from assigning one number to basic block vulnerability.

IV. New Vulnerability Representation

To isolate the path 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 *register liveness*, is the probability summarizing the path dependence, defined as the probability of the next access to the register being a read. This is essentially decomposing the vulnerability into *intrinsic vulnerability*, v^i, which definitely contributes to vulnerability, and *conditional vulnerability*, v^c, which does so only conditionally depending on the type of the first access in the following blocks. v^i is computed as the 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 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 [26] may be used.

Unlike v^i and v^c, which are determined from their own basic blocks, register liveness s is determined from other basic blocks. Once s is known either through profiling or static analysis, the total RFV (for one register) 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 basic block j, respectively, and v^i_j, v^c_j, and s_j are also of basic block j. 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 $V_5 = v^i_5 + v^c_5 \cdot 0$. The total vulnerability is given as below, which is equal to the earlier formula after replacing L_is

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

For execution traces involving each basic block no more than once, it is easy to see that the linear function representation of basic block vulnerability is exact for any value of s. For more general cases, where basic block j appears f_j times in an execution trace, we define v^i, v^c, s as expected values, such that $v^i_j = E(\nu^i_j) = \sum \nu^i_j(n)/f_j$, where $E(\cdot)$ is the expectation operator, ν^i_j is the random variable for the

TABLE I
DIFFERENT REGISTER LIVENESS

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

intrinsic vulnerability in basic block j, and $\nu_j^i(n)$ is the intrinsic vulnerability in the nth instance of basic block j. The other variables v_j^c and s_j are also similarly defined as expectations of random variables ν_j^c and σ_j: $v_j^c = E(\nu_j^c)$ and $s_j = E(\sigma_j)$. Now, by definition, the vulnerability contribution of basic block j is $V_j = \sum_n[\nu_j^i(n) + \nu_j^c(n)\sigma_j(n)]$, or $V_j = f_jE(\nu_j^i + \nu_j^c\sigma_j)$. If we assume that ν_j^c and σ_j are independent, we can write $V_j = f_j[E(\nu_j^i) + E(\nu_j^c)E(\sigma_j)] = f_j(v_j^i + v_j^c s_j)$. This vulnerability representation is exact under the independence assumption. If the intrinsic and conditional vulnerabilities are defined in terms of number of instructions, ν_j^c degenerates into a constant, and the independence assumption becomes unnecessary.

The accuracy of the vulnerability representation hinges on the accuracy of the constituent variables, especially s_j. In the next section, we discuss how to accurately estimate register liveness.

V. Estimating Register Liveness

A. Analysis

For a given program point, register liveness is the probability of the register being first read along the paths from the program point to the exit. Consider a CFG in Fig. 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, register liveness cannot be determined from branch probabilities alone. The figure lists two execution paths that result in the same branch probabilities. In both cases, execution frequencies of the nodes and edges are exactly the same. However, register liveness of some basic blocks are different, as shown in Table I.

Let $P(a|q)$ denote the conditional probability of visiting *node a* right after *path q*. We denote a path by a sequence of numbers representing basic block IDs that comprise the path. In our example, using the definition of conditional probability, 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 B_5 specifies $P(4|5) = 10/11$ only, but it does not specify $P(4|3,5)$ nor $P(4|4,5)$, which is why we can have different register liveness from the same branch probabilities. If we were to obtain these probabilities from profiling, we would have to find the frequencies of length-2 paths such as (3, 5, 4) and (4, 5, 4), since $P(4|3,5) = P(3,5,4)/P(3,5) = N(3,5,4)/N(3,5)$,

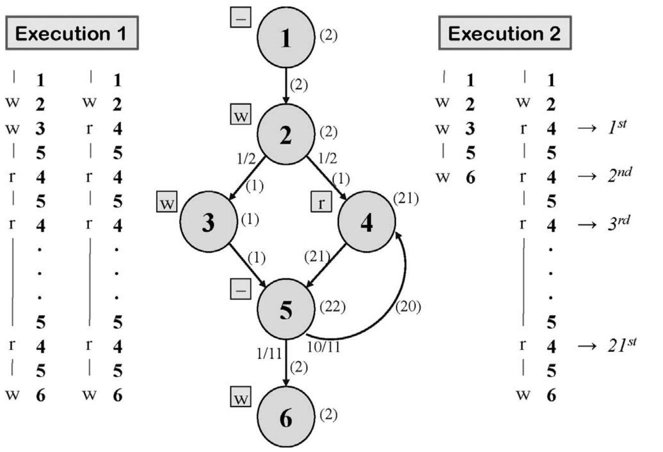

Execution 1
Execution 2
l 1 w 2 w 3 l 5 r 4 l 5 r 4 . . . 5 r 4 l 5 w 6
l 1 w 2 r 4 l 5 r 4 l 5 r 4 . . . 5 r 4 l 5 w 6
l 1 w 2 w 3 l 5 w 6
l 1 w 2 r 4 → 1st l 5 r 4 → 2nd l 5 r 4 → 3rd . . . 5 r 4 → 21st l 5 w 6
–
w
w
r
–
w
1/2
1/2
1/11
10/11
(2)
(1)
(21)
(22)
(20)

Fig. 2. Two possible executions with the same execution probabilities may result in different vulnerabilities, as they can have different register liveness as listed in Table I.

where $P(q)$ and $N(q)$ are the probability of 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 grow exponentially in the number of join nodes, even without a loop. Clearly this is very expensive and 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 methods for efficiently estimating register liveness.

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 B_5 has only two immediate predecessors B_3 and B_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 B_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.

B. Linear Equation Method

Once we approximate length-n conditional probabilities with branch probabilities, we can use linear equations to compute register liveness. In addition to s_i, let s_i^* be another variable associated with B_i, which is defined to be 1, 0, and s_i if the first-access-type of B_i is read, write, and no-access, respectively. Then between every node B_i and its immediate successors B_j's, this relationship holds: $s_i = \sum_j P(j|i)s_j^*$, which is a summation over all the immediate successors. Finally as the initial condition we require $s_{exit}^* = 0$ for the exit block of the interprocedural CFG [27]. This procedure is repeated for each register, since register liveness is independent across registers. The set of linear equations has a unique solution and can be solved efficiently for many applications. The complexity of generating the equations for a register is $O(E)$, where E is the number of edges in the interprocedural CFG, and far less than the runtime for solving the linear equations. In the example of Fig. 2, the linear equation method yields the solution listed in the last column of Table I, which is in between the two solutions illustrated.

C. Iterative Method Based on Data-Flow Analysis

Data-flow analysis [28] is often used by compilers to calculate certain attributes of a program (such as reaching definitions and live variables) at various program points. The analysis is typically formulated as solving data-flow equations concerning two attributes per basic block (i.e., *in* and *out*) on a given control flow graph. There are two versions of it depending on the direction of the information flow. In the backward version, for example, the data-flow equations, also called *transfer functions*, define how the *out* attribute of a basic block is transformed into the corresponding *in* attribute, as well as how the *in* attributes of the successor nodes are merged to create the *out* attribute of the current node, with the initial condition being given for the last block of the program, or the *exit* block. Thus the idea is to propagate these attributes in the direction opposite to the program execution, performing the transformation and merging specified by the transfer functions to find out the attributes of all basic blocks. However, the analysis is necessarily complicated by the existence of cycles in the control flow graph, and is most commonly solved using an iterative method.

Formally, a backward data-flow analysis is, given a control flow graph (N, E), where N is the set of basic blocks with *entry* and *exit* blocks in it, and E is the set of edges representing control dependence among the blocks, to compute $in(B), out(B) \in \mathbf{L}$ for every block $B \in N$ using the following data-flow equations, where *in*(B) and *out*(B) represent the data-flow information on entry to, and on exit from, B, respectively, and $\mathbf{L}$ is the lattice over which the two attributes are defined [29]

$$out(B) = \begin{cases} Init & \text{for } B = exit \\ \bigsqcup_{S \in Succ(B)} in(S) & \text{otherwise} \end{cases} \tag{1}$$

$$in(B) = F_B(out(B)) \tag{2}$$

where *Init* represents the initial value for the data-flow information on exit from the procedure (or from the program in the case of interprocedural CFG), $F_B(\,)$ represents the transformation of the data-flow information corresponding to block B, and the join operator $\sqcup$ models the effect of combining the data-flow information on its outgoing edges.

Before we discuss our register liveness problem, let us first consider a closely related problem. The *Live Registers* problem is to determine, for a given point in a program and a given architectural register, whether the register is first read along some path from the point to the exit. This is essentially the same as the live *variables* problem except that it is defined for registers instead of variables. The register-centric view can be useful if one wants to perform link-time optimization while respecting the general register allocation decisions. The live registers problem can be formulated as follows. The lattice $\mathbf{L}$ over which *in* and *out* attributes are defined is {0, 1} (1 if the register may be first read, 0 otherwise), and its join operator is the same as Boolean `or`. The transfer functions and the initial value are

$$Init = 0 \tag{3}$$

$$F_B(x) = \begin{cases} 0 & \text{if the register is first written in } B \\ 1 & \text{if the register is first read in } B \\ x & \text{if the register is not used in } B. \end{cases} \tag{4}$$

Since the transfer functions are monotone, the maximum fixed point solution to this problem can be computed [30] by an iterative algorithm.

Now our register liveness problem is to find, for a given point in a program and a given architectural register, the liveness of the register, or the probability of the register being first read after the point before the exit. Whereas the live registers problem asks a pure compiler-analysis question about a program (i.e., whether a particular register may be first read after a certain program point), the register liveness problem is concerned with the average behavior of the application (i.e., how often a particular register is first-read on average after a certain program point). Consequently, the register liveness analysis is done on real numbers rather than on lattice elements and uses statistical information such as branch probabilities.

Nonetheless we can cast the register liveness problem into the data-flow problem formalism, which allows us to use algorithms developed for data-flow analysis, such as worklist algorithm [29], [31]. The *Init* value and the transfer functions are the same as in the live registers problem, but the *in* and *out* attributes are defined over real numbers between 0 and 1 including the boundaries, and the join operator is replaced with the following summation operation:

$$\bigsqcup_{S \in Succ(B)} \text{in}(S) = \sum_{S \in Succ(B)} \text{in}(S) \cdot P(S|B) \tag{5}$$

where $P(S|B)$ is the branch probability from block B to block S.

An adapted version of the worklist algorithm is listed in Algorithm 1. Here, it is imperative to use an interprocedural CFG, since register liveness crucially depends on other functions in the program. Also, since our attributes are defined over real numbers, numerical precision may hinder the convergence of the algorithm; therefore, we allow some tolerance in the comparison in line 1 (the amount of tolerance is varied in our experiments). To further reduce the convergence time, we use a priority queue data structure for the worklist implementation, sorted in the postorder of the interprocedural CFG, which is shown to be effective for backward data-flow problems [31].

VI. Guiding Compiler Optimizations

Since vulnerability depends on when a register is read and written, and compiler optimizations (e.g., loop transformations, instruction scheduling, and register allocation)

Algorithm 1 Worklist algorithm

```
1:  for $\forall B \in N$: initialize $in(B)$ to 0
2:  worklist $\leftarrow N$      /* adding all blocks */
3:  while worklist is not empty do
4:      $B \leftarrow$ pop(worklist)
5:      update $out(B)$ using (1) and (5)
6:      new $\leftarrow$ evaluate transfer function (4) for $B$
7:      if new is different from $in(B)$ then
8:          $in(B) \leftarrow$ new
9:          add predecessors of $B$ to worklist
10:     end if
11: end while
```

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. To illustrate the use of our static analysis for compiler-based soft error reduction and to demonstrate its effectiveness, we consider optimizing the register assignment for a PPRF. 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 except [16]. Hardware techniques maintain the protection information (either the register value itself or its ECC value) in a small cache, which requires additionally storing tag information (i.e., register number) and the use of content addressable memory to best utilize the cache. Moreover, they require certain decisions to be made at runtime, such as which register variable to protect and which entry of the cache to evict, which necessarily incur significant power overheads. Compiler approaches can enhance this by: 1) making those decisions at compile-time, and 2) hard-wiring to protect only the K highest-numbered registers, eliminating the need for tags and content addressable memory.

The problem in compiler approaches is to determine the register allocation to best utilize the K protected registers. 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 very similar to the traditional problem of register allocation, with preferential allocation to the protected registers—the RFV will be minimized if all the protected registers are busy with dynamically live variables all the time, assuming that there is no side effect such as additional spills. However the problem of power-efficient RFV 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). Therefore, the goal of our compiler optimization is to find the register allocation that will minimize RFV (V) as well as the RF energy (E), with a weighting factor β for E (β is a design parameter). This becomes easier if it is done as a post-link optimization, which estimates vulnerability of each register (from the compiled binary program) and swaps

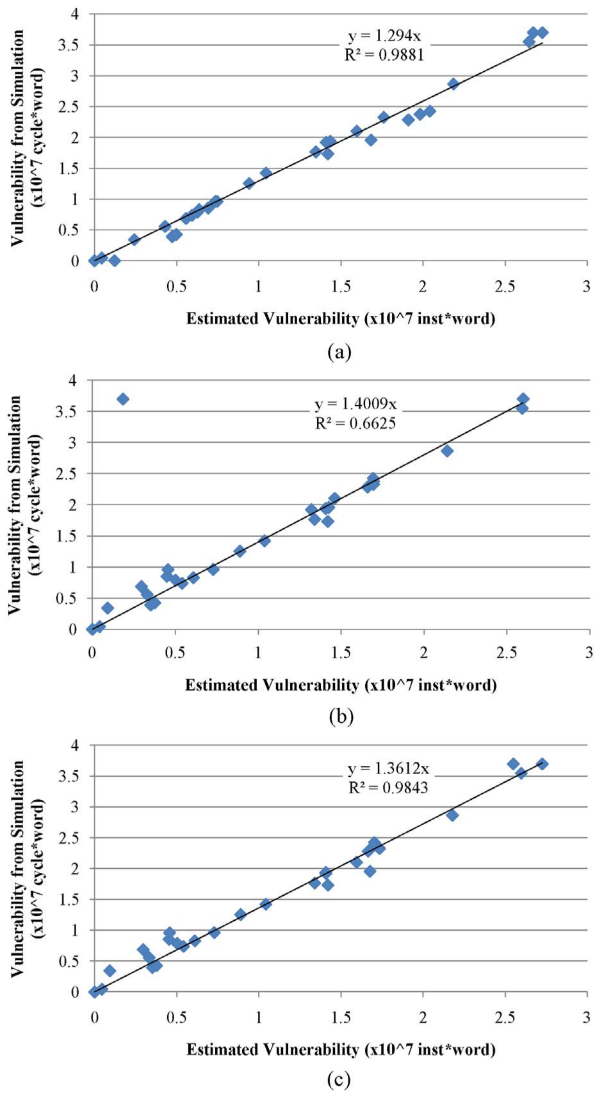


Fig. 3. Simulation versus static estimation for `jpeg`. (a) Linear equation method. (b) Data-flow-based method, tolerance = 10^{-4}. (c) Data-flow-based method, tolerance = 10^{-6}.

some registers if profitable, both on a procedure level [called function-level register swapping (FRS)] and on a global level [called program-level register swapping (PRS)].

Determining the optimal register reassignment (RR) for PRS is easy. Since protecting a register decreases V but increases E (assuming that the energy increase is proportional to the number of additional accesses), energy efficiency can be maximized by protecting the registers with highest vulnerability and fewest accesses. It is shown [16] that sorting the registers according to their $V_r - \beta E_r$ values gives the optimal RR for 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 the 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 [16]. Therefore, we can either do the global optimization, considering all the functions simultaneously, which is computationally prohibitive, or deter-

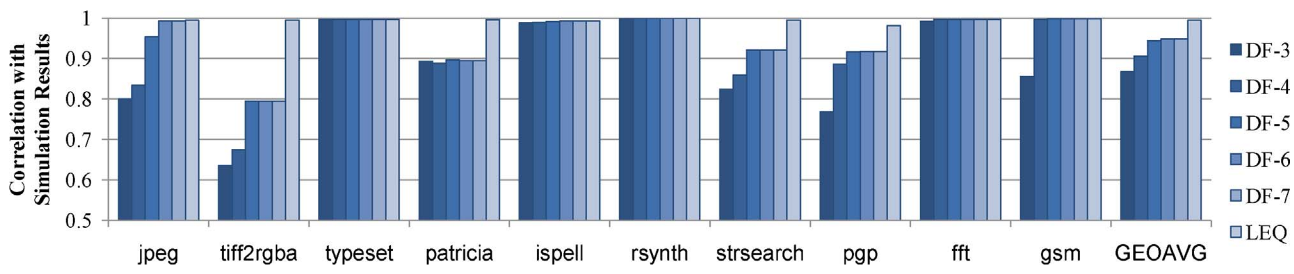
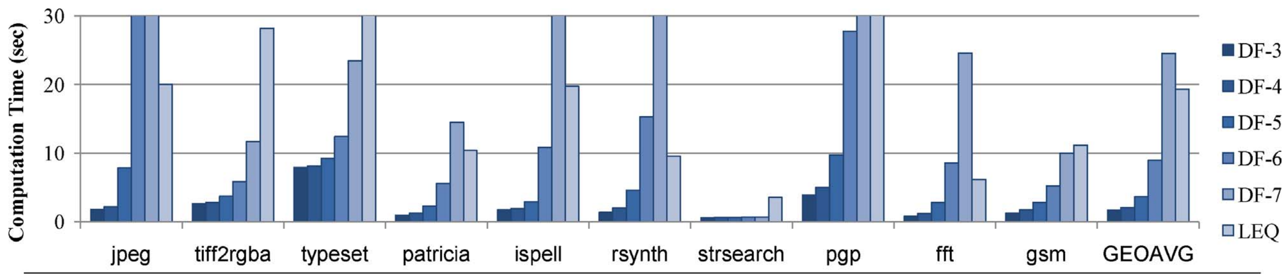


Fig. 4. Correlation between simulation and static estimation.



	jpeg	tiff2rgba	typeset	patricia	ispell	rsynth	strsearch	pgp	fft	gsm	GEOAVG
DF-6	61	6	12	6	11	15	1	28	9	5	9
DF-7	356	12	23	14	59	58	1	94	25	10	25
LEQ	20	28	246	10	20	10	4	110	6	11	19

Fig. 5. Computation time of static estimation (see the embedded table for large values exceeding 30 s).

mine 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 may depend on the order of visiting functions. 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 [16].

VII. Experiments

A. Experimental Setup

We evaluate the effectiveness of our compiler approach using embedded application benchmarks [32]. For the target architecture we use the SimpleScalar-PISA [33], which is based on the MIPS instruction set. To emulate an embedded processor the SimpleScalar simulator is configured for in-order execution. For the other parameters we use the simulator's default setting. Applications are compiled using GCC 2.7.2.3, one of the latest versions supporting the SimpleScalar target, with the benchmark-specified optimization levels. From an executable binary we construct an interprocedural CFG, from which we estimate register liveness using either linear equation method or data-flow analysis based method. For linear equation method, we use the `lp_solve` software [34] with problem scaling disabled. For data-flow analysis method, the worklist algorithm listed in Algorithm 1 is used, with tolerance value of 10^{-n}, where n is varied from 3 to 7 (difference within tolerance is considered to be equal). Branch probabilities are obtained from an initial profiling, though they can be also statically estimated [35]. To compute vulnerability from register liveness we need execution frequency and basic block vulnerability components (v^i, v^c). Execution frequency is computed from branch probabilities using linear equations, a method similar to [26], and basic block vulnerability components are approximated with instruction counts. All experiments are performed on a 2 GHz Xeon PC with 4 GB memory, and single threaded execution is implied whenever computation time is reported.

B. Validation of Static RFV Estimation

First we evaluate the accuracy of our static RFV estimation. Fig. 3 compares register vulnerability estimated statically (on x-axis) against that measured by simulation (on y-axis) for jpeg. Each dot represents one register. Due to the approximations, namely, basic block vulnerability components approximated to instruction counts and path probabilities to branch probabilities, some degree of inaccuracy is expected. Despite the approximations, however, we see most dots placed near $y = ax$ lines, which indicates that our static RFV estimation can closely follow the measured vulnerability most of the time. (The coefficient a merely tells about the cycle-per-instruction of the processor and not an indicator of the accuracy.) In (b), or data-flow based method with 10^{-4} tolerance, the dot far off the trend line in the upper left corner represents `gp`, the global pointer register, which seems to be particularly sensitive to the tolerance value. This error, while considerable in the 10^{-4} tolerance case, disappears in the 10^{-6} tolerance case (c).

We make similar comparisons for all applications. The result is summarized in Fig. 4, where DF-n represents the iterative

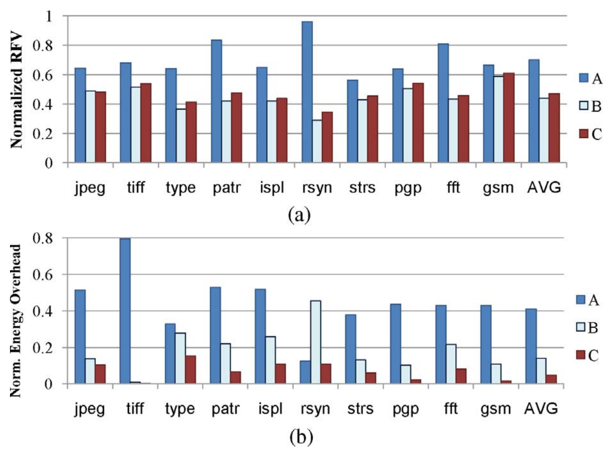


Fig. 6. Comparing three cases for eight protected registers ($K = 8$). Case A: PPRF using runtime prediction, Case B: hardwired PPRF, and Case C: hardwired PPRF with compiler optimization. (a) RF vulnerability. (b) Energy overhead.

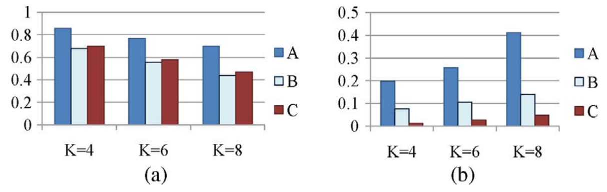


Fig. 7. Comparing the three Cases A, B, and C (see Fig. 6 for their definitions). Averaged over all applications. (a) RF vulnerability (normalized). (b) Energy overhead (normalized).

method with the tolerance value of 10^{-n}, and LEQ represents the linear equation method. The correlation coefficient is computed per benchmark, per estimation method. In six out of the ten applications we find that the estimation results by the iterative method can be as good as those of the linear equation method. However, in the rest of the applications there seems to be a gap, which is hard to close even when the tolerance is reduced to 10^{-9}. Fig. 5 compares the computation time of different methods. The computation time of the iterative method grows exponentially with decreasing tolerance. The computation time of the linear equation method is not high, being less than 30 s in 2 out of 10 applications. The linear equation method is fast because most of the equations generated by our method are simple involving only two or less variables and there is no inequality. Overall, the linear equation method gives the most accurate results in shorter time than DF-7.

C. 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 PPRF approach.

1) *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.
2) *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.
3) *Case C: hardwired PPRF with compiler optimization.* K registers are hardwired for protection. We apply our compiler optimization using static RFV estimation (using the linear equation method), 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 (i.e., RF vulnerability is recomputed for the top five functions) and β to the ratio of RFV to the total access count.

Fig. 6 compares the RF vulnerability and the RF energy overhead in the three cases. The number of protected registers is set to eight ($K = 8$), or one-fourth of the total number of registers [11]. The RF vulnerability 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, or register duplication and comparison) 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.

1) *Vulnerability Aspect:* In Case A, we observe that the runtime prediction can achieve an average of 30% reduction in RFV, which is consistent with earlier results (e.g., [11]). Compared to Case A, Cases B and C can consistently achieve greater RFV reduction, and on average nearly twice the RFV reduction with the same number of protected registers, though there is a wide variance across applications. 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 accurate vulnerability estimation requires knowledge of register liveness, which is not available to runtime 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. Fig. 7(a) shows the normalized RF vulnerability for different values of K. A similar trend is observed, while the difference between different schemes is larger with larger values of K.

2) *Energy Aspect:* The energy overhead difference is more dramatic. Fig. 6(b) shows that Case A makes about 40% accesses on average 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 40% more energy for RF protection, if the protection mechanism (i.e., duplication or ECC generation/checking takes about the same power as one RF access). The energy overhead of the runtime prediction can be exceptionally high in some applications; in tiff2rgba, the runtime prediction in A has 80% energy overhead whereas static (C) and offline (B) decisions achieve the same RFV reduction with negligible energy overhead. A primary reason of this is because the runtime prediction may evict existing variables from protected registers to accommodate new variables that appear to be longer-lived, in a bid to maximize the vulnerability reduction. Fig. 7(b) suggests that the energy overhead goes up rapidly as the number of protected registers increases. Compared to the runtime prediction, the hardwired

TABLE II
OPTIMIZATION TIME FOR $K = 8$ (IN S)

Appli.	jpeg	tiff	type	patr	ispl	rsyn	strs	pgp	fft	gsm
Time	59	46	919	39	65	30	10	436	21	40

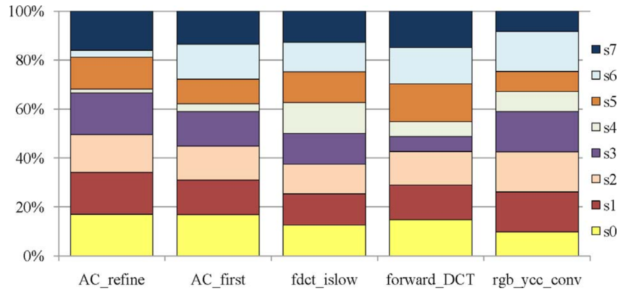


Fig. 8. Distribution of optimization cost metric across s-registers (on y-axis), showing that the ideal register swapping can be different across functions, which are listed on x-axis.

PPRF approach (Case B) has much lower energy overhead, but 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 $K = 6$ and 1/3 for $K = 8$.

Although our compiler optimization involves solving multiple sets of linear equations, the optimization time is modest as shown in Table II; the entire vulnerability estimation and optimization for each application took at most about one minute, except for two applications, pgp and typeset, which took about 7 and 15 min, respectively.

D. Scope of Compiler Approach

While our comparison on the MIPS architecture between Cases A and B seems to suggest that even a hardwired scheme of register protection can be superior to hardware-based runtime prediction in terms of energy efficiency, to demonstrate the need and scope of compiler optimization, we plot the optimization cost metric [16] for the top five functions of jpeg application for each register in Fig. 8. The cost metric represents the energy efficiency of protecting a register—the higher the metric is, the better it is to protect the register in terms 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; consequently, no fixed ordering will be optimal, which indicates significant need and scope for compiler techniques (at least for the MIPS architecture) to analyze and find out the registers that need to be protected.

VIII. CONCLUSION

This paper proposed 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 register liveness gave the exact and efficient way to represent the RFV. Second, since the exact computation of register liveness requires a path-sensitive analysis on an interprocedural CFG, which is very expensive, we presented fast, reasonably accurate methods that use branch probabilities only. 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 66% to 75% while not sacrificing vulnerability. Our work opens doors to develop compiler approaches for optimizing RF reliability. Future work includes studying more register allocation and instruction scheduling schemes for power and performance efficient RF protection.

ACKNOWLEDGMENT

The authors would like to thank all the members of the Compiler Microarchitecture Laboratory at Arizona State University, Tempe, for their valuable support in this paper.

REFERENCES

[1] *International Technology Roadmap for Semiconductors 2007 Executive Summary* [Online]. Available: http://www.itrs.net

[2] J. A. Blome, S. Gupta, S. Feng, and S. Mahlke, "Cost-efficient soft error protection for embedded microprocessors," in *Proc. CASES*, 2006, pp. 421–431.

[3] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K. S. Kim, "Robust system design with built-in soft-error resilience," *IEEE Comput.*, vol. 38, no. 2, pp. 43–52, Feb. 2005.

[4] A. Shrivastava, E. Earlie, N. D. Dutt, and A. Nicolau, "Operation tables for scheduling in the presence of incomplete bypassing." in *Proc. CODES+ISSS*, 2004, pp. 194–199.

[5] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan, "Temperature-aware microarchitecture," in *Proc. Int. Symp. Comput. Architecture*, 2003, pp. 2–13.

[6] J. Hasan, A. Jalote, T. Vijaykumar, and C. Brodley, "Heat stroke: Power-density-based denial of service in SMT," in *Proc. HPCA*, Feb. 2005, pp. 166–177.

[7] S. Park, A. Shrivastava, N. Dutt, A. Nicolau, Y. Paek, and E. Earlie, "Bypass aware instruction scheduling for register file power reduction," in *Proc. LCTES*, 2006, pp. 173–181.

[8] P. Dodd and L. Massengill, "Basic mechanisms and modeling of single-event upset in digital microelectronics," *IEEE Trans. Nuclear Sci.*, vol. 50, no. 3, pp. 583–602, Jun. 2003.

[9] P. Dodd, M. Shaneyfelt, J. Schwank, and G. Hash, "Neutron-induced latchup in SRAMs at ground level," in *Proc. 41st Annu. Int. Reliab. Phys. Symp.*, 2003, pp. 51–55.

[10] G. Memik, M. Chowdhury, A. Mallik, and Y. Ismail, "Engineering over-clocking: Reliability-performance tradeoffs for high-performance register files," in *Proc. DSN*, Jul. 2005, pp. 770–779.

[11] P. Montesinos, W. Liu, and J. Torrellas, "Using register lifetime predictions to protect register files against soft errors," in *Proc. DSN*, 2007, pp. 286–296.

[12] M. Kandala, W. Zhang, and L. Yang, "An area-efficient approach to improving register file reliability against transient errors," in *Proc. ISEC*, 2007, pp. 798–803.
[13] J. Yan and W. Zhang, "Compiler-guided register reliability improvement against soft errors," in *Proc. EMSOFT*, 2005, pp. 203–209.
[14] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proc. Int. Symp. Microarchitecture*, Dec. 2003, p. 29.
[15] V. Sridharan and D. Kaeli, "Eliminating microarchitectural dependency from architectural vulnerability," in *Proc. Int. Symp. HPCA*, 2009, pp. 117–128.
[16] J. Lee and A. Shrivastava, "A compiler-microarchitecture hybrid approach to soft error reduction for register files," *IEEE Trans. Comput.-Aided Des. Integ. Circuits Syst.*, vol. 29, no. 7, pp. 1018–1027, Jul. 2010.
[17] T. J. Slegel, R. M. Averill, III, M. A. Check, B. C. Giamei, B. W. Krumm, C. A. Krygowski, W. H. Li, J. S. Liptay, J. D. MacDougall, T. J. McPherson, J. A. Navarro, E. M. Schwarz, K. Shum, and C. F. Webb, "IBM's S/390 G5 microprocessor design," *IEEE Micro*, vol. 19, no. 2, pp. 12–23, Mar.–Apr. 1999.
[18] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Trans. Reliab.*, vol. 51, no. 1, pp. 63–75, Mar. 2002.
[19] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software implemented fault tolerance," in *Proc. Int. Symp. Code Gener. Optimization*, 2005, pp. 243–254.
[20] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," *IEEE Trans. Reliab.*, vol. 51, no. 1, pp. 111–122, Mar. 2002.
[21] J. Lee and A. Shrivastava, "A compiler optimization to reduce soft errors in register files," *ACM SIGPLAN Not.*, vol. 44, pp. 41–49, Jul. 2009.
[22] X. Li, S. V. Adve, P. Bose, and J. A. Rivers, "Online estimation of architectural vulnerability factor for soft errors," *SIGARCH Comput. Archit. News*, vol. 36, no. 3, pp. 341–352, 2008.
[23] K. R. Walcott, G. Humphreys, and S. Gurumurthi, "Dynamic prediction of architectural vulnerability from microarchitectural state," *SIGARCH Comput. Architecture News*, vol. 35, no. 2, pp. 516–527, 2007.
[24] A. Shrivastava, J. Lee, and R. Jeyapaul, "Cache vulnerability equations for protecting data in embedded processor caches from soft errors," *ACM SIGPLAN Not.*, vol. 45, pp. 143–152, Apr. 2010.
[25] J. Lee and A. Shrivastava, "Static analysis to mitigate soft errors in register files," in *Proc. Int. Conf. DATE*, 2009, pp. 1367–1372.
[26] K. Chen, S. Malik, and D. I. August, "Retargetable static timing analysis for embedded software," in *Proc. ISSS*, 2001, pp. 39–44.
[27] S. E. Richardson and M. Ganapathi, "Interprocedural analysis versus procedure integration," *Inform. Process. Lett.*, vol. 32, pp. 137–142, Aug. 1989.
[28] G. Kildall, "A unified approach to global program optimization," in *Proc. Symp. POPL*, 1973, pp. 194–206.
[29] S. Muchnick, *Advanced Compiler Design and Implementation*. San Mateo, CA: Morgan Kaufmann, 1997.
[30] J. Kam and J. Ullman, "Monotone data flow analysis frameworks," *Acta Informatica*, vol. 7, no. 3, pp. 305–317, 1977.
[31] K. Cooper, T. Harvey, and K. Kennedy, "Iterative data-flow analysis, revisited," Rice Univ., Houston, TX, Tech. Rep. TR04-100, 2004.
[32] M. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proc. IWWC*, 2001, pp. 3–14.
[33] D. Burger and T. Austin, "The SimpleScalar tool set, version 2.0," *SIGARCH Comput. Architect. News*, vol. 25, no. 3, pp. 13–25, 1997.
[34] *Open-Source Mixed-Integer Linear Programming System* [Online]. Available: http://lpsolve.sourceforge.net/5.5
[35] Y. Wu and J. Larus, "Static branch frequency and program profile analysis," in *Proc. PMICRO 27*, Nov. 1994, pp. 1–11.

Jongeun Lee (S'01–M'11) received the B.S. and M.S. degrees in electrical engineering, and the Ph.D. degree in electrical engineering and computer science, all from Seoul National University, Seoul, Korea, in 1997, 1999, and 2004, respectively.

He is currently an Assistant Professor with the School of Electrical and Computer Engineering, Ulsan National Institute of Science and Technology, Ulsan, Korea. His current research interests include reconfigurable architectures, compilers for low power and reliability, and design and optimization of embedded systems.

Aviral Shrivastava received the B.S. degree in computer science and engineering from the Indian Institute of Technology Delhi, New Delhi, India, and the M.S. and Ph.D. degrees in computer science and engineering from the University of California, Irvine.

He is currently an Assistant Professor with the Department of Computer Science and Engineering, Arizona State University, Tempe. His current research interests include compilers, computer architecture, and very large scale integration computer-aided design, with a particular focus on compiler, microarchitectural, and compiler-microarchitecture hybrid techniques for improving power, performance, temperature, codesize, reliability, and robustness of embedded and multicore processor systems.