Compiler-Managed Register File Protection for Energy-Efficient Soft Error Reduction *

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

Abstract— For embedded systems where neither energy nor reliability can be easily sacrificed, we present an energy efficient soft error protection scheme for register files (RF). Unlike previous approaches, our method explicitly optimizes for energy efficiency and exploits the fundamental tradeoff between reliability and energy. While even simple compiler-managed RF protection scheme is more energy efficient than hardware schemes, this work formulates and solves further compiler optimization problems to significantly enhance the energy efficiency of RF protection schemes by an additional 24%.

I. Introduction

Power density and reliability have risen to become the most important design concerns in the sub-nanometer 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 even cosmic particle strike can toggle the logic value of the 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 the soft error rate increases exponentially with temperature. Register file (RF) is most affected by both of these tightly coupled effects, since it is both the hottest component in the processor [2], and also extremely susceptible to soft errors [3].

The earliest forms of register file protection include ECC and parity checking [4]. However, error checking, especially that based on ECC, has a large overhead in terms of area, runtime, and energy [5, 6]. While the latency of ECC operations can be hidden by parallelizing ECC with other operations, the area and energy overhead cannot be. To reduce the area overhead of RF protection, later techniques only protect a part of the RF [7]. To further reduce overhead of ECC, later schemes simply replicate the registers they intend to protect. Blome et al. [3] uses a small cache to store duplicates of recently accessed register values and thus a simple comparison on every read operation can detect errors in registers. Memik et al. [8] proposes a technique to replicate some of the register values in unused physical registers in the context of superscalar processors, where there are a number of physical registers and register binding is done at runtime. Another interesting variation on register replication is in-register replication [9], which exploits the fact typically a large fraction of register values are narrower than half the register width, or 16 bits. Such values can be replicated in the same register, requiring no significant extra hardware.

All these microarchitectural techniques have persistent power overhead associated with error checking. Compiler techniques on the other hand promise very power-efficient RF protection, either on their own by instruction scheduling or register reallocation to shorten the live ranges of the variables stored in it [6], or by enhancing the effectiveness of microarchitectural techniques of partial RF protection schemes [3, 8, 7] by making the decision of which registers to protect at compile-time.

This work develops compile-time analysis that explicitly incorporates RF power consumption to come up with register renaming that can be used in both, existing pure-compiler techniques, or to enhance the existing microarchitectural schemes to achieve to power efficient RF protection. Our experimental results on embedded application benchmarks from MiBench [12] indicate that even the simplest of such compiler-management schemes can be more energy efficient than hardware schemes. In addition, our explicit optimizations can further increase the energy efficiency by 24% on average, as measured by our cost metric on register file reliability and energy overhead.

II. Background and Motivation

We use *vulnerability* as the measure of reliability. Following the architectural vulnerability factor [10] the vulnerability of a register is defined as the combined lifetime (or the sum of the live range lengths) of variables assigned to it. The live range of a variable is from its definition until its last use and represents the time when useful data is present in the register. Any transient fault occurring to the register during that time period therefore destroys data integrity and can manifest itself into an error. Thus given the same transient fault rate, vulnerability can be used to predict the soft error rate. The vulnerability of a register file is simply the sum of vulnerability of all registers. Vulnerability is fully determined by the register access pattern, and can be controlled by changing the program.

Researchers have found that not all registers are always vulnerable and thus significant vulnerability reduction can be achieved even with only K protected registers that is less than R, the number of registers, or K entries of redundant information for pro-

*This work is partially supported by grants from Microsoft, Raytheon and Stardust Foundation.

TABLE I
ADDITIONAL OPERATIONS OF A PARTIALLY PROTECTED REGISTER FILE (ONLY THE LAST COLUMN IS NECESSARY IN COMPILER APPROACHES)

	Decision	Entry	Redundant inf.
Write	Decide on protection	Allocate one	Generate one
Read	Find if value is protected	Find it	Check it

tected registers [3, 7]. To make the best use of the K entries we should protect only those register values that are being useful. The decision of which register values to protect is done, according to the previous work, at runtime by hardware logic. There are reasons for that. First it is better to flexibly choose K registers during execution than to fix them for the entire execution and for all applications. Second we can consider embedding protection commands into instructions, signaling to protect the destination register, for instance. This however changes the instruction set architecture, which is difficult to justify. Lastly, some versions of the technique were proposed for superscalar architectures. In this case it may be best to treat each physical register separately, which can only be done at runtime.

While a runtime decision approach has the advantage of potentially being able to select the best ones for protection—not only for minimum vulnerability but also for maximum energy efficiency—there is a significant drawback: the runtime decision making itself can be a very costly operation in terms of energy. Table I lists the additional operations of a partially protected register file. For every register access there has to be first a decision whether it is to a protected one or not (column 2). If it is to a protected one, additional operations are needed (columns 3 and 4). With a compile-time decision approach, on the other hand, the decision step is unnecessary, and allocating and finding an entry can also become trivial. Only the operations in the last column need to be done, thus reducing the energy overhead significantly. Particularly because the decision operations must be performed on every register access whereas the other operations are only for accesses to protected ones, the maximum energy overhead reduction can be very large depending on the number of accesses to protected ones. Assuming that 1 out of 4 accesses are to a protected register and the energy cost for columns 2 to 4 is 1:1:5 (per-access) respectively, a compile-time decision approach can reduce the energy overhead of partial protection from 2.5 $(= 1 \cdot 1 + (1+5)/4)$ to 1.25 $(= 5/4)$, or by 50% if the protection decisions are the same in both cases. Note that this energy cost assumption is overly pessimistic for compiler techniques, even when there is a huge energy overhead reduction, indicating greater energy reductions in real cases. This clearly motivates compiler approaches for energy-efficient vulnerability reduction in register files.

III. COMPILER-MANAGED REGISTER FILE PROTECTION

A. Architecture and Compiler Assumptions

One of the issues with compile-time decision is how to specify which registers to protect without altering the instruction set architecture. There is a very simple solution: use the register number. The architecture protects always K highest-numbered architectural registers, where K is a design parameter, and the compiler ensures that the protected registers are best utilized for the metric being used. This scheme can be easily implemented in hardware in processors with no register renaming, but it could also be approximately implemented in superscalar processors. The decision logic is unnecessary, and finding the corresponding entry for redundant information becomes trivial. The protection mechanism can be any of ECC, parity, or duplication. While the hardware complexity is reduced, the responsibility of best utilizing the K entries is solely upon the compiler, especially the register allocator.

Register allocation is the process of allocating registers to program variables, with the goal of minimizing memory spills. Modifying the performance-optimized register allocation for partially protected RF can result in an increase in runtime and consequently in vulnerability as well. Therefore our approach is to swap register assignments in an already optimized binary, and thus performance-neutral. Also, this post-compilation register reassignment approach is simpler to implement than full register re-allocation. Not requiring source code, it can be readily applied to library functions which may be important to maximize the overall efficiency.

B. Energy Model

The energy overhead of protecting a register using redundancy-based schemes (e.g., ECC, parity, and duplication) can be modeled as the number of reads and writes to the register if we consider dynamic power only.[1] It is because the additional actions due to protection are only (1) generating redundant information on every write and (2) checking the correctness of the redundant information on every read. The reads and writes may be weighted differently depending on the energy model of the particular protection mechanism. Thus the number of accesses, with possibly different weighting factors, can represent the energy overhead of protected registers.

C. Compile-Time Optimization Methods

We present two compile-time optimization methods. One is Application-level Register Swapping (ARS), which is to change register assignments at the program level. The register assignment change is captured by a Register Reassignment Vector (RRV), which is unique for the entire program. The other is Function-level Register Swapping (FRS), which can change register assignment differently in each function, captured by Register Reassignment Table (RRT), or one RRV for each function. FRS cannot be done homogeneously for all registers due to the compiler's calling convention, which classifies the architectural registers into groups. Among the groups the most interesting ones are the caller-saved registers (also called t-registers) and the callee-saved registers (also called s-registers). Thus there are two version of FRS, which are referred to as FRS/t and FRS/s. On the other hand, ARS can be done uniformly for all registers (assuming statically linked binaries), since calling conventions do not apply when changes are made consistently in the entire program. The only exceptions are

[1] Since our compiler technique do not affect the performance, leakage power remains the same.

the registers that are reserved for system calls and those that are architecturally distinguished, or that are treated differently by the instruction set. An example of architecturally distinguished register is the link register or r31 in the MIPS architecture [11] since jal instruction implicitly writes to r31.

IV. Finding Optimal Solutions

The flexibility of our compiler-managed register file protection approach allows for the use of different optimization goals at compile-time. We consider two optimization goals: vulnerability and energy efficiency. Optimizing for vulnerability may be thought of as a direct translation of previous hardware partial protection techniques into software domain. Optimizing for energy efficiency goes a step further and is expected to achieve higher energy reduction. Considering the two optimization goals we have six optimization problems: V-ARS, V-FRS/t, and V-FRS/s for vulnerability, and E-ARS, E-FRS/t, and E-FRS/s for energy efficiency. We formulate the problems and present efficient algorithms for four of them.

A. Problems

Since the ARS problems are special cases of the FRS problems (although the scope of ARS is bigger than that of FRS), we consider formulation of FRS problems. Let R be the number of registers (either caller-saved or callee-saved) and N the number of functions. Then a solution to an FRS problem is a RRT $T = \{\vec{\rho}^f \mid f = 1, 2, \cdots, N\}$, each of which is a RRV, or a permutation of integers $1, 2, \cdots, R$. A RRV $\vec{\rho}^f$ of function f means that all the occurrences of register ρ_r^f in function f should be replaced by register r in the transformed program. Stated another way, all the variables originally assigned to register ρ_r^f are now assigned to register r after transformation.

Given a register access trace P and a RRT T, there are two known procedures $\text{CompV}(P, T)$ and $\text{CompA}(P, T)$ that compute the vulnerability vector $\vec{V}$ and the access count vector $\vec{A}$ of all the registers, where each element of a vector corresponds to each register. For K number of protected registers the vulnerability in the unprotected registers is $V_1 + V_2 + \cdots + V_{R-K}$ or $\sum_{r=1}^{R-K} V_r$. Suppose that we want to minimize the vulnerability in the unprotected registers for all values of K. Assuming uniform distribution on K, where K varies from 0 to R, the expected vulnerability is proportional to $RV_1 + (R-1)V_2 + \cdots + V_R$. Thus the cost function C^V for vulnerability optimization is

$$\text{Minimize } C^V = \sum_{r=1}^{R} (R + 1 - r)V_r \,. \tag{1}$$

For energy efficiency optimization, since there are two metrics that need to be minimized, we can consider constraint optimization formulation or combining the two metrics into one cost function. The former yields an NP-complete problem (not shown here) while the other can be solved very efficiently, which is presented here. The energy overhead is modeled as the number of accesses to protected registers, which is $A_{R-K+1} + \cdots + A_R$ or $\sum_{r=1}^{K} A_{R-K+r}$. Again assuming uniform distribution on K, the expected access count is proportional to $A_1 + 2A_2 + \cdots + RA_R$ or $\sum_{r=1}^{R} rA_r$. Introducing β representing the weighting factor between vulnerability and the access count, the cost function C^E for energy efficiency is

$$\text{Minimize } C^E = \sum_{r=1}^{R} (R + 1 - r)V_r + \beta r A_r \,. \tag{2}$$

Thus the vulnerability (or energy efficiency) optimization problems are, given a trace P, find the RRT T that minimizes C^V (or C^E), where V_i and A_i are defined by known procedures $\text{CompV}(P, T)$ and $\text{CompA}(P, T)$, respectively. We note that in these formulations optimizing for vulnerability only is a special case of optimizing for energy efficiency with $\beta = 0$.

B. Difference between T-register and S-register

FRS/t problems are simpler than FRS/s problems since the live range of a t-register is confined to one function.[2] 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. 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. Thus, for instance, V-FRS/t can be solved by simply sorting the registers by vulnerability, which coincides with intuition.

FRS/s problems are much more complicated. In any function the first access to an s-register 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. (Unlike a t-register, an s-register can be vulnerable even if there is no access to it in a function, which is the case if the first access *after* the function is a read.) 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.

ARS problems are special cases of FRS/t problems with only one function. Intuitively, V-FRS/t problem, and consequently V-ARS problem also, can be easily solved by sorting the registers by vulnerability, which will become clear in the next subsection.

C. E-ARS and E-FRS/t Solutions

Unlike vulnerability optimization, optimizing for energy efficiency is not very intuitive. One intuition says that sorting the registers by vulnerability-to-access-count ratio so that high-numbered registers will have higher vulnerability and lower access count than low-numbered registers, will give overall good energy efficiency. Although very likely, it is not always the case. (Counter examples are not shown due to the space constraint.) Without an efficient algorithm even the E-ARS problem can be difficult to solve optimally. A naïve approach evaluating all the $R!$ register orderings, for instance, is virtually infeasible because $R! > 10^{29}$ for $R = 28$, which is the number of ARS-swappable registers in the MIPS architecture.

[2] In the context of FRS, we regard any register whose live range is limited to within a function as t-register.

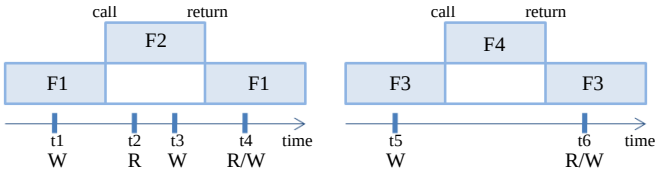


Fig. 1. S-register access pattern in relation to function calls.

Fortunately, there is an efficient algorithm for E-FRS/t and consequently E-ARS problems. For t-registers the register file vulnerability in a function is fully determined by the function itself, greatly simplifying CompV and CompA procedures. Let v_r^i and a_r^i be the vulnerability and access count of register r in function i before register reassignment, which can be easily found from the register access trace P. For a RRT $T = \{\vec{\rho}^{\,i} \mid i = 1, 2, \cdots, N\}$, V_r and A_r can be computed as:

$$V_r = \sum_{i=1}^{N} v^i_{\rho^i_r}, \quad A_r = \sum_{i=1}^{N} a^i_{\rho^i_r} \tag{3}$$

Substituting (3) in (2) and changing the order of summations shows that the cost function C^E can be minimized by minimizing C^E for each function, or C' defined as:

$$C'(\vec{\rho}) \quad = \quad \sum_{r=1}^{R} (R+1-r) v_{\rho_r} + \beta r a_{\rho_r} \tag{4}$$

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 (4).*

Proof Let x be the register number with which the register i is replaced in a register ordering $\vec{\rho}$. That is, $\rho_x = i$. Likewise let y be a number such that $\rho_y = j$. Then it is given that $x > y$. Let σ be equal to ρ except that $\sigma_x = j$ and $\sigma_y = i$. Since $C'(\vec{\sigma})$ is always less than $C'(\vec{\rho})$ as shown below, swapping i and j always gives a lower cost.

$$\begin{aligned} & C'(\vec{\rho}) - C'(\vec{\sigma}) \\ = \; & (R+1-x)v_{\rho_x} + \beta x a_{\rho_x} + (R+1-y)v_{\rho_y} + \beta y a_{\rho_y} \\ & -(R+1-x)v_{\sigma_x} - \beta x a_{\sigma_x} - (R+1-y)v_{\sigma_y} - \beta y a_{\sigma_y} \\ = \; & (x-y)(v_j - \beta a_j - v_i + \beta a_i) > 0 \end{aligned}$$

Thus sorting the registers by $(v_i - \beta a_i)$ in each function gives the optimum solution for E-FRS/t (and E-ARS). As for the weighting factor β, any positive value can be used and it is not even required for $(v_i - \beta a_i)$ to be positive, but the most sensible value would be the overall vulnerability-to-access-count ratio. The complexity of this algorithm is $O(NR\log(R))$ for N functions.

D. FRS/s Solution

To understand the complexity of the FRS/s problems let us consider an s-register access pattern illustrated in Fig. 1. Accesses are represented by small thick vertical bars annotated with time and type. 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. First example, illustrated by interval (t_1, t_2), includes a function call only, and is always vulnerable since t_2 must be a read. However second example, illustrated by interval (t_3, t_4), includes a function return, and may or may not be vulnerable since t_4 can be either read or write. Therefore the vulnerability of F2 (callee), and hence the optimal RRV for F2, depends on the RRV for F1 (caller).

Now suppose that not all s-registers are used in F2. Then on certain register reassignments of F2 the s-register in question can have no access, as illustrated on the right side of Fig. 1. In this case interval (t_5, t_6) may or may not be vulnerable, depending on the type of access at t_6. Thus not only the callee (F4) vulnerability may depend on the caller (F3), but 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 (on every invocation). 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 somehow breaking the inter-dependence the best we can get is $O(R^N)$, which is still exponential.

One simple heuristic exploits the fact that s-registers are more likely to be first read after control is transferred to another function (because every first access in a callee function is a read). Our profiling on GCC from SPEC 2000 indicates that this is true for more than 90%. Using this insight we can fix the s-register vulnerability of each function, which breaks the inter-dependence, and the problem becomes identical to FRS/t.

Table II summarizes the six problems we discussed. Vulnerability optimization is mostly obvious whereas optimizing for energy efficient is less intuitive. In both cases FRS/s problem presents a greater challenge, for which we proposed a simple algorithm.

TABLE II
COMPARISON OF PROBLEMS

Method (Scope)	For Vulnerability	For Energy Efficiency
ARS (Almost all regs)	Very simple	Efficient algorithm exists
FRS/t(T-registers only)	Very simple	Efficient algorithm exists
FRS/s(S-registers only)	Finding optimum is very complex; heuristic	

V. EXPERIMENTS

A. Evaluation Methodology

To evaluate the effectiveness of the proposed methods we perform simulations using embedded benchmark applications [12]. We use the SimpleScalar performance simulator [13], configured for in-order execution. Applications are compiled with GCC 2.7.2.3 using the benchmark-specified optimization level. The target architecture has the MIPS instruction set [11], in which there

are 11 t-registers (r1, r8 through r15, and r24 and r25) and 9 s-registers (r16 through r23 and r30).[3] In applying E-ARS and E-FRS algorithms in our experiments we set the weighting factor β to the overall vulnerability-to-access-count ratio of the original program.

One of our claims is that our compiler-managed register file protection is more energy efficient than hardware only schemes. Since the energy difference is expected to be high, it seems better to consider the ideal case for the entire hardware approach than comparing with every hardware scheme. The ideal case in the context of hardware schemes is *maximum vulnerability reduction*, which is however a hard problem by itself. Which variables to protect to maximize vulnerability can be decided optimally from a register access trace, but it is an NP-hard problem as it closely resembles the register allocation problem [14], of which the best known practical solution is the Chaitin-Briggs algorithm [14], whose complexity is $O(n^2)$ for n variables. Obviously running this algorithm on our register access trace (of 500K $\sim$ 100M dynamic variables) is almost impossible.

From the energy efficiency perspective the ideal case is to protect only those variables with the highest lifetime-to-access-count ratio. But at the same time we should try to maximally utilize the protected registers. Thus one reasonable upper bound on the energy efficiency or V/E can be found as follows. (i) Sort the dynamic variables in the decreasing order of $(v_i - \beta a_i)$, where v_i is the lifetime (representing vulnerability), a_i is the access count, and β is the overall vulnerability-to-access-count ratio. (ii) Take the sorted variables one by one adding their v_i to V and a_i to A until the selected variables cannot be mapped to K registers due to the interference (i.e., variables with overlapping live ranges cannot be mapped to the same register). Note that this can be done efficiently using the left-edge algorithm [15] and by fixing the allocation of already selected variables.[4] (iii) Then the resulting V and A will have high V and low A, approximating an ideal case from the energy efficiency perspective (denoted by *HWideal*).

B. Comparison between Compile-Time Optimizations

We first compare the four optimization methods (V- and E- versions of ARS and FRS) for their optimization capability. V-ARS represents the simplest form of compiler-managed RF protection while E-FRS is the most optimized for energy efficiency. For this comparison we limit ARS to s-registers (t-registers). The K protected registers are also assumed to be s-registers (t-registers). Figures 2 (a) and (b) compare the four methods in terms of vulnerability and energy overhead of protecting s-registers (for jpeg). The V-K graph visualizes the decreasing RF vulnerability as we increase K. While we observe some differences between optimization methods (with V-FRS achieving the greatest vulnerability reduction), the differences are rather limited. More interesting one is the V-E graph, which illustrates the tradeoff between vulnerability and energy overhead[5] for different K. Here the nearer to the lower left corner, the more energy efficient. Contrary to vulnerability optimizations (V-ARS, V-FRS), which are scattered along the diagonal line, energy efficiency optimizations form smooth curves. Quantitatively, when $K = 6$, E-ARS can achieve 24% energy overhead reduction over V-ARS while E-FRS gives additional 28% over E-ARS without affecting vulnerability. We observe similar trends across a number of applications. To summarize the results we again use the C^E cost metric defined in (2). The cost metric is computed using the simulation statistics after applying register reassignments. The result is summarized in Table III normalized to the V-ARS case. Though not shown in the table, from the efficiency perspective V-FRS is worse than V-ARS in several applications. On average, when used for s- or t-registers, E-ARS can reduce the cost metric by 13$\sim$20% over the simple V-ARS technique while E-FRS can give additional 5% reduction on average. The actual energy efficiency improvement however can be greater depending on K, especially in the medium range of K (see Fig. 2 (b)).

TABLE III
COMPARISON OF DIFFERENT OPTIMIZATION METHODS AND SCHEMES: IN TERMS OF THE COST VALUE DEFINED BY (2) WITH NORMALIZATION*

		S-REGISTERS		T-REGISTERS		
App.	#func**	E-ARS	E-FRS	E-ARS	E-FRS	SW-E
jpeg	11	0.84	0.77	0.82	0.74	0.81
tiff	2	0.87	0.81	0.40	0.39	0.71
typeset	13	0.88	0.85	0.91	0.88	0.78
patricia	21	0.87	0.82	0.86	0.76	0.77
ispell	16	0.91	0.89	0.91	0.90	0.78
rsynth	18	0.89	0.88	0.80	0.72	0.74
stringsearch	11	0.87	0.84	0.97	0.93	0.70
pgp	16	0.93	0.82	0.86	0.75	0.76
fft	18	0.94	0.89	0.87	0.72	0.77
gsm	7	0.73	0.67	0.84	0.84	0.84
Geo. Mean		0.87	0.82	0.80	0.75	0.76

*All the cost values are normalized to that of V-ARS (in the case of s-registers and t-registers) and SW-V (in the case of SW-E).
**The second column shows the number of functions each accounting for more than 1% of the runtime.

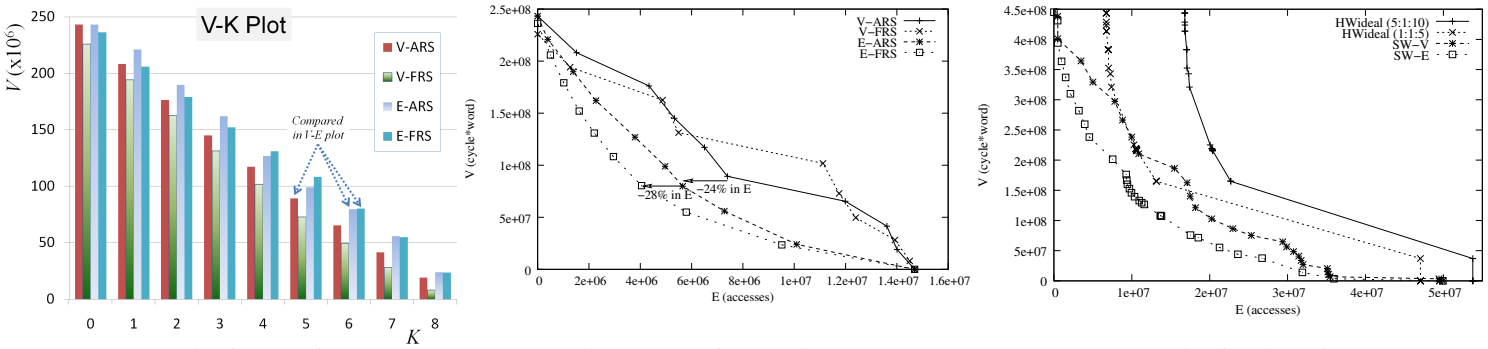
C. Compiler-Managed vs. Ideal Hardware-Managed

While FRS is superior to ARS in terms of optimization capability, they are complementary rather than competing since ARS can (and should) be used together with FRS. So the most interesting use cases would be: *SW-E* (applying all of E-FRS/t, E-FRS/s, and E-ARS) and *SW-V* (applying V-ARS only). We compare those cases against HWideal, the ideal hardware case. For the compiler-management schemes we assume that the architecturally distinguished registers are protected the first, which means r31 is always protected. On the other hand, those reserved for system calls are assumed to be protected the last, which is fair as they cannot be used in our experimental setup. For the energy ratio between the three operations, we use 5:1:10 and 1:1:5 (decision, entry, and re-

[3] r30 can be used as the frame pointer but our GNU C compiler is configured to use it as an s-register.

[4] This is to avoid running the left-edge algorithm anew whenever a new variable is added. Fixing the allocation can give inferior solutions while being faster (the complexity is $O(n \log(n))$); that is, less variables may be mapped to K registers than is possible with more time-consuming algorithms, which is why this method gives a higher V/A ratio.

[5] Here the energy overhead (E in the graph) is the number of accesses to protected registers.



(a) V-K plot for s-registers (b) V-E plot for s-registers (c) V-E plot for all registers

Fig. 2. Vulnerability and energy tradeoff of protecting registers (for the jpeg case).

dundant information, respectively), which can vary depending on the exact protection mechanism used.

Figure 2 (c) compares the tradeoff generated by different schemes for jpeg example. The x-axis (E in the graph) represents the energy overhead due to RF protection, considering the decision and entry related energy. First, even for the whole register file, SW-E can achieve much higher energy efficiency compared to the simple SW-V. Second, ideal hardware cases have a very steep slope for low values of K and thus high energy efficiency in that region. However, it is offset by constant energy overhead due to runtime decision. Third, only if the energy ratio is 1:1:5 (which is relatively low decision overhead) the ideal hardware case can overlap with the SW-V curve, but not the SW-E curve, which is the most energy efficient. We repeated similar experiments for other applications, which is summarized in Table III (last column). For the entire register file, our energy efficient optimization scheme can consistently improve the energy efficiency over the simple SW-V, by 24% on average. Quantitative comparison with the ideal hardware case is not reported as it would be highly sensitive to the energy ratio used.

VI. Conclusion

We presented a compiler approach to highly energy efficient register file protection for embedded systems. While previous approaches concentrated only on maximizing protections mainly through architectural changes, we show that pure hardware based solutions can suffer from high energy overhead, which can be a critical concern in embedded systems. Our approach is to explicitly maximize the energy efficiency through compilers, which can easily exploit the fundamental tradeoff between vulnerability and energy. Our proposed post-compilation 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 compile-time optimization problems and present efficient algorithms for some of them. Our experiments using embedded application benchmarks demonstrate that even the simplest of such compiler-management schemes (eg., V-ARS) can be more energy efficient than hardware schemes, and that explicit optimization (eg., E-FRS) can further increase the energy efficiency, by 24% on average as measured by our cost metric on register file vulnerability and energy overhead.

References

[1] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K. S. Kim, "Robust system design with built-in soft-error resilience," *IEEE Computer*, vol. 38, pp. 43–52, 2005.

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

[3] J. A. Blome, S. Gupta, S. Feng, and S. Mahlke, "Cost-efficient soft error protection for embedded microprocessors," in *CASES '06*, 2006, pp. 421–431.

[4] T. J. Slegel *et al.*, "IBM's S/390 G5 microprocessor design," *IEEE Micro*, vol. 19, pp. 12–23, 1999.

[5] R. Phelan, "Addressing soft errors in ARM core-based SoC," 2003, ARM white paper.

[6] J. Yan and W. Zhang, "Compiler-guided register reliability improvement against soft errors," in *EMSOFT '05*, 2005, pp. 203–209.

[7] P. Montesinos, W. Liu, and J. Torrellas, "Using register lifetime predictions to protect register files against soft errors," in *DSN '07*, 2007, pp. 286–296.

[8] G. Memik, M. Chowdhury, A. Mallik, and Y. Ismail, "Engineering overclocking: reliability-performance trade-offs for high-performance register files," *DSN '05*, pp. 770–779, 2005.

[9] M. Kandala, W. Zhang, and L. Yang, "An area-efficient approach to improving register file reliability against transient errors," in *Int'l Symp. on Embedded Computing*, 2007.

[10] 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. International Symposium on Microarchitecture*, Dec 2003.

[11] D. Patterson and J. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers, 2004.

[12] 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 *Int'l Workshop on Workload Characterization*, 2001.

[13] T. Austin, "SimpleScalar LLC."

[14] P. Briggs, K. D. Cooper, and L. Torczon, "Improvements to graph coloring register allocation," *ACM Trans. Program. Lang. Syst.*, vol. 16, pp. 428–455, 1994.

[15] R. E. Sant'Anna, M. E. de Lima, and P. R. M. Maciel, "A left-edge algorithm approach for scheduling and allocation of hardware contexts in dynamically reconfigurable architectures," in *Int'l Symp. on Field programmable gate arrays*, 2004, pp. 259–259.