A Compiler-Microarchitecture Hybrid Approach to Soft Error Reduction for Register Files

Jongeun Lee and Aviral Shrivastava

***Abstract*—For embedded systems, where neither energy nor reliability can be easily sacrificed, this paper presents an energy efficient soft error protection scheme for register files (RFs). Unlike previous approaches, the proposed method explicitly optimizes for energy efficiency and can exploit the fundamental tradeoff between reliability and energy. While even simple compiler-managed RF protection scheme can be more energy efficient than hardware schemes, this paper formulates and solves further compiler optimization problems to significantly enhance the energy efficiency of RF protection schemes by an additional 30% on average, as demonstrated in our experiments on a number of embedded application benchmarks.**

***Index Terms*—Compiler-architecture hybrid, embedded processor design, energy, partially protected register file (PPRF), register file vulnerability (RFV), reliability.**

I. Introduction

POWER DENSITY and reliability have risen to become first-class design concerns in the deep sub-micrometer fabrication era. On one hand, power density has increased so much that we cannot operate processors at the maximum possible clock frequency determined by design; on the other hand, the basic computational units, i.e., transistors, have become extremely susceptible to soft errors. Even a slight variation in signal voltage, noise in the power supply, or a cosmic particle strike can toggle the logic value of a transistor, eventually causing a system failure [1]. There is a clear need of techniques to mitigate the impact of soft errors at minimal power overhead. This need is aggravated by the fact that low-power designs using lower supply voltage and lower threshold voltage will result in higher soft error rates. Register file (RF) is most affected by both of these tightly coupled effects, since it is both the hottest component in the processor [2], [3] and also extremely susceptible to soft errors [4].

Fault-tolerant designs [5] can certainly enhance the reliability of a register file, through parity or error correcting code (ECC) added to every register, or by maintaining a duplicate copy of each register. The International Business Machines (IBM) G5 enterprise server, for instance, has all its registers and latches protected with parity or ECC [6]. However, error checking, especially that based on ECC, has a large overhead in terms of area, runtime, and energy [7], [8]. While the latency of ECC may be hidden by parallelizing ECC with other operations, the area and energy overhead cannot be. Thus, the real question is how to reduce the soft error rate with minimal cost, power, and energy overhead.

Previous paper on cost- and energy-efficient soft error reduction for register files can be divided into two groups. First, hardware approaches ([4], [9]) typically have a very small subset of registers protected, and try to maximize the utilization of the protected ones. Such a *partial* protection can bring down the cost of RF protection by reducing the size of ECC arrays, for instance, to a fraction compared to protecting all registers, while at the same time being effective at reducing soft error rates by protecting only the most important variables. Hardware approaches have 100% software compatibility and have no performance overhead, provided that the additional hardware does not increase the cycle time or register access latency; however, the power overhead is generally high. The other goup is software approaches ([8], [10], [11]). Exploiting the fact that soft errors can occur only to live variables, different instruction scheduling or different register allocation can result in lower expected soft error rates in register files [8], [12]. Software approaches do not require special hardware, but recompilation is a must, and some performance degradation seems unavoidable.

Hybrid approaches can provide a better alternative for soft error reduction in register files. Exploiting hardware features such as partially protected register file (PPRF), hybrid approaches can reduce soft error rates much more effectively than software approaches. Very little performance degradation or even no degradation can be achieved by limiting the amount or degree of code change. Further, power overhead can be substantially reduced by changing runtime decisions in hardware approaches to compile-time decisions. Lastly, the particular hybrid scheme that we propose in this paper uses a postlink optimization, which does not disturb the compiler optimization, and can be applied even when the source code is not available.

Manuscript received March 7, 2009; revised November 10, 2009. Date of current version June 18, 2010. This research was partially funded by the National Science Foundation (NSF), under Grants CCF-0916652, IIP-0856090, and NSF I/UCRC for Embedded Systems, by Microsoft Research, by Raytheon, by the SFAz, by the Stardust Foundation, and by the Basic Science Research Program through the National Research Foundation of Korea (funded by MEST), under Grant 2010-0011534. The authors would like to thank all the members of the Compiler Microarchitecture Laboratory for their valuable support in this paper. This paper was recommended by Associate Editor S. A. Edwards.

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 85287 USA (e-mail: aviral.shrivastva@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.2010.2049050

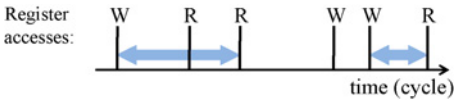

Fig. 1. Computing register vulnerability (shown as thick arrow) from a register access trace. "W" means a write and "R" means a read.

Our approach is based on software-managed PPRF, which is much simpler than hardware PPRF, and allows for highly power-efficient soft error reduction assisted by software optimization. The software optimization is to find the best register allocation that minimizes both the soft error rate and the energy overhead. While there is one previous hybrid approach [8] based on a similar idea, it does not consider energy efficiency, and therefore the software optimization problem becomes trivial. Our approach explicitly maximizes the energy efficiency through static register reallocation, which can easily exploit the fundamental tradeoff between reliability and energy. We formulate and analyze important postlink optimization problems and present efficient algorithms for some of them. Our experiments using embedded application benchmarks demonstrate that our proposed optimization schemes can increase the energy efficiency of RF protection, by 30% on average as measured by our cost metric on register file reliability[1] and energy overhead, compared to a simple optimization considering vulnerability only.

The rest of this paper is organized as follows. Section II introduces a metric for register file reliability and briefly discusses full-hardware PPRF. Section III presents our solution consisting of a software-managed PPRF architecture and postlink software optimizations. Section IV discusses how to obtain profile information required by our approach. Section V presents our experimental results evaluating the effectiveness of the proposed techniques. Section VI discusses related work, and Section VII concludes this paper.

II. Background

A. Register File Vulnerability

To model the soft error rate of RF at the microarchitectural level, we define *register file vulnerability (RFV)* along the lines of architectural vulnerability factor [13]. A register is *vulnerable* at any moment in time, if the next access to the register will be a read by the processor, including stores into the memory. A register is not vulnerable if it will be simply overwritten. A contiguous duration of time during which a register is vulnerable is called a *vulnerable interval* of the register. A register may have multiple vulnerable intervals during a program execution. Thus, given a program execution the vulnerability of a register, or simply the register vulnerability, is defined as the combined length of its vulnerable intervals, as illustrated in Fig. 1. Finally, the RFV is simply the sum of vulnerabilities of all registers.

B. Full-Hardware Partially Protected Register File

Full-hardware PPRFs [4], [9] are proposed to avoid the large overhead of protecting all registers in a processor. The overhead of protecting all registers can be large, since protecting a register involves not only extending the register with a few extra bits for the redundant information such as ECC, parity, or duplicate, but also generating or checking the redundant information on every access to the register. PPRF aims to reduce the protection overhead without necessarily sacrificing the protection quality much, by protecting only the most important variables. The importance of a variable can be measured by its contribution to the RFV, and thus, a variable with a longer live range is preferred to one with a shorter live range or a dead variable [9].

To implement such a preferential scheme however, the control logic of full-hardware PPRF can become rather complicated. The Shield architecture [9], for instance, has a prediction and decision logic, which, on every register write, determines whether to protect the variable by predicting its live range. If it decides to protect the variable, then it activates the ECC generator to compute the ECC code, which is stored into a small ECC array; otherwise, ECC computation is skipped. This scheme allows for not only significant reduction of ECC power, but also sharing of ECC generators, as ECC generation can take several processor cycles [7], [9]. Similarly, on every register read, instead of always checking the ECC code (ECC checking has a similar complexity as ECC generation), the control logic first determines whether the variable is being currently protected, and only if it is, performs ECC checking. Thus, a full-hardware PPRF requires a hardware logic to monitor every register access to determine if the variable is, or should be, protected, in addition to the generation and checking of redundant information. Our software-managed PPRF aims to remove this nonessential hardware logic by encoding the variable protection information in the software itself.

III. Our Proposed Solution

A. Architecture: Software-Managed PPRF

An important issue with software-managed PPRF is how to specify which registers to protect without altering the instruction set architecture. Adding new instructions or modifying existing instructions to encode register protection information has many disadvantages, including software compatibility problem and unwanted design changes that are likely to affect the processor speed negatively. Thus we use a very simple approach: hardware protects only the K highest numbered architectural registers, where K is a design parameter. Then, the responsibility of utilizing the K protected registers is on the programmer or the compiler, but they have the freedom to choose their own objectives such as minimizing the RFV, the energy, or some combination of the two.

At first, the idea of protecting only the K highest numbered registers may seem odd because registers often have specific roles and they are not very interchangeable with each other. For example, the global pointer register (`gp`) is used exclusively to hold the global pointer, and is not even subject to register allocation. However, such a role definition for registers needs to be respected only within a program (across different parts of a program), and swapping two registers throughout

[1]See Section II-A for our definition of register file reliability.

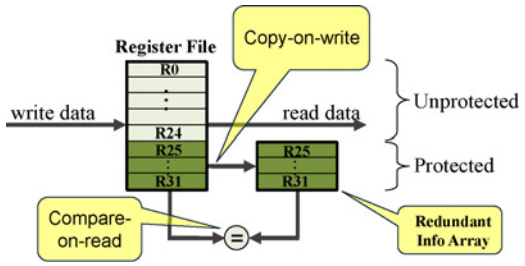

Fig. 2. Software-managed partially protected register file architecture. The underlying protection mechanism can be any of ECC, parity, or duplication, but this figure uses duplication for a simpler illustration.

the entire program does not alter the behavior of the program, except for a few registers.

Fig. 2 illustrates the architecture of our software-managed PPRF. The underlying protection mechanism can be either ECC, parity, duplication, or any combination of the three, but duplication is used in the figure for a simpler illustration. Compared to a full-hardware PPRF, our software-managed PPRF lacks the decision logic, and always protects K highest numbered registers. Consequently the redundant information array (shown next to the protected registers) is also greatly simplified: 1) it is directly addressed unlike a fully-associative content addressable memory, which is used in the Shield architecture (the accompanying tag array in Shield is also unnecessary); and 2) it can be merged with the original register file, sharing the address decoder. While the hardware becomes simpler, the software now should be optimized to best utilize the K protected registers.

B. Software Optimization: Register Swapping

The problem of software optimization for our software-managed PPRF is how to find the best register allocation that minimizes both the soft error rate and the energy overhead. There are at least two approaches: 1) complete register reallocation optimizing for both the traditional objectives (such as performance and code size) and new objectives (such as RFV and energy); or 2) register swapping[2] after first performing traditional (e.g., performance-optimal) register allocation. While the first approach should be, in theory, no worse than the second in terms of the optimization quality, there are practical issues. The first approach is extremely complicated, since even the traditional register allocation is an NP-hard problem [14]. The performance may be degraded as a result of register re-allocation. More importantly, register re-allocation requires recompilation and thus the source code, which practically excludes all library functions from its scope. The second approach is a postlink optimization, and has several advantages: no significant performance degradation, no recompilation or source code necessary, and applicability to optimized binaries and libraries. We take the second approach.

We consider two register swapping strategies. One is program-level register swapping (PRS), which is to change the register assignments at the program level. That is, a pair of registers swapped in one function are swapped in every other function. To capture the register swapping information in PRS we need only one register reassignment vector (RRV), which is common for all functions. The second strategy we employ is function-level register swapping (FRS), which can change register assignment differently in each function. To capture the register swapping information in FRS we need one RRV per function, or a register reassignment table (RRT) for the entire program. FRS must respect the calling convention; otherwise, either copy instructions should be inserted around function calls and returns to ensure correct behavior, or different functions will not be able to work together. Thus, only caller-saved registers (or t-registers[3]) and callee-saved registers (or s-registers[3]) can be swapped within their own register groups. This leads to two versions of the FRS problem, which are referred to as FRS/t and FRS/s, for t-registers and s-registers, respectively. On the other hand, PRS can be applied for all registers, since the calling conventions do not apply when changes are made consistently throughout the entire program (we assume statically linked libraries). The only exceptions are the registers that are reserved for system calls and those that are architecturally distinguished, or treated differently by the instruction set. An example of architecturally distinguished register is the link register or `r31` in the MIPS architecture [15], since `jal` instruction implicitly writes to `r31`.

C. Problem Formulation

1) *Objective:* Our optimization goal is to maximize reliability per energy, or minimize the product of vulnerability and energy as reliability is the inverse of vulnerability.[4] Let ΔV be the vulnerability of RF and ΔE be the energy *overhead* due to RF protection. The total vulnerability and energy of the system can be written as $V_0+\Delta V$ and $E_0+\Delta E$, where V_0 is the vulnerability of the system without RF and E_0 is the energy consumption of the system without RF protection. Then our objective function can be written as follows, assuming that $E_0 \gg \Delta E$ and $V_0 \gg \Delta V$:

$$\begin{aligned} \min \quad EV &\Leftrightarrow \min \quad (E_0+\Delta E)(V_0+\Delta V) \\ &\Leftrightarrow \min \quad E_0\Delta V + V_0\Delta E + \Delta^2 \\ &\Leftrightarrow \min \quad \Delta V + \alpha \Delta E \end{aligned} \tag{1}$$

where $\alpha = V_0/E_0$ is a constant. Note that we can also optimize for vulnerability only simply by setting α to zero.

2) *Energy Model:* The energy overhead of protecting a register using redundancy-based schemes (e.g., ECC, parity, and duplication) has two components. The dynamic component is proportional to the number of reads and writes to the register whereas the static (leakage) component is proportional to the runtime. Since we are considering software optimization for a given architecture and because our optimization does not affect the application performance, the static energy is

[2]Swapping registers i and j in the assembly code means replacing all ocurrences of register i with j and those of j with i.

[3]Following the MIPS architecture's register naming convention.

[4]Considering the energy overhead is important, since if we could afford to spend more energy in RF, we could have spent the energy to make other components more reliable too.

constant irrespective of our optimization. For dynamic energy, we can approximate the energy overhead with the number of accesses to protected registers, since accessing unprotected registers does not contribute to the energy overhead and each access to protected registers will have an equal contribution—unless read and write accesses have different per-access energy consumptions.[5] To account for this asymmetry in per-access energy, read and write may be weighted differently according to the energy model of the particular protection mechanism. Thus, the energy overhead of RF protection is

$$\Delta E = c_r A_{RF}^{(r)} + c_w A_{RF}^{(w)} \tag{2}$$

where c_r, c_w are the per-access energy overhead of protected registers (read/write), and $A_{RF}^{(r)}, A_{RF}^{(w)}$ are the number of reads and the number of writes to protected registers. For a symmetric energy model: $\Delta E = c \cdot A_{RF}$, where c is the per-access energy overhead of protected registers.

3) *Effect of Register Swapping:* Since the PRS problem is a special case of the FRS problem with only one function (even though PRS has a larger scope than FRS), we consider formulation of the FRS problems only. Let R be the number of registers under consideration (either t-registers, s-registers, or PRS-swappable registers) and N be the number of functions in the program including library functions. Then, any solution of the FRS problem can be represented as a RRT $T = \{\vec{\rho}^f \mid f = 1, 2, \ldots, N\}$, each of which is a RRV, or a permutation of integers from $1, 2, \ldots, R$. A RRV $\vec{\rho}^f = \{\rho_1^f, \rho_2^f, \ldots, \rho_R^f\}$ of function f implies that all the occurrences of register ρ_r^f in the assembly code of function f should be replaced with register r in the transformed program. To put it another way, all the variables originally assigned to register ρ_r^f are now assigned to register r after transformation.

4) *Problem:* How the registers are swapped, or RRT, can substantially change the RFV and the energy overhead of a program. Thus our optimization problem is, given a program and input parameters (R, K, α, c), to find the RRT that minimizes the objective function defined in (1). Unfortunately this problem is ill-defined, since the RFV and the energy overhead depend on the execution trace as well as the program and register swapping. Therefore to make the problem formulation complete, let us assume a program execution trace (e.g., basic block sequence), which is not changed by our software optimization. Note that execution trace is introduced just for the sake of problem formulation and our proposed solution does not require an expensive execution trace. Now, given a program P, an execution trace of the program X, and register swapping information T, one can uniquely determine the RFV and the energy overhead. The problem is then to find T that minimizes (1).

While this problem is interesting, we can think of a more general one. Suppose that we want to find a RRT T that will minimize our objective function for all values of K. This can be useful, for instance, if we want to optimize the software for all implementations of PPRFs with different values of K, or if the parameter K can be software-controlled at runtime to dynamically trade off between energy consumption and vulnerability depending on runtime requirements. To find a single objective function for all values of K, we take the expectation of our objective function assuming a uniform distribution on K with K varying from 0 to R. First, let V_r and A_r be the vulnerability of and the number of accesses to register r, which can be deterministically computed from (P, X, T). Then, the RFV with K protected registers is $V^{(K)} = V_1 + V_2 + \cdots + V_{R-K} = \sum_{r=1}^{R-K} V_r$, since the K highest numbered registers are protected in hardware. Similarly, the energy overhead with K protected registers is $E^{(K)} = c \cdot A_{R-K+1} + \cdots + c \cdot A_R = c \sum_{r=1}^{K} A_{R-K+r}$, assuming that read and write have the same energy overhead per access. Finally, taking the expectation of $V^{(K)} + \alpha E^{(K)}$ on K, we get a new function, which we refer to as the cost function C

$$\begin{aligned} C &= \frac{1}{R+1} \sum_{K=0}^{R} \left(\sum_{r=1}^{R-K} V_r + \alpha \cdot c \sum_{r=1}^{K} A_{R-K+r} \right) \\ &= \frac{1}{R+1} \sum_{r=1}^{R} (R+1-r) V_r + \beta r A_r \end{aligned} \tag{3}$$

where $\beta = \alpha \cdot c$ is a constant, and setting $\beta = 0$ gives the vulnerability-only version of the same problem.

We distinguish optimization problems that consider vulnerability only, as some of them can be trivially solved. Vulnerability-only versions are denoted by prefix V such as V-PRS, V-FRS/t, and V-FRS/s, whereas energy-efficiency versions are denoted by prefix energy efficiency (EV) such as EV-PRS, EV-FRS/t, and EV-FRS/s. Among those the most interesting ones are EV-FRS/t and EV-FRS/s, which are discussed next.

D. Solutions to FRS/t and PRS Problems

FRS/t problems are much simpler than FRS/s problems. In any function the first access to a t-register must be a write, and t-registers, if used, should be vacated before making a function call. Therefore the live range of a t-register is confined to one function, and when a function calls another (which divides the caller function into parts), the live range of a t-register used in the caller function is confined to each part of the caller function. Consequently the register allocation in one function does not alter the vulnerability or the access counts of registers in another function. Nested function calls do not complicate the matter.

1) *Vulnerability Only:* Optimizing for vulnerability only simply means to use protected registers as much as possible. Thus the V-PRS problem, for instance, can be solved by first finding out the vulnerability of each register for the entire application and then swapping the register assignments so that higher numbered registers will have greater vulnerability than lower number ones. Extension to V-FRS/t is also trivial, since register allocation in one function do not affect vulnerability in another.

2) *Energy Efficiency (EV):* However, optimizing for energy efficiency is not very intuitive. One intuition says that sorting the registers by vulnerability-to-access-count ratio so that

[5]For instance, in a duplication-based scheme a protected read involves not only reading a duplicate but also a comparison between two register values, whereas a protected write involves writing a duplicate value only.

higher-numbered registers will have higher vulnerability and lower access count than lower-numbered ones, will give overall good energy efficiency. Likely as it may sound, the result is not always as expected. Without an efficient algorithm even the EV-PRS problem can be very difficult to solve optimally. A naïve approach evaluating all the $R!$ register orderings, for instance, is practically infeasible, since the number of possible swapping for $R = 28$ (the number of PRS-swappable registers in the MIPS architecture) is $28! > 10^{29}$.

Fortunately, there is an efficient algorithm for EV-FRS/t and EV-PRS problems. For t-registers, the register vulnerability in a function can be fully determined from the function itself.[6] Thus by defining v_r^i and a_r^i as the vulnerability and the access count of register r in function i in the original program, we can now represent V_r and A_r in terms of v_r^i, a_r^i, and $T = \{\vec{\rho}^{\,i} \mid i = 1, 2, \ldots, N\}$

$$V_r = \sum_{i=1}^{N} v^i_{\rho^i_r} \qquad A_r = \sum_{i=1}^{N} a^i_{\rho^i_r}. \tag{4}$$

Then, substituting (4) into (3) and changing the order of summations reveals that the cost function C can be minimized simply by minimizing C for each function, or C' as defined

$$C'(\vec{\rho}) = \sum_{r=1}^{R} (R + 1 - r) v_{\rho_r} + \beta r a_{\rho_r}. \tag{5}$$

Lemma 1: For two registers i and j, where $v_i - \beta a_i < v_j - \beta a_j$, if any register ordering $\vec{\rho}$ puts register i in a higher-numbered register than j, swapping the two registers i and j always gives a lower C' value defined by (5).

The proof is straightforward, and can be done [16] by comparing $C'(\vec{\rho})$ with another register ordering $C'(\vec{\sigma})$ where i and j are swapped. Using the lemma, one can find the optimum solution to the EV-FRS/t problem by sorting the registers according to their $(v_i - \beta a_i)$ values for each function. The complexity of this algorithm is $O(NR\log(R))$ for N functions. The EV-PRS problem can also be solved similarly.

E. Solutions to FRS/s Problems

Unlike FRS/t problems, FRS/s problems are very complicated. In any function the first access to an s-register, if it exists, is a read and the last is a write. Therefore the live range of an s-register is not limited to one function but may span several functions. One implication of that is that, unlike with a t-register, an s-register can be vulnerable in a function even if the function does not even use the register, which is the case if the register is first read after the function returns. Consequently the register allocation in one function can alter the vulnerability of s-registers in another function. The complexity is further elevated by nested function calls.

1) *Complexity:* To understand the complexity of the FRS/s problems let us consider two examples illustrated in Fig. 3. In (a), function F1 calls function F2, which returns without calling any function. Below the function call sequence, accesses to a particular s-register are shown on a time axis, represented by small vertical bars and annotated with time and type. Fig. 3(b) illustrates a different register access pattern for the same function call scenario. In any function the first access to an s-register must be a read and the last must be a write, which necessarily creates vulnerable intervals spanning multiple functions.

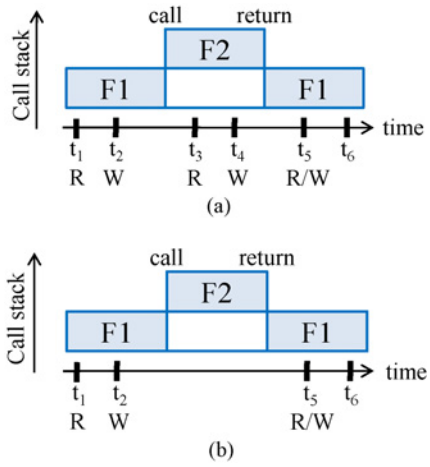

Fig. 3. S-register access patterns with respect to function call. (a) When the register is not used in callee function. (b) When the register is used in callee function.

1) *Callee vulnerability depends on caller:* Consider the interval (t_1, t_2), which includes a function call. This interval is vulnerable since t_3 must be a read. However, another interval (t_4, t_5), which includes a function return, may or may not be vulnerable since t_5 can be either a read or a write. Therefore the vulnerability of F2 (callee), and hence the optimal RRV for F2, depends on the RRV for F1 (caller).
2) *Caller vulnerability depends on callee:* Now suppose that not all s-registers are used in F2. Then, with certain register reassignments in F2, it is possible that the s-register in question has no access in F2, as illustrated in Fig. 3(b). In this case, the interval (t_2, t_5) may or may not be vulnerable depending on the type of access at t_5. If the access at t_5 is indeed a write, the interval from t_2 until the function call is not vulnerable in (b), but is vulnerable in (a). Thus, the caller vulnerability and its optimal RRV can also depend on the callee.

This inter-dependence between functions exists for all caller–callee pairs unless every s-register is used in the callee and on every invocation of the callee function. Since all functions are connected through caller–callee relationship, optimizing for one function in general depends on the optimization of every other function. This tight inter-dependence between functions makes it very unlikely to find an efficient algorithm. Exhaustive search has $O(R! \cdot R! \cdots R!) = O(R^{RN})$ complexity, and without breaking the inter-dependence the best we can get is $O(R^N)$ (if a linear-time algorithm is used to determine the register swapping of a function), which is still exponential.

[6]For s-registers, the register vulnerability in a function cannot be determined without considering other functions as well.

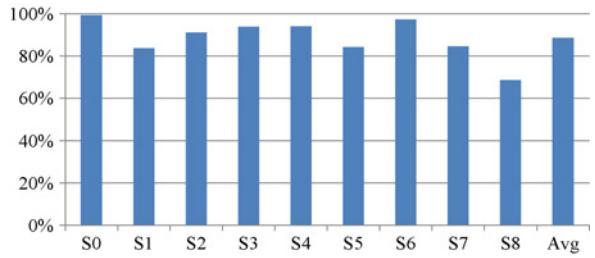

Fig. 4. Probability of s-registers being first read after function call or return (for jpeg).

TABLE I
COMPARISON OF PROBLEMS

Method (Scope)	Vulnerability Only	Energy Efficiency (EV)
PRS (almost all regs)	Very simple	Efficient algorithm exists
FRS/t (t-registers)	Very simple	Efficient algorithm exists
FRS/s (s-registers)	Finding optimum is very complex; use heuristic	

2) *Heuristic:* The key observation for our heuristic to the FRS/s problem is that s-registers are much more likely to be first read after control is transferred to another function. Fig. 4 shows the probability of s-registers being first read after a function call or return for jpeg. The statistics are obtained during simulation by keeping track of all s-registers after every function change. Thus for every function change (i.e., call or return) at runtime, an s-register can have either read-follow or write-follow attribute, which is resolved as simulation progresses. After simulation, read-follows are collected for each register and the ratio of read-follows over the total number of function changes gives the probability shown in the graph. Clearly s-registers are read most of the time after function change. This is in part because, after a function call, s-registers must be first read if they are used at all. Exploiting this observation we assume that the next access after function change is always a read, which makes s-register vulnerability calculation independent of other functions. Then, our FRS/s problem becomes identical to the FRS/t problem, and we can apply FRS/t optimization algorithm to FRS/s problems.

Another interesting observation that can be made from the graph is the abundance of write-after-function-return cases, which was a basis for our argument for caller–callee inter-dependence. Since in any program execution the number of function calls should be exactly the same as the number of function returns, the probability of s-registers being first read after function return is no more than half[7] the probability of the register being first read after function call or return, the latter of which is plotted in the graph. For instance, the graph indicates that `s8` is first read 68 per cent of the time, which means that it is first read after function return 34 per cent of the time or less. In other words, `s8` is first written to two-out-of-three after function return, which validates our premise on caller–callee inter-dependence in FRS/s optimization.

Table I summarizes the six problems we consider, among which PRS problems are special cases of FRS/t problems.

[7]It can be less than half, since a register may be read after a function call but written to as soon as the function returns.

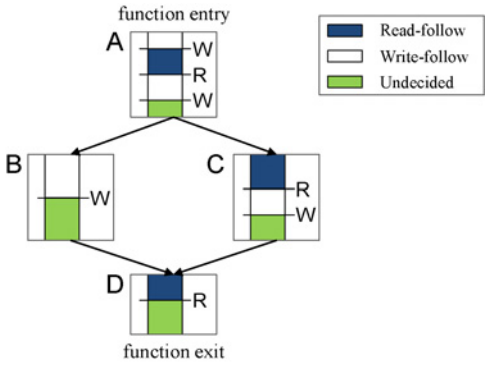

Fig. 5. Register access pattern and vulnerable intervals in a function with four basic blocks.

Optimizing FRS/t for vulnerability only is obvious, and for its energy efficiency version, which is less intuitive, we present an efficient algorithm. The FRS/s problems present a greater challenge, for which we propose a simple heuristic.

IV. VULNERABILITY PROFILING

A. Challenge and Problem

Our proposed algorithm requires the vulnerability and access count information of each register for every function in a program. Also for PRS the vulnerability of each register for the entire program is required. This profile information is certainly much less than an execution trace, which grows in length as runtime increases. Still, obtaining such information, especially register vulnerability, can be challenging. The challenge is that unlike access count, register vulnerability depends on what comes next. For instance, the vulnerability of an s-register in a function cannot be determined without knowing whether the register will be first read or write after the function returns, which is why the function vulnerability of an s-register cannot be determined [10]. This difficulty is not limited to s-registers only. Even t-registers cannot have basic block vulnerability uniquely determined without knowing what other blocks will come next, making it impossible to get exact register vulnerability from the execution count of each basic block alone.

Fig. 5 illustrates an example access pattern of a t-register in a function with four basic blocks. Since it is a t-register it should be first written to in basic block A, which is the entry of the function. But in other basic blocks it may be first read (e.g., C, D) as well as written to (e.g., B). Thus if we divide the whole time when the function is running into intervals, some intervals are definitely vulnerable (those ending with a read; painted dark in the figure), some definitely not vulnerable (those ending with a write; painted white), but there are some intervals whose vulnerability depends on other blocks (painted light). The last type of intervals (i.e., the undecided ones) should be resolved by profiling. In particular, since the length of each interval can be known, at least in terms of instruction counts, we need only find out how often each undecided interval is followed by read versus write at runtime.

B. Solution

Every basic block has one undecided interval (for each register), which is the interval from the last access until the end of the basic block. If the register is not accessed at all in a basic block, the entire length of the basic block becomes an undecided interval. Finding the read-follow or write-follow frequencies of undecided intervals through profiling is similar to finding branch probabilities. In essence one monitors every branch instruction and records the outcome in terms of read-follow versus write-follow.

To avoid searching for the first access in each basic block at runtime, we build at compile-time a table associating basic block IDs with their first-access-type attributes. The first-access-type attribute can be either read, write, or no-access. The no-access value means that the particular register is not accessed at all in the basic block. If the program branches to a block with no-access value, we cannot immediately resolve read-follow versus write-follow. The decision is pending until the program eventually reaches a block with read or write value. We keep a list of undecided intervals with pending decisions so that they can be revisited when the decision is made.

Computing register vulnerability is straightforward, once the read-follow frequencies are found out through profiling. It can be calculated as the sum of interval lengths weighted by their read-follow frequencies, which is accurate as much as the interval lengths are accurate [10].

V. Experiments

A. Experimental Setup

To evaluate the effectiveness of the proposed optimizations we use applications from the MiBench benchmark suite [17]. Applications are compiled using GCC 2.7.2.3, which is one of the latest versions supporting the SimpleScalar target, with the benchmark-specified optimization levels. The binaries are then transformed using our register swapping optimizations. The profile information required by our optimization is obtained from an initial run of the application. Finally the modified binaries are simulated again to find out the vulnerability and energy overhead impact of our optimizations. As for the simulator we use an extended version of the SimpleScalar performance simulator [18], configured for in-order execution to model embedded processors. As discussed in Section III-C.2, we use access count to approximate energy overhead of RF protection. The target instruction set is based on the MIPS architecture [15], and has 11 t-registers and 9 s-registers.[8] For our optimization, β is set to the overall vulnerability-to-access-count ratio of RF in the original program.

B. Comparison Between Optimization Methods

To see the effectiveness of our proposed methods (EV-FRS, EV-PRS), we compare them with simple compiler approaches (V-FRS, V-PRS).[9] Fig. 6(a) and (b) plots the RFV (in cycle* word) and energy overhead (as the number of accesses), optimized by the four methods for different values of K (# protected registers). Optimizations are applied to s-registers only. As expected, larger K values reduce RFV but increase energy overhead. When $K = 9$, all s-registers are protected, and thus, there is no difference between the four methods. When $K = 0$, however, different methods have slightly different RFV values. This is because software optimizations were still performed even when $K = 0$. And the differences show that RFV can be reduced by using a different register allocation even without protected registers.

Since all four methods try to minimize RFV by using protected registers more often, they show little difference in RFV reduction, with V-FRS slightly leading. However, their energy overheads show a large variance, with more than 3 times difference between the highest (V-FRS) and the lowest (EV-FRS). To compare them in terms of energy efficiency, we

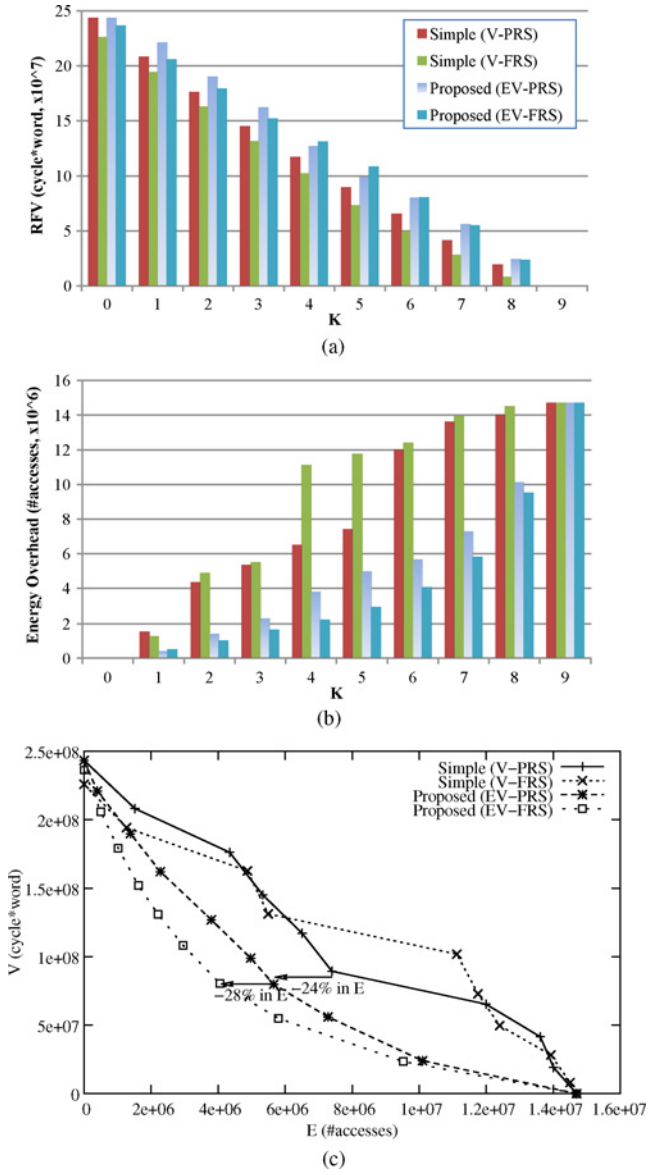

Fig. 6. Trade-off between vulnerability and energy overhead by different optimization methods. Dots represents different values of K. For s-registers and using jpeg: (a) vulnerability (V), (b) energy overhead (E), (c) *V-E plot*.

[8]T-registers are `r1`, `r8` through `r15`, and `r24` and `r25`, and s-registers are `r16` through `r23` and `r30`.

[9]We use V-PRS and V-FRS as our baseline, since they are the most straightforward optimizations for PPRF.

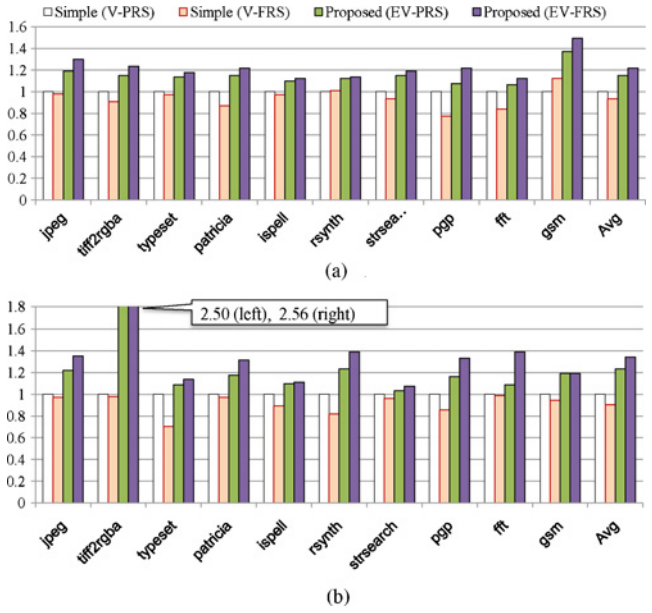

Fig. 7. Comparing energy efficiency of the proposed optimizations with simple compiler optimizations (normalized to V-PRS results). (a) For s-registers. (b) For t-registers.

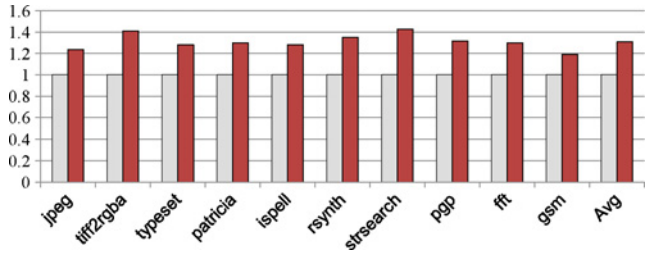

Fig. 8. Comparing energy-efficiency of vulnerability-only optimization (left) versus our energy efficiency optimization (right).

plot the V-E graph in Fig. 6(c). The graph clearly shows the trade-off between vulnerability and energy overhead, and the lower-left corner is the most energy efficient region. While the simple compiler approaches have dots near the diagonal line, our proposed methods have dots near a concave curve facing the energy efficient region. Quantitatively, when $K = 6$, our EV-PRS can achieve similar vulnerability reduction with 24% less energy overhead over simple V-PRS. Further, our EV-FRS can save additional 28% in energy overhead over EV-PRS without increasing vulnerability.

We repeat the same comparison for all the applications. To summarize the results we use the inverse of the cost metric defined in (3) as relative energy efficiency. Fig. 7(a) compares the four methods in terms of energy efficiency for various applications. *Y*-axis is normalized to the energy efficiency of V-PRS. Understandably the aggressive optimization by V-FRS proves to be harmful, decreasing the energy efficiency by up to more than 20% compared to the simplest strategy. On the other hand, our proposed optimizations can increase energy efficiency by 15% (PRS) and 22% (FRS), on average, for s-registers. Results for t-registers are similar.

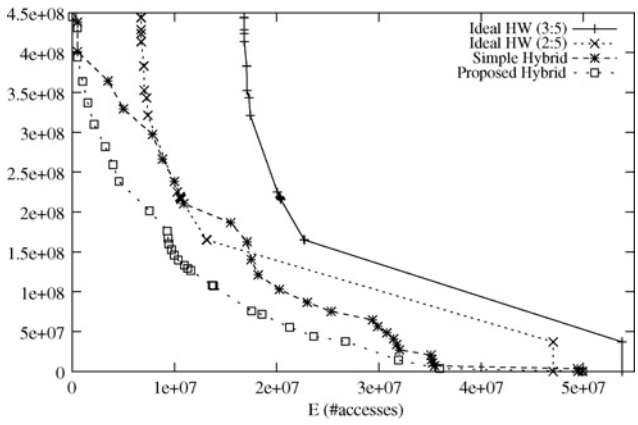

Fig. 9. V-E plot of different RF protection approaches (ideal full-hardware, simple hybrid, and proposed hybrid). For all registers and using jpeg. To represent the energy overhead on the *x*-axis in number of accesses, full-hardware's runtime decision energy is converted to equivalent accesses.

C. Our Optimization Versus Other Compiler-Assisted Approaches

We compare our proposed scheme against other compiler-assisted methods. For our scheme, we first apply EV-FRS to s-registers and then to t-registers, and finally apply EV-PRS to all registers. For other compiler-assisted methods, we simply use V-PRS, which gives higher energy efficiency than V-FRS.[10] In both cases, we assume that `r31` is protected first (protected if $K \geq 1$), and system call registers are protected last. Fig. 8 compares the two approaches in terms of energy efficiency. Clearly our proposed scheme can consistently improve the energy efficiency over simpler vulnerability-only optimization, by up to 42%, and by 30% on average.

D. Compiler-Managed Versus Ideal Hardware-Managed

To demonstrate the energy efficiency of hybrid schemes, we compare our compiler-managed RF protection with full-hardware schemes. Since the energy difference is expected to be high, we compare it against the ideal hardware case rather than comparing it against every hardware scheme. We define the ideal hardware case as follows: 1) sort the runtime variables in the decreasing order of $(v_i - \beta a_i)$, where v_i the lifetime and a_i is the access count of variable i; and 2) allocate protected registers to the variables in the sorted order in a greedy fashion until allocation fails. Variables selected this way are expected to have a very high vulnerability-to-access-count ratio. As for the hybrid approach we consider our energy efficiency optimization (*proposed hybrid*) as well as simple vulnerability-only optimization (*simple hybrid*). The energy model needs to be extended to include the runtime decision energy in the hardware scheme. To simplify the comparison we ignore leakage energy and assume a fixed energy ratio (2/5 or 3/5) between runtime decision operation and redundant information checking or generation operation. The exact energy ratio may vary depending on many factors

[10]We choose V-PRS to approximate Yan *et al.* [8], which is the only other compiler-assisted method that we are aware of. Unfortunately it is described only in a very cursory manner with very little information on how, but it seems to be simple as it considers vulnerability only.

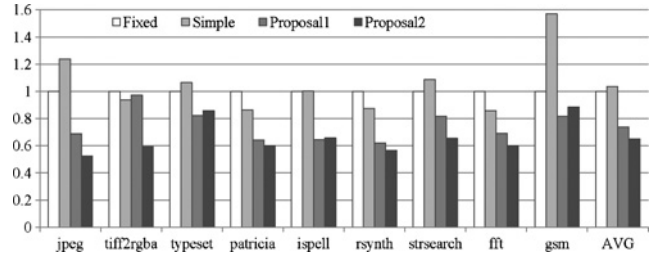

Fig. 10. Comparing the four schemes in terms of $\sqrt{V \cdot E}$, using test data different from training data (normalized to Fixed).

including the protection mechanism, hardware design, and implementation.

Fig. 9 compares the different trade-offs with the approaches. We did this comparison only for one application (jpeg) due to the time-consuming nature of the experiment. From the graph, in the upper left corner, we observe that the hardware approaches have significant energy overhead even when K is very small, which is due to the runtime decision overhead. As K increases, hardware schemes catch up, raising energy efficiency very rapidly, which is probably because we use an ideal allocation scheme. What is interesting is, even compared with the ideal hardware scheme, our proposed hybrid approach exhibits a competitively steep curve, indicating that the compile-time decisions generated by our combined optimizations of FRS and PRS, have indeed very good quality. Note that this is not the case with the simple hybrid approach. Particularly the crossover between the simple hybrid approach and the lower decision-overhead version (2/5) of the ideal hardware suggests that being hybrid alone does not necessarily guarantee its energy efficiency in all cases, but it should be assisted by good software optimization. Overall, our results on jpeg application demonstrate that our hybrid approach combined with the proposed software optimization can outperform the ideal hardware case for all ranges of K and for any energy ratio not less than 2/5.

E. Sensitivity to Input Data Change

As our primary target is embedded applications, our optimizations use application's profile information. While profiling gives the richest information about application behavior, its accuracy depends on input data. Therefore it is interesting to see how well our proposed schemes work when presented with test data that is different from training data. We compare four schemes.

1) *Fixed* is to protect the highest numbered registers regardless of the application (system call registers are excluded).
2) *Simple* is to use V-PRS.
3) *Proposal* 1 is to use EV-PRS.
4) *Proposal* 2 is to use EV-FRS and EV-PRS together.

In all cases but Fixed, the register swapping decisions are made using one of the two inputs provided by the benchmark suite, and evaluation is done using the other input.[11] For evaluation, we use the geometric mean of the RF vulnerability and the energy overhead, which represents the energy *inefficiency* of the protection scheme. The number of protected registers is set to 6.

If our schemes rely "too much" on the profile information, the optimization for one data set could worsen the results for another data set. In the extreme case, rather than optimizing for a wrong input, using a fixed policy could be a better choice. The Fixed scheme represents such a fixed policy.[12] Our results summarized in Fig. 10 show that our schemes perform significantly better than the Fixed scheme. Our schemes can generate optimizations that are consistently more energy efficient than the fixed policy, bringing down the metric by 25% (Proposal 1) and 35% (Proposal 2) on average. This is not easy to achieve as demonstrated by the Simple scheme; only well-guided optimizations can achieve such an improvement. When using only one set of input data, we observe that Proposal 2 consistently outperforms Proposal 1, which is because FRS has much more parameters, and therefore can fit more closely to the application profile. However, the possible overfitting in the case of FRS can make Proposal 2 sometimes worse than Proposal 1 when multiple sets of input data are used, as can be seen in the graph.

VI. Related Work

Many techniques have been proposed to mitigate the impact of soft errors in register files. To reduce the area overhead of adding ECC to every register, Montesinos *et al.* [9] propose a partial RF protection scheme called Shield, which maintains a very small ECC cache for some register variables. To avoid the relatively large overhead of ECC, recent proposals advocate simply replicating register contents. For instance, Blome *et al.* [4] use a small cache to store duplicates of recently accessed register values, and therefore error detection requires only a simple comparison on every read. Similarly, Memik *et al.* [19] propose a technique to replicate some of the register values in unused physical registers in the context of superscalar processors, which often have a number of physical registers. Another interesting variation on register replication is in-register replication [20], which exploits the fact that register values are very often narrower than 16 bits, or half the register width. Such values can be replicated in the same register, by adding small hardware extension.

Software or compiler techniques promise to reduce the impact of soft errors with little or no power increase. They exploit the fact that not every bit contributes equally toward soft errors, as conceptualized in architectural vulnerability factor [13]. Yan and Zhang [8] demonstrate that just altering instruction scheduling can reduce the chances of system failures due to soft errors in register files by 30%. A similar level of soft error mitigation is reported on another compiler technique [8], which maximizes the use of a protected registers with reliability-aware register allocation. Yet, energy is not considered in their register allocation, which then becomes no different than traditional register allocation, except that protected registers are given higher priority. Lee and Srivastava [12] demonstrate that optimized spill code insertion

[11] pgp is excluded since it has only one set of input data.

[12] The Fixed scheme is rather a good policy since it reduces RF vulnerability by almost 30% on average (not shown in the graph) using only 6 out of 32 registers.

can also lead to significant vulnerability reduction with no hardware modification. In such compiler approaches, accurate RF vulnerability estimation [10] is a key. A compile-time method to estimate RF vulnerability is proposed in [10]. Software techniques [11], [21], [22] enhance the reliability of the system by adding duplicate instructions or flow checking code. While they can protect a larger part of the system with no extra hardware, the runtime overhead can be quite significant.

In summary, there is a lack of research that explicitly minimizes energy consumption while at the same time maximizing the reliability of register files. Considering that register files are the hottest block in microprocessors [2], an approach that mitigates soft errors with minimum energy overhead can be very desirable and effective.

VII. Conclusion

We presented a compiler-microarchitecture hybrid approach to highly energy efficient register file protection for embedded systems. While previous approaches concentrated only on maximizing protection mainly through architectural changes or by pure software modifications, we show that hybrid approaches can be far more energy efficient with little software modification. Our approach is based on software-managed PPRF, which is much simpler than full fledged hardware PPRF. While the simpler hardware reduces power overhead substantially, we find that adequate software optimization is key to achieving high energy efficiency. Our approach is to explicitly maximize the energy efficiency through static register reallocation, which can easily exploit the fundamental tradeoff between vulnerability and energy. Our proposed postlink optimizations do not disturb existing optimizations but merely improves reliability with minimum energy overhead by swapping registers both at the function and program levels. We formulate and analyze important postlink optimization problems and present efficient algorithms for some of them. Our experiments using embedded application benchmarks demonstrated that our proposed optimization schemes can increase the energy efficiency of RF protection, by 30% on average as measured by our cost metric on RFV and energy overhead, compared to a simple optimization (V-PRS) considering vulnerability only.

References

[1] 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.

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

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

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

[5] I. Koren and C. M. Krishna, *Fault-Tolerant Systems*. San Mateo, CA: Morgan Kaufmann, 2007.

[6] 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.

[7] R. Phelan, "Addressing soft errors in ARM core-based SoC," *ARM White Paper*. Cambridge, U.K.: ARM Ltd., Dec. 2003.

[8] J. Yan and W. Zhang, "Compiler-guided register reliability improvement against soft errors," in *Proc. 5th ACM Int. Conf. EMSOFT*, 2005, pp. 203–209.

[9] P. Montesinos, W. Liu, and J. Torrellas, "Using register lifetime predictions to protect register files against soft errors," in *Proc. 37th Annu. IEEE/IFIP Int. Conf. DSN*, 2007, pp. 286–296.

[10] J. Lee and A. Shrivastava, "Static analysis to mitigate soft errors in register files," in *Proc. Int. Conf. DATE*, 2009, pp. 1367–1372.

[11] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software implemented fault tolerance," in *Proc. Int. Symp. Code Generation Optimization*, 2005, pp. 243–254.

[12] J. Lee and A. Shrivastava, "A compiler optimization to reduce soft errors in register files," *Assoc. Comput. Mach. Special Interest Group Program. Languages Notices*, vol. 44, no. 10, pp. 41–49, 2009.

[13] 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. Microarchit.*, Dec. 2003, pp. 29–42.

[14] P. Briggs, K. D. Cooper, and L. Torczon, "Improvements to graph coloring register allocation," *Assoc. Comput. Mach. Trans. Program. Lang. Syst.*, vol. 16, no. 3, pp. 428–455, 1994.

[15] K. C. Yeager, "The MIPS R10000 superscalar microprocessor," *IEEE Micro*, vol. 16, no. 2, pp. 28–40, Apr. 1996.

[16] J. Lee and A. Shrivastava, "Compiler-managed register file protection for energy-efficient soft error reduction," in *Proc. ASP-DAC*, 2009, pp. 618–623.

[17] 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. Int. Workshop Workload Characterization*, 2001, pp. 3–14.

[18] D. Burger and T. Austin, "The simplescalar tool set, version 2.0," *SIGARCH Comput. Archit. News*, vol. 25, no. 3, pp. 13–25, 1997.

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

[20] M. Kandala, W. Zhang, and L. Yang, "An area-efficient approach to improving register file reliability against transient errors," in *Proc. Int. Symp. Embedded Comput.*, 2007, pp. 798–803.

[21] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," *IEEE Trans. Rel.*, vol. 51, no. 2, pp. 111–122, Mar. 2002.

[22] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Trans. Rel.*, vol. 51, no. 1, pp. 63–75, Mar. 2002.

Jongeun Lee 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, Gwanak, South 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 embedded systems, reconfigurable architecture, and compilers for low power and reliability.

Aviral Shrivastava received the B.S. degree in computer science and engineering from Indian Institute of Technology, 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 intersection of 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.