A Joint SW/HW Approach for Reducing Register File Vulnerability

HAMED TABKHI and GUNAR SCHIRNER, Northeastern University

The Register File (RF) is a particularly vulnerable component within processor core and at the same time a hotspot with high power density. To reduce RF vulnerability, conventional HW-only approaches such as Error Correction Codes (ECCs) or modular redundancies are not suitable due to their significant power overhead. Conversely, SW-only approaches either have limited improvement on RF reliability or require considerable performance overhead. As a result, new approaches are needed that reduce RF vulnerability with minimal power and performance overhead.

This article introduces Application-guided Reliability-enhanced Register file Architecture (ARRA), a novel approach to reduce RF vulnerability of embedded processors. Taking advantage of uneven register utilization, ARRA mirrors, guided by a SW instrumentation, frequently used active registers into passive registers. ARRA is particularly suitable for control applications, as they have a high reliability demand with fairly low (uneven) RF utilization. ARRA is a cross-layer joint HW/SW approach based on an ARRA-extended RF microarchitecture, an ISA extension, as well as static binary analysis and instrumentation. We evaluate ARRA benefits using an ARRA-enhanced Blackfin processor executing a set of DSPBench and MiBench benchmarks. We quantify the benefits using RF Vulnerability Factor (RFVF) and Mean Work To Failure (MWTF). ARRA significantly reduces RFVF from 35% to 6.9% in cost of 0.5% performance lost for control applications. With ARRA's register mirroring, it can also correct Multiple Bit Upsets (MBUs) errors, achieving an 8x increase in MWTF. Compared to a partially ECC-protected RF approach, ARRA demonstrates higher efficiency by achieving comparable vulnerability reduction at much lower power consumption.

Categories and Subject Descriptors: B.8.1 [Performance and Reliability]: Reliability, Testing, and Fault-Tolerance

General Terms: Reliability, Performance, Experimentation

Additional Key Words and Phrases: Register File (RF), vulnerability factor, register mirroring, binary instrumentation

ACM Reference Format:

Hamed Tabkhi and Gunar Schirner. 2015. A joint SW/HW approach for reducing register file vulnerability. *ACM Trans. Architect. Code Optim.* 12, 2, Article 9 (May 2015), 28 pages.
DOI: <http://dx.doi.org/10.1145/2733378>

1. INTRODUCTION

The shrinking feature size introduces new challenges to designing reliable embedded processors. Among all reliability threats, soft errors are the main contributor. Soft errors, or transient errors, are provoked by high-energy particle strikes causing incident charges in both memory and computational components [Azambuja et al. 2011a; Benso et al. 2000]. In deep submicron technologies, many processor execution errors origin from soft errors [Shivakumar et al. 2002; Azambuja et al. 2011a]. The challenge of soft

This article is a journal extension over previously published work [Tabkhi and Schirner 2012a, 2012b].

Authors' addresses: H. Tabkhi and G. Schirner, Department of Electrical and Computer Engineering at 328 Dana Research Center, Northeastern University, Boston (MA), USA 02115; emails: {tabkhi, schirner}@ece.neu.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2015 ACM 1544-3566/2015/05-ART9 \$15.00

DOI: <http://dx.doi.org/10.1145/2733378>

errors is more pivotal in embedded processors, as they are set to work in markets with higher-reliability demands such as automotive, biomedical, or cyberphysical systems.

Power and energy constraints are considered as the main challenge for widely deployment of error-hardened approaches in embedded processors. Error-hardening approaches often impose significant energy and power overheads by employing varying types of redundancies including time redundancy (e.g., replicating execution), information redundancy (e.g., error correction codes) and modular redundancy (physical copy of the same instance). Conversely, embedded processors are mainly employed in mobile devices, thus their lifetimes are bounded by batteries. This introduces a tradeoff between reliability and power consumption, which is a key exploration aspect in designing of many embedded processors. In order to break free this tradeoff, innovative solutions that can deliver major improvements in the reliability at minimal power cost are in high demand.

Among components within a processor core, the Register File (RF) is a key contributor to processor's performance. Simultaneously, RF has a pronounced impact on processor's reliability [Lee and Shrivastava 2010; Amrouch and Henkel 2011; Fazeli et al. 2010; Hu et al. 2009b]. Being an integral part of the processor with very frequent accesses, soft errors originating in the RF can rapidly propagate to the data and control path, leading to potential processor crash or silent data corruption [Memik et al. 2005]. As an example, on an ARM core (ARM926EJ embedded microprocessor), more than 50% of errors appearing at the software interface originate from errors in the RF [Blom et al. 2006]. Similar measures have been reported for Alpha cores, where 40% of failures in datapath stems from errors occurred in RF [Wang et al. 2004]. The very frequent RF accesses and being the origin of many exposed errors necessitate reliability enhancement in RFs.

RFs are also the primary energy and power consumers within the processor cores. Power breakdowns of embedded processors reported in Ayala et al. [2003] and Guan and Fei [2010] show that RFs consume 15% to 36% of the cores' overall power and up to 42% of cores' datapath [Gonzales 1999]. As an example, 20% of core power (including data and instruction caches) in Blackfin processor is attributed to the RF itself [Oliver et al. 2004]. Furthermore, RF is a hotspot with a high power density, leading to a higher leakage power [Zhou et al. 2008], which in turn intensifies the associated RF reliability issues. Overall, the high energy and power consumption of RFs make the utilization of conventional error-hardening approaches highly challenging.

Previously proposed redundancy-based approaches are power costly and not suitable for reliability enhancement of RFs. While Error Correction Codes (ECCs) or Triple Modular Redundancy (TMR) have been successfully applied to protect the memory hierarchy, their applicability to RFs is very limited due to their considerable power and energy overheads [Lee and Shrivastava 2009, 2010]. As an example, the energy for ECC checking to correct a single bit-flip alone is three times higher than the register accesses itself [Yan and Zhang 2005]. In order to address the tight power budget of embedded processors, partially protected RF approaches have been proposed to only protect the most vulnerable registers [Fazeli et al. 2010; Blom et al. 2006; Lee and Shrivastava 2010]. However, even a partially protected RF still imposes a considerable static and dynamic power overhead. As a result, innovative RF-specific error-hardening approaches are necessary to enhance the RF reliability with a minimal power overhead.

Modern embedded processors are often designed with large RFs, composed out of heterogeneous register banks with special-purpose registers [Analog Devices Inc. (ADI) 2008; Qualcomm Inc. 2011; Texas Instrument (TI) 2010]. Large heterogeneous RFs aid to efficiently run a variety of applications including control, DSP, and media applications [Katz et al. 2009a]. Embedded cores such as TI C64x+ [Texas Instrument (TI) 2010], ADI Blackfin [Analog Devices Inc. (ADI) 2008] and Qualcomm Hexagon [Qualcomm Inc. 2011] have large RFs. In the TI series, for example, the RF size has

Fig. 1. ARRA overview joining SW and HW for RF reliability improvement.

doubled from thirty-two 32-bit registers in C62x/C67x series to 64 registers in C64x+. Although, large heterogeneous RFs considerably improve the efficiency of embedded processors, some registers are not utilized during portions of execution.

The RF utilization varies significantly among registers both between and within applications [Memik et al. 2005; Collins et al. 2006]. Based on our observation, this is more pronounced in heterogeneous RFs as individual register banks have dedicated functionality not fully utilized during the entire execution. The unbalanced register utilization offers a potential for RF reliability improvement by mirroring the content of active registers to passive registers not contributing to program execution. However, the processor core by itself does not have sufficient information to distinguish a register in use (will be read at some time) from an unused register (will be overwritten with a new value). Therefore, an application-guided mechanism is needed to notify the core about register utilization. To enable wide usage, an automatic instrumentation of SW binaries to control runtime register mirroring is highly desirable.

In this article, we propose an Application-guided Reliability-enhanced Register file Architecture (ARRA) to mitigate soft errors in RFs. ARRA relies on the fact that control applications have high reliability demands while their overall RF utilization is fairly low. This creates an opportunity for mirroring the content of the vulnerable (active) registers to invulnerable (passive) registers, improving overall RF vulnerability. To realize register mirroring, ARRA employs a cross-layer HW/SW approach including RF microarchitecture enhancement, Instruction-Set-Architecture (ISA) extension, as well as static application binary analysis and instrumentation.

Figure 1 outlines the flow of our proposed ARRA approach. The ARRA is a cross-layer joint HW/SW approach. At the SW level, ARRA employs a static application binary analysis to identify unused (passive) registers for mirroring the content of active registers at the function-level granularity. The binary analysis at the function-level granularity ignores the fine-grained register activity within the function bodies and instead limits the register tracking across the functions. At the HW level, ARRA extends RF microarchitecture to support a programmable register mirroring. It introduces a MAP controller and an error detection/correction unit. The MAP controller includes a configuration register, MAP register, to control register mirroring at runtime and governs register mirroring in HW. ARRA extends the ISA with a MAP instruction to control the MAP register from SW. The application analysis and instrumentation of ARRA instruments all functions with MAP instruction according to their register mirroring potential.

We evaluate the efficiency of ARRA based on a Blackfin core. The ARRA-enhanced Blackfin core reduces RF Vulnerability Factor (RFVF) from 35% to 6.9%, on average, for control applications with performance overhead of 0.3%. With ARRA's register mirroring, it can also correct Multiple Bit Upsets (MBUs) errors result in 5x to 7.5x

increase in Mean Work To failure (MWTF). We also compare the ARRA reliability and power efficiency with the closest related approach (partially ECC-protected RF [Lee and Shrivastava 2011]). An ARRA-protected RF consumes 70% less power than a partially ECC-protected RF on average, while offering a 2.5x higher MWTF.

This article is an extension over previously published work [Tabkhi and Schirner 2012a, 2012b] which presented limited aspects. Tabkhi and Schirner [2012a] illustrated initial binary analysis for the purpose of register mirroring, and Tabkhi and Schirner [2012b] illustrated some ideas about hardware extensions for realizing runtime register mirroring. In this article, we provide a more comprehensive description of the ARRA approach, covering both application and microarchitecture aspects with significantly added details about previous work, approach, and results. Furthermore, this article extends over the initial ARRA publications in four aspects:

- (1) It explores the design decision tradeoffs including granularity for controlling register mirroring, as well as flexibility of register mirroring at hardware (Section 5.1).
- (2) It enhances the binary instrumentation to reduce the performance overhead of register mirroring for recursive function calls (Section 5.5.1).
- (3) It provides a more extensive reliability evaluation by utilizing Mean Work To Failure (MWTF) metric to simultaneously consider the effect of MBUs on error correction and performance overhead (Section 6.5).
- (4) It investigates on activity-aware register mirroring to improve the efficiency of direct-map register mirroring (Section 7.1).

The remainder of this article is organized as follows. Section 2 discusses previous work and Section 3 briefly reviews relevant background. Section 4 highlights the potential of register mirroring motivating ARRA. Following that, Section 5.1 discusses design options guiding the ARRA design. Section 5 then comprehensively describes ARRA realization. Then, Section 6 evaluates ARRA benefits. Finally, Section 8 concludes this article.

2. PREVIOUS WORK

Improving RFs reliability is a challenging problem due to the considerable energy overhead of redundancy-based techniques and the high power density of RFs. As an example, while ECC is widely used in memory subsystems and cache hierarchy, it is often impossible to design a fully ECC-protected RF due to the high power consumption associated with ECC. Therefore, a sizable body of research deals with power-efficient approaches for hardening register file against soft errors. In this section, we roughly divide RF hardening approaches into three categories: HW-only, SW-only, and hybrid HW-SW approaches. Furthermore, we briefly distinguish our proposed ARRA from the existing hybrid approaches.

2.1. HW-Only

HW-only approaches only focus on microarchitecture techniques to improve RF reliability independent from the application (SW) executing on top. The research in this area can be roughly divided into two main categories with respect to core's microarchitecture: out-of-order and in-order (embedded) processor. For out-of-order cores, the approaches focus on selectively protecting registers in the large physical RF used for register renaming and out-of-order execution [Memik et al. 2005; Montesinos et al. 2007]. Montesinos et al. [2007] demonstrate that a considerable portion of physical registers only keep speculative values and do not contribute to actual execution. Therefore, Montesinos et al. [2007] selectively applies ECC to only those physical registers with a high probability of containing useful data. Alternatively, Memik et al. [2005] duplicates the content of live registers to those microarchitectural unused registers not

contributing to speculative execution. These approaches are not applicable to embedded processors as they typically do not support register renaming.

For embedded (in-order) cores, some prior work focuses on pure hardware solutions [Blom et al. 2006; Fazeli et al. 2010; Amrouch and Henkel 2011; Hu et al. 2009b]. Blom et al. [2006] and Fazeli et al. [2010] proposed register caching mechanisms to only protect the cached registers either by SEU-tolerant latches [Fazeli et al. 2010] or Cyclic Redundancy Check (CRC) [Blom et al. 2006]. Other approaches protect narrow bits values either by storing ECC codes [Amrouch and Henkel 2011] or duplicating the content of lower-half word to higher-half [Hu et al. 2009b]. These approaches are restricted to limited data range and are constrained by high power overhead of ECC checking logic. Furthermore, the ECCs suggested in these approaches only focuses on a single bit-flip correction (as design bounded by power consumption). However, the studies show that transient bit-flips often lead to multiple bit errors that cannot be corrected by a single-bit correcting ECC [Yoshimoto et al. 2011].

2.2. SW-Only

Software-oriented approaches are alternatives for reliability enhancement without any special HW support. One example is the SoftWare Implemented Fault Tolerance (SWIFT) [Reis et al. 2005c] approach, which keeps a copy of critical variables in memory system. SWIFT compares primary and copy values upon read access at the software layer. SWIFT can achieve higher reliability, however, at the cost of a sizeable overhead to the performance. For the Register File (RF), in particular, SW-only approaches mainly focus on instruction rescheduling and selective duplication of crucial registers [Yan and Zhang 2005; Xu et al. 2011]. Yan and Zhang [2005] demonstrated that by rescheduling instructions alone, the RF vulnerability factor can be reduced up to 30%. Similarly, Benso et al. [2000] propose the reordering of variables at the source level to reduce the variable lifetimes, thus reducing their vulnerability to soft errors. One step further, Rehman et al. [2012] propose instruction rescheduling, considering both spatial and temporal vulnerability of variables to reduce the vulnerability of RF, as well as intermediate datapath registers. Rehman et al. [2012] assume that the compiler already knows the pipeline organization and microarchitecture registers.

Few analytical approaches also attempt to address the challenge of identifying registers with higher vulnerability [Lee and Shrivastava 2011; Xu et al. 2010]. Identifying vulnerable registers at earlier stages helps compilers or even HWs to achieve a more efficient RF protection. However, Lee and Shrivastava [2011] and Xu et al. [2010] only propose an analytical model and do not show an integration solution for protecting RF either at the SW or HW layers.

2.3. Hybrid HW-SW

Hybrid HW-SW approaches promise greater reliability improvements, combining application analysis with special HW support. As an example, Reis et al. [2005a] propose a small dedicated HW unit expanding over the original SWIFT approach [Reis et al. 2005c] and demonstrate a higher efficiency through a joint HW/SW approach. Instead of duplicating the critical register values into memory, Reis et al. [2005a] introduce a Checking Store Buffer (CSB) in hardware for fast and efficient variable duplication. Similarly, Azambuja et al. [2011b] propose a watchdog and specialized buffer guided by compiler to protect the critical variables.

Only few hybrid approaches aim for RF reliability improvement with low power overhead [Lee and Shrivastava 2009, 2010; Hu et al. 2009a]. Lee and Shrivastava [2010] suggest a partially protected RF where only the content of high vulnerable registers is protected by ECC. The approach in Lee and Shrivastava [2010] analyzes the power/reliability tradeoff to find the optimum number of registers can be ECC

protected. With a different angle, Lee and Shrivastava [2009] uses ECC-protected memory as backup for registers with a long vulnerable period. Hu et al. [2009a] selectively duplicates register values through duplicated instruction execution. Instead of the main RF, the duplicated values are written in a dedicated backup HW.

Similar to Lee and Shrivastava [2010]; Azambuja et al. [2011b], our approach is a hybrid HW-SW solution that combines application binary analysis with ISA and RF microarchitecture extensions. In contrast to previous approaches, ARRA minimizes the HW overhead by taking benefit of unutilized (passive) registers for mirroring vulnerable registers. The ARRA integrated HW-SW extensions improve reliability at significantly lower power consumption than HW-only approaches and outperforms SW-only approaches in improving reliability. To demonstrate the efficiency of ARRA, we also compare ARRA approach with partially ECC-protected RF (outlined in Lee and Shrivastava [2010]).

3. BACKGROUND

This section first illustrates advantages of heterogeneous RF in embedded processors and also overviews our case study—ADI Blackfin processor. Following that, it outlines the fault model targeted with ARRA. Lastly, this section reviews Register File Vulnerability Factor (RFVF) for quantifying the vulnerability of RF to soft errors.

3.1. Heterogeneous RF

Employing large RFs is a common approach to increase core performance. A large RF potentially reduces the number of accesses to the memory hierarchy. Different trends exist between out-of-order and in-order cores. In out-of-order designs, the physical RF can be extended for register renaming supporting speculative execution. This increases the physical registers, while the ISA registers remain constant. Conversely, one trend in embedded processors is composing a large logical RF out of heterogeneous registers with specialized functionality [Analog Devices Inc. (ADI) 2008; Qualcomm Inc. 2011; Texas Instruments (TI) 2010]. Heterogeneous registers are part of ISA and thus are exposed to the programmer/compiler. Embedded processors such as ADI Blackfin families [Analog Devices Inc. (ADI) 2008], TI C6xx series [Texas Instruments (TI) 2010], and Qualcomm Hexagon [Qualcomm Inc. 2011] are designed with a large RF. In the TI C6xx series, for example, the RF size has been doubled from thirty-two 32-bit registers in C62x/C67x series to 64 registers in C64x+ series with special-purpose registers.

3.1.1. Blackfin Core. As a case study for this article, we chose the Blackfin processor [Katz et al. 2009b] designed to efficiently support both control and signal processing applications. Blackfin has 38 x 32 bits ISA register file composed out of 8 data registers (#0-7), 8 pointer registers (#8-15), 16 circular addressing registers (#16-31), and 6 zero-hardware-loop registers (#32-37). Circular-addressing registers and zero-hardware-loop registers are special-purpose registers for fast and continuous data memory addressing or continuous execution of loop's instructions reducing pipeline stalls. Similar RF organization concepts have been applied to other embedded processors such as the TI C64x series [Texas Instruments (TI) 2010]. In general, special-purpose registers are mainly utilized in data streaming and DSP applications.

3.2. Error Model

ARRA targets Single Bit Upset (SBU) as well as Multiple Bit Upsets (MBUs) [Yoshimoto et al. 2011] as the soft error models. SBU refers to a single bit-flip in a register word. It is a well-known soft error model for memory and RF, and it is widely used in the research community [Yan and Zhang 2005; Fazeli et al. 2010]. MBUs refers to multiple bit-flips in a register word. MBUs frequently appear in adjacent bits when

Fig. 2. Vulnerable and nonvulnerable periods.

a high-energy radiation (alpha particle or neutron strikes) changes the value of 2 or more bits in a row [Fazeli et al. 2010; Georgakos et al. 2007]. Recent studies of modern SRAMs report that from 10% (as in Fazeli et al. [2010]) up to 55% (as in Georgakos et al. [2007]) of high-energy particles result in MBUs in 2 or more adjacent bits. ARRA aims to detect all possible adjacent bit-flips and correct up to three adjacent bit-flips.

3.3. Register File Vulnerability Factor (RFVF)

The Register File Vulnerability Factor (RFVF) [Lee and Shrivastava 2010] is a metric for measuring the RF vulnerability against soft errors. The basic idea of RFVF is quantifying vulnerability based on the status of registers (active/passive) in program execution. Figure 2 exemplifies active and passive periods over time progress. An active period starts from a write operation to a register and continues until the last read from that register. During an active period, the register contributes to correct the architectural state of the program, and any error on it can potentially affect the correct execution. Therefore, the register is vulnerable to errors. The active (vulnerable) period is also called an *Architecturally Correct Execution (ACE)* period. Adversely, the passive period is the duration between the last read and the next following write. In a passive period, the register does not contribute to the execution thus nonvulnerable to errors (un-ACE period).

$$RFVF = \frac{\sum_{0 < i < N} \sum_{0 < j < M_i} Active_{i,j}}{N \times \text{execution time}} \quad (1)$$

Equation (1) formulates RFVF calculation for a given RF, where N refers to the length of RF and M_j is the number of active periods for register index i . In the innermost summation, all active periods per individual register ($Active_{i,j}$) is added. The outermost summation accumulates the total active duration for each register. The summation result is divided over entire execution time multiplied by the length of the RF. RFVF has a direct relation to RF utilization; higher RF utilization results in higher RFVF and thus higher vulnerability to soft errors. Overall, when focusing only on the RF, RFVF is a very good indicator of registers vulnerability [Lee and Shrivastava 2010]. This is more pronounced in embedded and in-order cores due to much higher transparency between ISA and physical registers [Sridharan and Kaeli 2009].

Based on the RFVF metric, RF reliability can be enhanced by reducing the RFVF value. Two orthogonal approaches are (1) shorten the length of active periods (e.g., instruction rescheduling), thus reducing the duration of time registers are vulnerable to error, and (2) protect the registers during their vulnerable (active) periods, which can lead to much more effective vulnerability reduction.

4. REGISTER MIRRORING POTENTIALS

To motivate our proposed ARRA approach, this section investigates the register utilization as well as potentials for register mirroring. In order to evaluate the register utilization, we execute 10 optimized (compiled with GCC -O3) benchmarks on an instrumented Blackfin Instruction-Set-Simulator (ISS). The benchmarks include four DSP applications (from the DSPStone [Zivojnovic et al. 1994] suite) and six control applications (from the MiBench [Guthaus et al. 2001] suite). Figure 3(a) shows the

Fig. 3. Average RFVF in Blackfin core for control and DSP benchmarks.

measured register utilization of different register groups in Blackfin processors. We measured register utilization through runtime tracing of the RF access (read & write) for each register. Figure 3(a) distinguishes between control applications (blue) and DSP applications (red) for four groups of registers: data registers (#0–7), pointer registers (#8–15), and two groups of special-purpose registers (#16–23 and #24–31). The figure also shows the error bars indicating the range of utilization among all benchmarks.

Looking at Figure 3(a), we observe a considerable difference between register utilization of control and DSP applications. On average, the utilization of all registers is 28% and 57% for the control and DSP applications respectively. The difference is more pronounced for special-purpose registers (#24–31), with a much higher RFVF gap between control and DSP applications (33% for DSP to 7% for control applications). The low utilization of special-purpose registers stems from two factors: (1) compilers may not detect code patterns to benefit from heterogeneous registers in a large set of applications, and (2) many applications (mainly control applications) inherently do not require all register resources. As an example, control applications may not use circular addressing registers. Nonetheless, embedded programmers can hand optimize assembly code to improve performance—typically increasing register utilization. Having an approach to utilize unused registers allows the assembly programmers to balance reliability and performance. Low register utilization is in line with other results. As an example, Reis et al. [2005a] reported 19% average RFVF for the Intel Itanium processor with 128 * 64 bits integer RF.

The RF utilization of a nonprotected RF is equivalent to its RFVF (see Section 3.3). Therefore, the utilization percentages in Figure 3(a) also presents RFVF of Blackfin core for the benchmark under evaluation. Overall, special-purpose registers have low utilization (thus lower vulnerability) in control applications. At the same time, compared to DSP applications, control applications have more stringent reliability requirements. This provides a substantial potential for utilizing the passive (nonvulnerable) registers for protecting of active (vulnerable) registers by mirroring the content of active registers to passive registers (register mirroring). Register mirroring can prevent power costly redundancy-based approaches by using already available (free) resources. However, the efficiency of register mirroring is dependent on RF utilization.

To better understand the opportunities of register mirroring, we roughly estimate its benefit. Our estimation aims to show how many registers can be protected by mirroring versus how many remain unprotected. We use idealistic assumptions for the estimation: a nonvulnerable register can be used freely for mirroring at instruction granularity (i.e., register mirroring can change with each instruction, and mirroring is fully set associative). Figure 3(b) presents the estimated RFVF after optimistic register-mirroring, control (blue) and DSP (red), for the same group of registers. To report RFVF after register mirroring, we assume that the mirrored registers are not vulnerable to error. We also consider a higher mirroring priority for general-purpose registers,

data registers (#0-7), and pointer registers (#8-15), as they have higher reliability demands. Looking at Figure 3(b), after the ideal register mirroring, the RFVF for all control applications is zero. In other words, there are enough passive registers at each point of execution for mirroring the content of all active registers. In contrast, for DSP applications, the mirroring potential is still substantial, but not as large with an RFVF reduction from 57% to 35%.

Overall, the estimated results demonstrate a significant potential for reducing RF vulnerability. However, there are significant milestones from an ideal register mirroring to a realizable approach for embedded processors. The next section investigates and describes the detail of a realizable register mirroring approach.

5. APPROACH

The previous section showed that special-purpose registers have low utilization in a large set of applications. This happens particularly in control applications when there is not enough potential for taking the benefits of all special-purpose registers. To utilize this potential, vulnerable registers can be mirrored to unused (passive) registers. In this article, we propose ARRA to improve the RF reliability. ARRA automatically detects register passive periods in special-purpose registers and harnesses them for mirroring the content of vulnerable (active) registers. ARRA is a cross-layer joint HW/SW approach consisting of static binary analysis and instrumentation, an ISA extension, as well as an ARRA-extended RF microarchitecture.

Before discussing design details, the next section explores two essential design decisions for identifying the granularity and flexibility of register mirroring.

5.1. Design Tradeoffs

Our earlier estimations (Section 4) used very optimistic assumptions not suitable for a realistic implementation. To realize ARRA, two primary design decisions for runtime register mirroring have to be considered: (1) granularity at which applications control the register mirroring and (2) flexibility of register mirroring in hardware. Both decisions introduce tradeoffs between RF reliability improvement and design complexity. The next two parts discuss the tradeoffs individually.

5.1.1. Granularity of Register Mirroring. Theoretically, register mirroring can be controlled at application, function, loop, basic-block, or even instruction levels. There is a tradeoff between the reliability improvement and the configuration overhead. An application granularity would be too coarse, resulting in limited potential for register mirroring. On the other extreme, a fine-grained instruction-level or even basic-block granularity can utilize more of the potential for register mirroring (being able to react to register utilization more rapidly). However, each runtime change in RF mirroring introduces at least one additional instruction. In the result, instruction or basic-block granularity would add many instructions, resulting in potentially too much performance overhead. This introduces a tradeoff between the register mirroring and the configuration overhead. As the granularity of register mirroring moves from coarse application-level to fine instruction-level, more and smaller invulnerable periods can be utilized for register mirroring in cost of more frequent configuration changes.

To select the most suitable granularity, we first look into the number and length of passive periods. Figure 4 shows the cumulative contributions both in amount and length (both on the y-axis), over an increasing size of passive periods. The red line shows how passive periods of increasing size contribute to the total number of passive periods in the application. The blue line shows how these passive periods contribute to the total length of all passive periods combined. The results are gathered based on the profiling of the applications introduced in Section 3.3, running on Blackfin ISS.

Fig. 4. Contribution to unused time over increasing length of unused period.

Fig. 5. Instruction-level vs. function-level estimation of active registers for FFT benchmark.

Looking at the cumulative contribution for the number of passive periods (red line of Figure 4), the number of passive periods reduces rapidly as their length increases. About 99% of passive periods are less than 1,024 instructions in length. The remaining 1% are more than 1,024 instructions long. Looking at the cumulative contribution to the total length of all passive periods combined (blue line), only less than 40% appear in periods of less than 1,024 instructions in length. Conversely, passive periods longer than 1,024 instructions contribute to more than 60% of overall passive period length. Short passive periods (less than 1,024 instructions) mainly appear at instruction and basic-block granularity or in short loops with few number of iterations. In contrast, the long passive periods (more than 1,024 instructions) appear either in functions or loops with many iterations. The results indicate that it is inefficient to target small passive periods because they do not significantly contribute in terms of duration. Conversely, targeting long passive periods are more promising offering a significant potential for register mirroring while resulting in low overhead for mirroring reconfiguration.

Both loop and function-level seem to be suitable granularity for register mirroring; however, loop-level poses additional complexities. Detecting a loop with many-iterations may be data dependent. Additionally, register dependency analysis would be more complicated as it needs to take into account runtime conditions and input data. In contrast, functions provide well-known isolation over register accesses in the program code. Function-call conventions are predefined rules over register access within functions. They simplify tracking register dependencies across different function calls. Therefore, our ARRA approach selects the function-level granularity for register mirroring.

To estimate the effect due to the coarser granularity, Figure 5 compares the number of active (vulnerable) registers at instruction-level with our function-level estimation during the execution of 10,000 instructions for FFT benchmark running on the Blackfin ISS. On average, the coarser function-level estimates four more active registers compared to instruction-level. In fact, function-level estimation provides an upper bound

Fig. 6. The average RLVF for different mirroring flexibility.

number of active registers per function. Therefore, the number of active registers at instruction-level is always below of that in function-level estimation. The sharp edges in function-level estimations indicate function calls as estimated RF activity is updated based on the new executing function. Tracking registers utilization at the function-level granularity may leave some registers with no backup, meaning that the number of the active registers is higher than the number of passive registers in the same function. For compute-intense FFT benchmarks, there is enough passive registers per function to cover the most of active registers. The observed results further motivate us to consider function level as an efficient granularity for runtime register mirroring.

5.1.2. Flexibility of Register Mirroring. Another design decision is the flexibility of register mirroring between the passive and active registers. Register mirroring flexibility defines the number of register candidates for backup. The most limited, *direct map*, only allows one backup candidate register for mirroring. The most flexible, *fully associative map*, allows mirroring of an active register in any passive register. Medium flexibility is offered by two- and four-way associativity, allowing selection from two and four candidates, respectively.

To estimate the potentials of different mapping policies, we perform a high-level analysis over a set of control benchmarks from the MiBench suite. The experiment statically analyzes each programs' binary by tracing the register accesses of instructions at the function level with no actual execution. We assume that in every callable function of the program, each nonreferenced register is unused and thus can be applied for mirroring while every instruction only executes once. Figure 6 presenters the predicted RLVF when applying different mapping functions. We observe that the most significant drop in vulnerability from no mirroring ($>50\%$) to *direct map* (about 12%). The associative mappings only slightly improve the RLVF over direct mapping (down to about 6%). In contrast, the hardware complexity of associative mappings and their associated power overheads is much higher.

In order to achieve a considerable reduction in RLVF with minimum hardware complexity, ARRA chooses *direct map* with a one-to-one mapping between the active register and backup target. A design-time optimization opportunity exists for defining register pairs. As an example, higher vulnerable registers can be mirrored to the least utilized registers to maximize the efficiency of direct mapping mirroring. We will further explore this aspect in Section 7.1.

5.2. Microarchitecture Extension

This part discusses our RF microarchitecture extension for realizing runtime register-mirroring including error detection and correction logic.

In embedded processors, RFs are typically split into multiple Register Banks (RBs). Multiple RBs reduce the dynamic energy associated with RF access and furthermore streamline designing RFs with multiple read/write ports. Multibank RFs are common in heterogeneous RFs in which RF banks differ by purpose. Figure 7 exemplifies a

Fig. 7. The ARRA-enhanced RF microarchitecture extension.

simple heterogeneous RF with two RBs (RB0 and RB1) with two read and one write ports (2R/1W). RB0 holds general-purpose data, while RB1 is dedicated to special-purpose registers.

The basic approach is to utilize RB1's unused registers to protect RB0's registers through mirroring RB0's content into RB1. To realize register mirroring, the original RF microarchitecture is extended in four different aspects (see Figure 7): (a) a new logic component, called map controller, to support direct map register-mirroring; (b) error detection (register comparators) for error detection; (c) two parity generators to generate parity bits of odd and even bits, assisting error correction; (d) a Error Correction Unit (ECU) to identify the faulty copy of data.

The map controller implements direct-map functionality for register mirroring. The map controller contains a Map Register (MR) to enable/disable mirroring of individual registers during the runtime. Each bit in the in MR corresponds to one register. To enable mirroring for a register in RB0, its corresponding bit in MR needs to be set to one (see also the example in Section 5.3 about ISA extension). The map controller traces the RB0 read/write address lines to capture the register accesses. During each write access to RB0, the map controller checks whether the backup register in RB1 is free for mirroring by checking the corresponding bit in MR. If the bit is set, the same data will be written to both RB0 and RB1. Upon a read access from a mirrored register, the map controller will send the read request to both RBs to get both original and backup values.

For error detection, a register comparator is added to the output stage of RBs, checking the consistency between data read from RB0 and RB1. To reduce the delay overhead, the comparison occurs in the background. While the RB0 output is sent to the outside, the comparator checks the data consistency between both RBs. In the case of mismatch, the comparator initiates an error signal alerting a comparison mismatch. The error signal stalls the pipeline as there would be a chance of error in the last recent read value. The mirroring of data and the comparator combined provides an effective way for error detection. With spatially separated copies, the probability of having identical bit upsets in the original and backup register is close to zero.

To perform error correction, the correct copy needs to be distinguished from the faulty copy. For this purpose, we enhance the RF microarchitecture with a *parity generator* calculating parity bits for both original data and its backup copy upon every write access. To reduce the power overhead and to increase error correction capabilities, we split parity generation across original and backup copy (highlighted in Figure 7). The parity of even bits (bit 0, 2, 4) is only generated for the copy in RB0, and stored as

Fig. 8. Parity checking in the ARRA approach.

P_E . Odd bits parity (P_O) is only generated for backup data written to RB1. Upon a mismatch detected in the comparator, P_E and P_O are used for detecting the faulty copy of data.

The ECU is responsible for identifying the faulty copy of data. Figure 8(a) highlights the basic flow for error correction. Both copies (primary and backup registers) with their own parities (P_E and P_O) are forwarded to ECU. The ECU computes the parity for even bits and for odd bits on both the original as well as on the backup copy. The parity checker identifies the faulty copy of data by checking the validity of P_E and P_O for odds and evens bits of both primary and backup registers. Through setting the select signal, the correct copy will be send to the output. In the case that both copies have been corrupted (i.e., parities do not match), an exception will be raised, indicating a noncorrectable error.

To illustrate the working principle, Figure 8(b) shows a simple example based on an 8-bit register. The active register's content is 10101010b (binary). Therefore, the parity for even bits stored alongside the active register is 0, and the parity for the updates start alongside the passive register is 0. Assuming that a particle strikes the active register and causes three bit-flips creating a value of 10010010b (the parity bits as well as the backup data are not impacted). Upon reading, the error correction unit detects a mismatch between the two copies and issues a stall. Then, in the ECU, the parity checker again calculates the parity bits based on the read values for active and backup registers. Due to the double bit-flip in the odd bits of R0, the new calculated parity matches the stored parity, and the error cannot be detected. Conversely, the even bits of R0 yield a different parity than the one stored alongside the mirror register. Therefore, R0 is flagged as faulty. For the backup register, the retrieved and the computed parities match. Consequently, the backup R0 is flagged as valid. Its the data is forwarded into the data path, and execution continues as normal.

The introduced parity bits are similar to other physical bits and thus are equally vulnerable to soft errors. An SBU of any parity bit will not alter execution because both data copies are still intact, and error correction is not triggered. The parity bit will simply be regenerated (thus corrected) upon starting of next vulnerable period (write after the last read access). In the case of an MBU involving both a parity and a content bit, the error will be detectable due to mismatch occurrence between primary and backup. However, the error is not correctable. In this article, we target MBUs that occurred in multiple adjacent bits (see Section 3.2). By physically separating parity from its content bits, MBUs across content bits and its corresponding parity bit can be avoided.

Overall, the error detection and correction is efficiently realized through a register compare and parity check. It detects and corrects SEUs and MBUs that occurred both in

Fig. 9. ISA extension.

primary or backup registers. By separating the error detection from correction (parity checking), the correction is only performed when a mismatch error occurs. Furthermore, the error detection can be done in background without adding additional latency to the RF read critical path. When the error detector recognizes a mismatch, it stalls the pipeline for one cycle and then invokes error correction.

The proposed microarchitecture extension can also be applied to a monolithic RF when one single physical register bank contains both general- and special-purpose registers. Additional ports may be needed to read the backup register. For a heterogeneous RF, additional ports can be avoided depending on the parallel read accesses allowed by the ISA. For a monolithic RF, additional ports are required; however, they do not need to be exposed to the processor's datapath.

5.3. ISA Extension

To support the associated operations of the introduced MR register, the ISA is extended in two ways: (1) loading the intermediate value to MR and (2) preserving MR content across the sequence of function calls.

First, a new instruction (MAP instruction) is introduced for loading an immediate value to the MR. This bitmap value determines the set of registers for register mirroring. To use a register for mirroring, its corresponding bit in MR needs to be set to one. Figure 9(a) exemplifies an RF with six registers (R0–R2 as general-purpose registers and SR0–SR2 as special-purpose registers) assuming R0–R2 are directly mapped to SR0–SR2. In the result, a 3-bit MR register is sufficient to configure runtime mirroring. Loading the intermediate value of *110* indicates the duplication of R0's content to SR0 and R1's content to SR1 (highlighted in Figure 9(a)). Conversely, the bit for R2 is not set. Therefore, R2 and SR2 operate independently. Figure 9(b) also shows an example of instrumented binary to judge the instruction footprint for four functions (F0, F1, F2, F3). Each function has its own dedicated MAP instruction for setting register mirroring with respect to its register activity. Later, Section 5.4 (see Figure 11(a)) will describe how ARRA identifies the active/passive registers per function to set its corresponding MR bitmap. Note that if the number of backup registers exceeds the number of bits available in a constant field, multiple MAP instructions may be required.

Second, to support function calls, the content of MR needs to be preserved upon function calls and pushed to/popped from the stack (please see the next section for function call support for more details). Briefly, we consider the MR to be similar to Frame Pointer (FP) in respect to call preservation rules and function-call conventions. The associated MR stack operations are realized either by a new instruction or by extending already-available instructions. For our case study of the Blackfin core, we chose to extend the Link/Unlink instructions to support the associated MR stack operations as well. In the result, the code size overhead is lower not needing additional MR push/pop instructions.

Fig. 10. Flow of ARRA binary analysis for register mirroring.

5.4. Application Binary Analysis

With the RF microarchitecture and ISA extensions in place, the next two subsections deal with the two major software support components. Figure 10 illustrates the flow with the *binary analyzer* and the *binary instrumentation*. The binary analyzer detects unused registers within functions and identifies the register mirroring configuration per function. The binary analyzer operates at the binary level after all compiler optimizations and library linking. In this way, the analysis achieves a comprehensive insight of register accesses in the program. The binary instrumentation uses the results of the binary analyzer to generate a new binary including the register mirroring instructions. This subsection (Section 5.4) discusses the binary analyzer, while Section 5.5 introduces the binary instrumentation.

The Binary Analyzer by itself is a composition of (a) the *Function Graph Generator*, (b) the *Register Profiler*, and (c) the *Dependency Analyzer*. At first, (a) the Function Graph Generator extracts all callable functions and creates a Function Call Graph (FCG) out of the application binary. Next, (b) the Register Profiler parses each function body detecting register accesses. Following that, (c) the Dependency Analyzer traces the register dependency between the callee and caller. The Dependency Analyzer uses the FCG, and register accesses information to identify registers for backup, as well as identifying those registers that need to be stored/restored into the stack to avoid data loss during the use as backup.

Our ARRA approach assumes that each function has one set of working registers (active registers). Working registers are registers that are potentially accessed within the function's body. A register is considered to be active as long as it is referenced somewhere inside function body, regardless of runtime conditions. Conversely, registers not referenced inside the function body are considered passive thus unused registers. Although this approach is somewhat conservative, it makes register mirroring independent of input data—guaranteeing unaltered functionality of the program independent of input data and the execution path within the function body.

Following the direct mapping policy, unused registers are utilized for mirroring the content of their corresponding vulnerable registers. Similarly, a vulnerable register can only be duplicated if its corresponding backup register is passive (unvulnerable). Based on this analysis, the binary analyzer generates a proper bitmap value for the MR reflecting the register mirroring policy.

To exemplify the analysis, Figure 11(a) shows a FCG including five functions (f0–f4) based on the functions illustrated in Figure 9(b). Figure 11(a) assumes that the RF has only six registers in two register banks (R0–R2 are general-purpose registers and SR0–SR2 are special-purpose registers). Following the direct mapped register mirroring, SR0, SR1, and SR2 are backup candidates for R0, R1, and R2, respectively. In function f0, R1 and R2 are active while their backup candidates (SR1 and SR2) are passive. In the result, the contents of R1 and R2 are mirrored into SR1 and SR2. Conversely, as both R0 and SR0 are active within function f0, R0 cannot be backed up.

Fig. 11. An example of application binary analysis at function level.

To enable register mirroring, MR should be set to 011b in f_0 . In function f_2 , all backup registers are passive. Thus, all general-purpose registers are mirrored by MR = 111b.

Overall, ARRA uses a static application binary analysis and does not need simulation-based profiling. The static analysis has the advantage of avoiding long profiling runs. Furthermore, it is immune against ambiguity through data-dependent register accesses, as it conservatively assumes that all registers occurring within a function are active. The binary analysis at function-level granularity is agnostic to the fine-grained variations in register activity within a function and only tracks register utilization across function-calls. With this, ARRA is independent of data-dependent register accesses and has identical register mirroring for any input data set. At function-level granularity, binary analysis relies on the function call preserved rules to track the register access across the functions. In the following, we further expand that how ARRA utilizes the function-calls preservation rules to keep the correct execution of the program.

5.4.1. Dependency Analysis. Using registers as a backup carries the potential of data loss. ARRA's register mirroring has to respect calling conventions to protect backup registers against potential data loss. Calling conventions defined in the Application Binary Interface (ABI) govern how function arguments are passed and how registers are accessed across a sequence of function calls. The ABI divides registers into three different types: dedicated, call-preserved, and scratch registers.

The content of dedicated registers (e.g., Frame Pointer [FP] and Stack Pointer [SP]) must be always valid for every function. Similarly, the content of call-preserved registers need to be preserved (saved by the callee before they are used). Therefore, the registers appear unchanged to the caller. Registers used for keeping global data and memory addresses belong to call-preserved registers. In contrast, there is no need to preserve the content of scratch registers. In other words, scratch registers are not preserved across function calls. Many special-purpose registers, like those used for circular addressing or zero-hardware-loop implementation in the Blackfin, belong to this group.

In the Blackfin processor, registers (#4–7) of data registers and (#11–15) of pointer registers belong to the call-preserved category, and the remaining are scratch registers. Scratch registers also exist in other architectures; most of the Vector Floating-Point (VFP) registers in ARM-based architectures belong to scratch category [ARM Inc 2012]. Similarly in TI C64xx/C32xx series, half of general-purpose registers are configurable to use as circular-addressing and belong to scratch register category [Texas Instruments (TI) 2010].

There is a potential data loss for call-preserved registers when used for register mirroring. To avoid data loss, the dependency analyzer applies one rule: the content of

ALGORITHM 1: Procedure of detecting preserved registers

```

1: for  $F$  belongs to  $Functions$  do
2:   for  $R$  belongs to  $Passive(F)$  do
3:     for  $P$  belongs to  $Parents(F)$  do
4:       if  $R$  is  $Call - Preserved$  and  $R$  belongs to  $Active(P)$  then
5:         add  $R$  to  $Preserved(F)$ 
6:       end if
7:     end for
8:   end for
9: end for

```

a passive call-preserved register in a callee function that is used as a backup and is active in at least one of the caller functions has to be preserved. Therefore, extra stack pushes should be inserted into the callee's prologue before setting the MAP instruction. Stack pops are inserted in the epilogue after restoring back the previous MR content (note that functions with multiple returns have multiple prologues).

Algorithm 1 illustrates the procedure of detecting registers that need to be preserved. The dependency analyzer operates on the FCG and creates four lists for each function: *parent* (caller functions), *active*, *passive*, and *preserved* (presenting the registers that need to be preserved). For each passive register, the algorithm checks all direct parents of the function. If the passive register is active at least in one of the parents and also the register belongs to the call-preserved category, its content should be preserved. Therefore, the register is added to the stack list of the function. Please note that each function only needs to keep track of its direct parents (caller functions).

Figure 11(b) shows the results of a dependency analysis for the same FCG as presented in Figure 11(a). For example, during function f_2 , register SR_0 is passive and used for mirroring while it has been active in the parent function f_0 . In the result, the content of SR_0 needs to be preserved to avoid data loss. Similarly, during functions f_3 and f_4 , SR_2 is used for register mirroring while it has been active in parents function f_1 ; thus, SR_2 is preserved before being used as a backup in f_3 and f_4 .

For dynamic function calls, since the address of the callee is determined at runtime, the caller has to preserve all active call-preserved and scratch registers to avoid any data loss. Register mirroring is disabled for Interrupt Service Routine (ISR) and exception handlers.

5.5. Binary Instrumentation

The previous steps have determined the register mirroring map as well as identified registers to be preserved. The next step is instrumenting the application binary so that it performs the register mirroring during execution on the ARRA-enhanced microarchitecture. The *Register Mirroring Instrumentation* tool instruments application binary at the function-level granularity. It inserts (1) MAP instructions for configuring register mirroring and (2) stack operations (push/pop) for call-preserved registers.

We limit binary instrumentation only to the prologue and epilogue of individual functions. Instrumenting the function body is not needed, as the register mirroring is fixed within individual functions. Furthermore, limiting the instrumentation to prologue/epilogue simplifies keeping track of relative branches within a function's body. At runtime, as a function gets called, the instrumented prologue is executed, which rearranges the register mirroring before function execution. By returning from the function, reaching an instrumented epilogue restores, the previous register mirroring configuration. For functions with multiple returns, multiple epilques need to be instrumented.

Fig. 12. Binary instrumentation at function-level granularity.

Figure 12(a) illustrates binary instrumentation; $f()$ as the caller function calls $g()$ being the callee. Zooming into callee function $g()$, the *MAP* instruction is inserted into the prologue for setting register mirroring configuration with respect to the register activity of $g()$. The *MAP* instruction loads an intermediate value to MR, enabling the register mirroring. Before updating MR, an extra push is required to preserve the caller's MR (highlighted in the prologue of Figure 12(a)). In the epilogue, before returning to the caller, the caller's MR is popped, restoring the previous configuration. In that, the MR follows call-preserve semantics (like FP).

Upon adding the instrumentation instructions, the jump and call addresses need to be updated correspondingly. As the newly introduced instructions are only bounded into the epilogue/prologue of each function, there is no need to keep track of relative branches within a function's body. However, relative addresses between function calls need to be updated correspondingly. For updating the target addresses, we employ a three-phase mechanism—each starting from the routine with the lowest address in the code proceeding to the highest address. At first, a linear list of branch/jump sources and their associated destinations is created for all jumps/branches, including absolute jumps and relative and indirect branches across functions. In the second stage, the based addresses of branches and absolute jump targets are updated based on the total number of instrumented instructions in prologue of current function and prologues/epilogues of all prior functions in the code section of program. In the third pass, the index addresses for relative branches are also updated in cases of multiple returns or indirect jumps within multiple functions in case relative addresses are affected.

5.5.1. Recursive Function-Calls. In recursive function-calls, when a single function continuously calls itself, there is a possibility for significant performance overhead due to newly instrumented instructions for register mirroring. Since the caller and callee are identical, they have an identical MR configuration. Yet still, the instrumentation configures an identical mirroring repeatedly in the prologue. To eliminate this overhead, we generate an interface function between the parent caller and the recursive callee function. Instead of directly calling the recursive function, the caller function is updated to call the interface function. The interface function is instrumented according to the register activity of the recursive callee. Consequently, the recursive callee functions remain untouched because the register mirroring is already setup. In the result, the recursive function can call itself without additional overhead.

The function's recursive pattern can be statically recognized, as the caller and callee are identical. The instrumentation tool inserts the proposed wrapper function upon detecting recursive-call patterns (illustrated in Figure 12(b)). In Figure 12(b), function $f()$ is the parent function, $g()$ is the recursive callee function, and $g'()$, is the wrapper function between $f()$ and $g()$. The epilogue and prologue of $g'()$ are instrumented with

the MAP instruction and stack operations in respect to $g()$'s register activities. When $g'()$ is called, it creates the mirroring configuration for the noninstrumented function $g()$. Consequently, $g()$ can recursively call itself without additional overhead.

6. EXPERIMENTAL RESULTS

This section comprehensively evaluates the benefits and costs of ARRA and compares it against the partially ECC-protected approach. It first analyzes the reduction of the vulnerability factor as a primary benefit. It then investigates into the associated cost, both in terms of performance (due to instrumented code) and power (due to microarchitectural extensions). Finally, it highlights the benefits based on the Mean-Work-to-Failure (MWTF) metric, which offers a more holistic view that takes into account correcting MBUs and the performance overhead.

6.1. Experiment Setup

In this article, we target the Blackfin core [Analog Devices Inc. (ADI) 2008], a DSP/RISC processor. The Blackfin has a 38 x 32-bit ISA register file (see the brief overview in Section 5.1). For our experiments, we developed an ISS based on the Trap-ADL [Fossati 2010]. Both Blackfin ISS and the ISA have been extended to support the enhanced microarchitecture with the MR register, and MAP instruction, as outlined in Sections 5.2 and 5.3. In the ARRA-enhanced Blackfin core, the general-purpose registers (#0–15) can be mirrored to special-purpose registers (#16–31). In the current implementation, the Zero-HW-Loop registers (#32–37) do not participate. With this, MR is 16 bits wide. To read and operate on the binary, we use *gcc (ld)* tools. The *objdump* tool generates assembly code out of binary which is then used as the input to the ARRA tool-chain. Benchmarks have been selected from MiBench [Guthaus et al. 2001] and DSPstone [Zivojnovic et al. 1994] representatives of workloads for control and signal processing applications, respectively. The benchmarks are compiled with *gcc (-O3)*.

We also compare ARRA's efficiency against next closest approach: partially ECC-protected RF [Lee and Shrivastava 2011]. For the comparison, we implemented a reference model for partially ECC-protected RF as described in Lee and Shrivastava [2011]. Here, the most frequent accessed registers are protected by EEC. We, selected all general-purpose registers (#0–15), namely the data and pointer registers, for ECC protection. For the purpose of power comparison, we developed a synthesizable VHDL model reflecting the Blackfin's heterogeneous RF, and implemented our proposed ARRA approach and partially ECC-protected RF. We use Synopsis Design and Power Compiler tools with 65nm technology (similar to the current technology for Blackfin processors) to estimate power consumption. The estimated power consumption are gathered under realistic workloads by employing a trace-based cosimulation of Blackfin ISS and the HDL models.

6.2. Vulnerability Improvement

To evaluate the ARRA approach and compare it against partially ECC-protected RF, this subsection uses Register File Vulnerability Factor (RFVF) [Lee and Shrivastava 2010] as a metric for evaluating RF vulnerability against soft errors (see the brief overview of RFVF in Section 3.3).

Figure 13 shows the for three RF architectures: nonprotected RF, ARRA register-mirrored RF, and partially ECC-protected RF. Figure 13 also separately shows average RFVF results for control and DSP benchmarks. Comparing ARRA against nonprotected RF, Figure 13 demonstrates that ARRA reduces the RFVF of control applications from 35% to 6.9%, on average. For control benchmarks, the amount of RFVF reductions varies across different benchmarks based on their available unused registers. Overall,

Fig. 13. RFVF in different register protection.

Fig. 14. ARRA performance overhead.

ARRA achieves a good RFVF reduction across all control benchmarks. The maximal RFVF reduction is achieved for BasicMath with 10-fold reduction in RFVF from 31% to 3.2%. Conversely, ARRA slightly reduces the RFVF for the DSP applications from 60% to 52%, on average. DSP applications use special-purpose registers more frequently, leaving little room for register-mirroring. Nonetheless, the difference between the RFVF of control and DSP applications is acceptable, since DSP applications typically have a much lower reliability demand compared to control applications.

For control applications, the partially ECC-protected RF yields a slightly lower RFVF (3.6%, on average) in comparison to ARRA (6.9%). The difference is more pronounced for DSP applications where partially ECC-protected achieves a 27% average (and ARRA 52%). While ARRA relies on unused registers for protecting vulnerable registers, the partially ECC-protected RF is by using ECC independent of register utilization. Partially ECC-protected RF can protect the general-purpose and pointer registers independent of special-purpose register utilization.

Overall, ARRA achieves a significant reduction in RFVF for control applications which is important as they have higher reliability demands (in comparison to DSP applications). By utilizing unused special-purpose registers, ARRA adds fewer additional resources compared to the partially ECC-protected RF. The next subsections compare the power overhead of both approaches.

6.3. Performance Overhead

The benefits of register mirroring come at a low cost of some performance overhead and code size increase. Compared to partially ECC-protected RF, ARRA introduces a slight performance overhead due to additional instructions for MAP initialization (MAP instruction) and stack operations for preserving the necessary call-preserved registers. With the instructions added in the instrumentation process, instrumented binary execution time slightly increase. This subsection evaluates the performance overhead (i.e., an increase in runtime) and growth in code size.

Figure 14 shows the performance overhead of ARRA. We measure performance overhead through execution on Blackfin ISS both for original and instrumented (ARRA-enhanced) binary. The performance overheads shown in Figure 14 vary considerably depending on the granularity of functions and the frequency of function calls. Frequently called functions with short execution time per invocation like in FFT and

Table I. ARRA Code-Size Overhead

Benchmarks	BitCnt	StrSearch	BasicMath	CRC	QuickSort	Fib	Control AVG	FFT	Matrix	Fir2	AES	DSP AVG
Overhead [%]	2.7	1.5	1.5	4.4	2.7	3.2	2.6	1.3	5.2	3.7	1.6	2.8

Fig. 15. RF power consumption in different register protection.

BasicMath yield a measurable performance overhead of 0.9% and 0.4%, respectively. In contrast, CRC, BitCnt, and Matrix with fewer function calls (each running longer) show practically no overhead. The performance overhead of benchmarks with recursive function calls is also minimal due to the wrapper functions introduced in Section 5.5.1. As an example, QuickSort and StrSearch use recursive functions. Their overhead without the wrapper amounts to 1.5% for each. With the wrapper, their overhead drops to 0.2% and to less than 0.1%. Overall, the performance overhead of ARRA is negligible in comparison to overall execution time. On average, we observe a performance overhead of 0.3% for the control benchmarks and 0.2% for DSP benchmarks.

Table I quantifies the code size overhead due to the added MAP instructions and the wrapper functions. We only consider the text section which is the only modified section. Both DSP and control benchmarks exhibit a low increase in text size (2.6% and 2.8%, respectively). Benchmarks consisting of many small (code-size) functions grow slightly more in text size (e.g., 5.2% for Matrix and 4.4% for CRC). Here, the few added instructions contribute more to the relative code size increase. Note that the code increase (which depends on the number of functions instrumented) does not translate to performance overhead (which depends on the number of calls to instrumented functions). An example is FFT, which showed with 0.9% the highest performance overhead (due few frequently called short functions). At the same time, it has lowest code-size overhead (1.3%) because its size is dominated by functions large in code size.

6.4. Power Overhead

The main motivation of ARRA is to utilize existing passive registers to reduce RF vulnerability requiring only few new resources. This subsection quantifies ARRA power efficiency and compares it against partially ECC-protected RF. To obtain power numbers, ISS traces drive the corresponding HDL models synthesized for 65nm (see Section 6.1).

Figure 15 compares the power consumption of the three RF architectures (nonprotected RF, ARRA, and partially ECC-protected RF). The average baseline power consumptions for nonprotected RF is 24mW (control) and 27mW (DSP). With ARRA, the RF power consumption slightly increases across all benchmarks to 31mW (control) and 32mW (DSP). ARRA increases power consumption by 29% (control) and 18% (DSP) over a nonprotected RF. The overhead is smaller for DSP applications, as register mirroring occurs less frequently.

The partially ECC-protected RF consumes 44mW for control applications and 54mW in DSP applications, on average. With this, it increases power consumption by 83% (for control) and by about 100% (for DSP) over a nonprotected RF. Conversely, the

ARRA-protected RF has a lower power consumption across all measured applications. It consumes 30% and 40% less power than a partially ECC-protected RF (31mW vs. 44mW in control and 32mW vs. 54mW in DSP). Looking only at the power increase over a nonprotected RF (ie. power overhead), ARRA has a 3x to 5x smaller overhead than the partial ECC-protection (29% vs. 83% in control and 18% vs. 100% in DSP).

The reported 29% power overhead includes both dynamic and static power. The static power overhead is fairly stable across all benchmarks. The base RF (not-protected) consumes about 11mW and ARRA-enhanced RF only minimally more with 12mW. Conversely, the ARRA dynamic power overhead is directly related to the mirroring opportunity and its associated additional read/write operations. Looking at Figure 15, the power overhead in control benchmarks is relatively higher than DSP benchmarks due to their higher mirroring potentials (leading to a higher number of mirrored R/W access). As an example, our profiling on FFT benchmark shows that only 21% of R/W access are mirrored. This results in an RFVF reduction from 39% to 27% in cost of 29% RF power overhead.

In addition to the power consumption increase (due to the RF microarchitecture extensions), ARRA also introduces a slight performance overhead (due to MAP instructions) as discussed in Section 6.3). The performance overhead ultimately increases power consumption (due to a longer runtime). To analyze combined power overhead (due to RF microarchitecture extensions and increased runtime), we estimate the core power consumption. Joy [2007] reports a power consumption of the Blackfin core with 254mW at 500MHz, which we use as a base consumption (identical for all applications for purpose of this comparison). With the ARRA-protected RF, the consumption increases to 261mW for control and 259mW for DSP applications. In comparison, a core with ECC-protected RF is more power hungry with 273mW and 281mW. The power overhead due to the additional MAP instructions is minimal and the benefits of ARRA's power efficient microarchitecture clearly dominates.

To give an indication of area increase, we use the cell area as reported by the design compiler. The ARRA-enhanced RF uses 21% more area than the base RF ($13.6\mu\text{m}^2$ vs. $11.2\mu\text{m}^2$). The area increase with ARRA is slightly smaller than what has been reported for an ECC-protected RF (30% [Lin et al. 2014]). Even though the area comparison is favorable for ARRA, the power consumption is of higher importance.

Overall, the results show that ARRA is much more power-efficient. ARRA eliminates the costly ECC generation and checking in every read/write access. Instead, ARRA takes the advantage of current unused registers leading lower power. In addition, by separating the error correction and detection units (illustrated in Section 5.2), the error detection (parity checking) is only performed when a mismatch error occurs, minimizing power overhead.

6.5. Mean Work To Failure (MWTF)

One of the advantages of ARRA approach is correcting Multiple Bits Upsets (MBUs) as illustrated in Section 5.2. The RFVF is a very good metric to demonstrate ARRA effectiveness in improving RF vulnerability. However, RFVF does not take MBUs error correction and execution time increase into account. In this section, we use the metric MWTF to quantify the combined effect of MBUs error correction while slightly increasing execution time.

$$MWTF = (\lambda \times AVF \times \text{execution time})^{-1} \quad (2)$$

MWTF, introduced by Reis et al. [2005b], is a generalized metric representing the longevity of the system under the presence of errors. MWTF captures the execution times and error rates to estimate the longevity of system under evaluation. Equation (2) highlights the general formulation of MWTF for a given system, proposed by Reis et al.

Fig. 16. Normalized MWTF over nonprotected RF.

[2005b]. The MWTF definition leaves it open how to quantify the amount of work completed. For the purpose of this article, we define the amount of work as the number of executed instructions without error. In the right side, AVF refers to Vulnerability Factor of system under presence of errors, and λ quantifies the error rate. The value of λ depends on technology size. In general, by shrinking transistor dimensions, λ increases. Following Equation (2), a lower vulnerability and execution time lead to higher MWTF.

$$\lambda_N = (1 - q) \times q^{N-1} \times \lambda \quad (3)$$

To quantify the type of errors (SEU vs. MBUs), we use Equation (3) proposed in Chugg et al. [2004]. Chugg et al. [2004] describe the error rate for different number of bit-flips as an exponential probability distribution. In Equation (3), λ_N presents error rate for N number of bit-flips and q is a constant coefficient indicating how many errors are MBUs among all occurred errors in RF. In other words, q is a ratio indicating the probability of MBUs. Thus, the value of q is always between 0 and 1. Overall, Equation (3) provides unique probabilities for SEU and MBUs with different number of bit-flips. Based on Equation (3), the probability of $N + 1$ bit-flips is always less than the probability of N bit-flips.

$$MWTF_{RF} = \left(\sum_N \lambda_N \times RFVF \times execution\ time \right)^{-1} \quad (4)$$

To quantify the effect of MBUs in MWTF, we merged Equation (3) and Equation (2). The resulting Equation (4) shows the MWTF for a register file under varying number of bit-flips (λ_N). For this, the more general AVF has been replaced with RFVF. Figure 16 presents the MWTF (normalized over nonprotected RF) for both ARRA and partially ECC-protected RF. MWTF is calculated according to Equation (4). Following Chugg et al. [2004], we assume $\lambda = 7.4 \times 10^{-4}$ FIT/bit (1 FIT/bit is one failure every 10^9 hours per bit) for 65nm technology. Different values have been reported for MBU constant q . While Fazeli et al. [2010] assume 10% of errors appear as MBUs ($q = 0.1$), Georgakos et al. [2007] showed a 5x higher value 50% ($q = 0.5$). For the purpose of our exploration, we consider these as boundaries, showing both $q = 0.1$ in Figure 16(b), and $q = 0.5$ in Figure 16(a).

Looking at Figure 16(a) and Figure 16(b), ARRA achieves significant improvements for control applications in both cases with MWTF increasing 5x ($q = 0.5$) and 7.5x ($q = 0.1$). For DSP applications, ARRA still increases MWTF by 1.3x and 1.4x. The benefits in DSP applications are smaller than in control applications due to their limited mirroring potentials. ARRA increases MWTF for all benchmarks even for the ones with comparatively higher performance overhead. The improvement in MWTF

completely dwarfs ARRA's performance overheads. ARRA achieves the largest MWTF improvement in *BasicMath* with 8x and 10x increase for $q = 0.5$ and $q = 0.1$. Even in FFT, in which ARRA occurs a 1% overhead, it still improves MWTF by 1.5x and 1.6x.

Comparing ARRA against the partially ECC-protected based on Figure 16(b) and Figure 16(a) is in favor for ARRA. For control applications, ARRA's MWTF increase is 2.8x and 1.4x larger than in partially ECC-protected RF. For DSP applications, the partially protected RF delivers slightly higher MWTF for $q = 0.1$ due to the lower rate of MBUs. Conversely, for $q = 0.5$ (higher rate of MBUs), ARRA and partially protected RF have comparable impact on MWTF.

Overall, ARRA clearly outperforms partially ECC-protected RF for control applications based on MWTF. The MWTF enhancement is more pronounced for with more MBUs ($q = 0.5$) as ARRA can correct multiple errors up to three consecutive bit-flips. Considering the power efficiency of ARRA, ARRA is very beneficial solutions particularly for control application in which many special-purpose registers are unused.

7. DISCUSSION

Our experimental results demonstrate the tremendous benefits of ARRA. On average, ARRA reduces RFVF from 35% to 6.9% and increases MWTF between 5x and 7.5x for control benchmarks. Furthermore, ARRA power overhead is much lower than partially ECC-protected RF (29% compared to 83% in control applications). The higher power saving of ARRA stems from utilizing available passive registers for mirroring the content of active registers. To achieve this, ARRA leverages a static binary analysis to detect register usage (active/passive). Then, ARRA instruments binary with mirroring configurations (MAP) to guide the ARRA-enhanced RF. During the execution, ARRA-enhanced RF performs register-mirroring following direct-map policy. ARRA reliability improvement is achieved at a very low cost of performance overhead, 0.2% (control) and 0.3% (DSP), due to executing MAP and stack instructions.

So far, ARRA has implemented an index-based direct mapping between active and backup register (e.g., R0 maps to SR0). However, this may have limited benefits if both registers are frequently used as it limits the mirroring opportunities. To improve register mirroring potentials (b), the direct-map mirroring can be optimized if average register activities are known (activity-aware direct-map). The next section investigates the potentials and benefits of activity-aware direct-map policy. It is followed by a brief discussion of the ARRA potentials and limitations.

7.1. Activity-Aware Direct Map

The goal of activity-aware register mirroring is to pair primary and backup registers based on their activity (rather than by index). Ideally, the most vulnerable register should be paired with the least utilized backup register. If sufficiently general activity traces can be found, activity-aware direct mapping may lead to utilizing more of the backup potential. We have profiled the utilization of Blackfin's RF for the same DSP and control benchmarks. Figure 17(a) lists the general-purpose registers with register index numbers on the x-axis sorted by descending utilization (y-axis). Similarly, Figure 17(b) shows the special-purpose registers sorted by ascending utilization. Given these profiling results, registers R15, R13, and R12 would be mapped to R20, R31, and R23, respectively, in the activity-aware direct-map policy.

An activity-aware direct map can be realized either (1) at the compile stage (through register renaming) or (2) at the RF microarchitecture level (design time). The opportunity of register renaming is very limited for heterogeneous RFs due to specializations of registers. In the result, designing RF microarchitecture with activity-aware direct-map is more feasible. This assumes that applications generally have similar register access distributions. This appears as a valid assumption, given that the register allocation

Fig. 17. Sorted Register Vulnerability Factor for Blackfin core.

Fig. 18. RFVF for activity-aware direct map.

is performed by the compiler and that special purpose registers are purpose bound. To evaluate the potential benefits, we have modified ARRA-enhanced Blackfin ISS to support activity-aware direct-map guided by the profiling results in Figure 17.

Figure 18 compares the RFVF between index-based and activity-aware direct-map policies. Most benchmarks (both DSP and control) benefit from the activity-aware direct-map as it opens more room for register mirroring. QuickSort benefits the most (from 12.4% to 8.2%). Conversely, few benchmarks (e.g., bcnt, StrSearch, and CRC) do not see a change in RFVF. On average, activity-aware mapping achieves RFVF of 5.6% better results than the direct-mapped counter part with 6.9% for control benchmarks. A similar trend is visible for DSP applications with 47% instead of 52%. Overall, the activity-aware direct-mapped register mirroring utilizes more mirroring potentials thus achieves a lower RFVF.

The presented activity-aware direct-mapped register mirroring uses a static mapping determined at design time. Hence, the mapping is identical for all applications. A further potential improvement could offer a programmable direct map, which would facilitate application-specific direct-mapped mirroring. This however would increase hardware complexity and power consumption. Conversely, the anticipated benefits are low. Even a full associative mapping did not significantly outperform the direct-mapped policy in our estimations shown in Figure 6.

7.2. ARRA Potentials and Limitations

Overall, ARRA is a hybrid HW/SW approach to improve RF reliability. ARRA aims to maximally utilize already available resources and minimize hardware and power overheads. ARRA is primarily suited for in-order cores with their transparency between ISA (architectural) registers and physical registers. At the same time, out-of-order cores, in principle, could benefit from ARRA. A possible approach would be to guide register renaming to limit the number of physical registers per function—with respect to function's register demands. In this scenario, a portion of physical registers is used for

renaming, and the remaining ones can be used to mirror vulnerable (active) registers. Nonetheless, applying ARRA to out-of-order cores remains future work.

The benefits of ARRA's RF protection with low power overhead come with some limitations: (1) a slight inefficiency for shared libraries and (2) a slight performance overhead due to the MAP instructions. To support shared libraries, they have to be analyzed and instrumented. However, the actual target of a shared library call is determined at runtime and, hence, unknown to the instrumentation. As a result, ARRA will need to assume that all registers are used in the first shared library function call. This will lead to additional stack operations for storing/restoring register values. Nonetheless, subsequent calls within the shared library can still use the normal ARRA principle.

ARRA's performance overhead is already very low. Its static binary analysis makes ARRA independent of the data input. On the down side, it also makes ARRA unaware of functions' contribution to the program execution. In result, all functions are instrumented including nonrecursive but frequently called small functions with negligible contribution to execution and RF vulnerability. This in turn may lead to an avoidable performance overhead. Alternatively, a hybrid static/dynamic binary analysis could be employed that takes function execution frequency and duration into account. It would then only instrument functions with significant register-mirroring potential and by that reduce performance overhead at minimal RFVF cost. Nonetheless, the benefits of static analysis outweigh the potential loss in performance, which is anyway only in the subpercent range.

A different potential approach aims for fewer MAP instructions similar to Section 5.5.1, which avoided MAP instructions of recursive function calls through an instrumented interface function. More generally, if the caller's unused register set is a super-set of the callee's unused register set, the callee does not need a new MAP instruction. However, this reduces the mirroring coverage when the callee has more mirroring potential than the caller. The benefit is rapidly diminishing, when considering multiple caller functions as only the common subset would then be mirrorable. An orthogonal approach is to take advantage of variable length instructions (up to 64 bits) of the Blackfin core. A 64-bit call instruction could be added that includes the MAP value to control register mirroring. This would eliminate the overhead of a separate MAP instruction (except when limited through the memory interface) at cost of a longer call instruction. However, this solution is limited to processors with variable length instructions.

8. CONCLUSION

In this article, we introduced ARRA to improve the reliability of a large, heterogeneous register file against soft errors. ARRA utilizes unused (passive) registers as the backup of vulnerable (active) registers. ARRA realizes register mirroring through an interplay of application binary analysis and instrumentation, ISA extension, and RF microarchitecture enhancement. We applied ARRA to a Blackfin processor and validated the effectiveness using MiBench and DSPStone benchmarks. Our results demonstrate the tremendous potential of ARRA specifically for control applications. ARRA reduces the RFVF from 35% to 6.9%. With being able to correct up to 3 adjacent bit-flips, it increases the MWTF by 5x to 7.5x. Executing ARRA instrumented binaries comes at a low (0.3%) performance loss. ARRA clearly outperforms partially ECC-protected RF for control applications with a 1.4x to 2.8x larger increase in MWTF. At the same time, ARRA is more power-efficient, carrying only 29% power overhead compared to 83% in partially ECC-protected RF. Altogether, the results demonstrate that RF reliability can be significantly improved by utilizing already available unused registers with minimal power overhead to RF.

REFERENCES

- H. Amrouch and J. Henkel. 2011. Self-immunity technique to improve register file integrity against soft errors. In *International Conference on VLSI Design (VLSI Design'11)*. 189–194.
- Analog Devices Inc. (ADI). 2008. ADSP-BF52x blackfin processor hardware reference (volume 1 of 2). *Reference Guide, Part Number 82-000525-03* (May 2008).
- ARM Inc. 2012. *Procedure Call Standard for the ARM Architecture, Current through ABI Release 2.09*.
- José L. Ayala, Alexander Veidenbaum, and Marisa López-Vallejo. 2003. Power-aware compilation for register file energy reduction. *International Journal on Parallel Programming* 31 (Dec. 2003), 451–467. Issue 6.
- J. R. Azambuja, A. Lapolli, L. Rosa, and F. L. Kastensmidt. 2011a. Detecting SEEs in microprocessors through a non-intrusive hybrid technique. *IEEE Transactions on Nuclear Science* 58, 3 (2011), 993–1000.
- J. R. Azambuja, A. Lapolli, L. Rosa, and F. L. Kastensmidt. 2011b. Detecting SEEs in microprocessors through a non-intrusive hybrid technique. *IEEE Transactions on Nuclear Science* 58, 3 (June 2011), 993–1000.
- Alfredo Benso, Silvia Chiusano, Paolo Prinetto, and L. Tagliaferri. 2000. A C/C++ source-to-source compiler for dependable applications. In *International Conference on Dependable Systems and Networks (DSN'00)*. 71.
- Jason A. Blom, Shantanu Gupta, Shuguang Feng, and Scott Mahlke. 2006. Cost-efficient soft error protection for embedded microprocessors. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'06)*. 421–431.
- A. M. Chugg, M. J. Moutrie, and R. Jones. 2004. Broadening of the variance of the number of upsets in a read-cycle by MBUs. *IEEE Transactions on Nuclear Science* 51, 6 (Dec. 2004), 3701–3707.
- Ryan Collins, Fernando Alegre, Xiaotong Zhuang, and Santosh Pande. 2006. Compiler assisted dynamic management of registers for network processors. In *International Conference on Parallel and Distributed Processing (IPDPS'06)*. 53–53.
- M. Fazeli, A. Namazi, and S. G. Miremadi. 2010. Robust register caching: An energy-efficient circuit-level technique to combat soft errors in embedded processors. *IEEE Transactions on Device and Materials Reliability* 10, 2 (June 2010), 208–221.
- Luca Fossati. 2010. Trap-ADL SystemC Simulator: User Manual. Retrieved from <http://code.google.com/p/trap-gen/>.
- G. Georgakos, P. Huber, M. Ostermayr, E. Amirante, and F. Ruckerbauer. 2007. Investigation of increased multi-bit failure rate due to neutron induced SEU in advanced embedded SRAMs. In *IEEE Symposium on VLSI Circuits*. 80–81.
- D. R. Gonzales. 1999. Micro-RISC architecture for the wireless market. *IEEE Micro* 19, 4 (July 1999), 30–37.
- Xuan Guan and Yunsu Fei. 2010. Register file partitioning and recompilation for register file power reduction. *ACM Transactions on Design Automation of Electronic Systems* 15, 3 (June 2010), 24:1–24:30.
- M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE International Workshop on Workload Characterization*. 3–14.
- Jie Hu, Feihui Li, Vijay Degalahal, Mahmut Kandemir, N. Vijaykrishnan, and Mary J. Irwin. 2009a. Compiler-assisted soft error detection under performance and energy constraints in embedded systems. *ACM Transactions on Embedded Computing Systems* 8, 4, Article 27 (July 2009), 30 pages.
- Jie Hu, Shuai Wang, and S. G. Ziavras. 2009b. On the exploitation of narrow-width values for improving register file reliability. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 17, 7 (July 2009), 953–963.
- B. Joy. 2007. Estimating power for ADSP-BF534/BF536/BF537 blackfin processors. *Technical notes on using Analog Devices DSPs, processors and development tools*. Retrieved from <http://www.analog.com/ee-notes>.
- D. Katz, T. Lukasiak, and R. Gentile. 2009a. Understanding advanced processor features promotes efficient coding. Technical Report, Analog Devices Inc.
- D. Katz, T. Lukasiak, and R. Gentile. 2009b. Understanding advanced processor features promotes efficient coding. Technical Report, Analog Devices Inc.
- Jongun Lee and Aviral Shrivastava. 2009. A compiler optimization to reduce soft errors in register files. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'09)*. 41–49.
- Jongun Lee and A. Shrivastava. 2010. A compiler-microarchitecture hybrid approach to soft error reduction for register files. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 29, 7 (July 2010), 1018–1027.

- Jongeeun Lee and A. Shrivastava. 2011. Static analysis of register file vulnerability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 4 (April 2011), 607–616.
- Y. Lin, M. Zwolinski, and B. Halak. 2014. An energy-efficient radiation hardened register file architecture for reliable microprocessors. In *Silicon Errors in Logic - System Effects (SELSE'14)*.
- G. Memik, M. T. Kandemir, and O. Ozturk. 2005. Increasing register file immunity to transient errors. In *Design, Automation and Test in Europe (DATE'05)*. 586–591.
- P. Montesinos, Wei Liu, and J. Torrellas. 2007. Using register lifetime predictions to protect register files against soft errors. In *International Conference on Dependable Systems and Networks (DSN'07)*. 286–296.
- John Oliver, Ravishankar Rao, Paul Sultana, and Jedidiah Crandall. 2004. Synchroscalar: A multiple clock domain, power-aware, tile-based embedded processor. In *International Symposium on Computer Architecture (ISCA'04)*. 150.
- Qualcomm Inc. 2011. Snapdragon S4 Processors: System on Chip Solutions for a New Mobile Age. Retrieved from <https://www.qualcomm.com/documents/snapdragon-s4-processors-system-chip-solutions-new-mobile-age>. 327–339.
- Semeen Rehman, Muhammad Shafique, and Jörg Henkel. 2012. Instruction scheduling for reliability-aware compilation. In *Design Automation Conference (DAC'12)*. 1292–1300.
- G. A. Reis, J. Chang, N. Vachharajani, S. S. Mukherjee, R. Rangan, and D. I. August. 2005a. Design and evaluation of hybrid fault-detection systems. In *International Symposium on Computer Architecture (ISCA)*. 148–159.
- G. A. Reis, J. Chang, N. Vachharajani, S. S. Mukherjee, R. Rangan, and D. I. August. 2005b. Design and evaluation of hybrid fault-detection systems. In *International Symposium on Computer Architecture (ISCA'05)*. 148–159.
- G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. 2005c. SWIFT: Software implemented fault tolerance. In *International Symposium on Code Generation and Optimization*. 243–254.
- Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. 2002. Modeling the effect of technology trends on the soft error rate of combinational logic. In *International Conference on Dependable Systems and Networks (DSN'02)*. 389–398.
- V. Sridharan and D. R. Kaeli. 2009. Eliminating microarchitectural dependency from Architectural Vulnerability. In *IEEE 15th International Symposium on High Performance Computer Architecture (HPCA'09)*. 117–128.
- H. Tabkhi and G. Schirner. 2012a. Application-specific power-efficient approach for reducing register file vulnerability. In *Design, Automation Test in Europe (DATE'12)*. 574–577.
- H. Tabkhi and G. Schirner. 2012b. ARRA: Application-guided reliability-enhanced registerfile architecture for embedded processors. In *International Conference on VLSI and System-on-Chip (VLSI-SoC)*. 299–302.
- Texas Instruments (TI). 2010. TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide. Retrieved from <http://www.ti.com/lit/ug/spru732j/spru732j.pdf>.
- Nicholas J. Wang, Justin Quek, Todd M. Rafacz, and Sanjay J. patel. 2004. Characterizing the effects of transient faults on a high-performance processor pipeline. In *International Conference on Dependable Systems and Networks (DSN'04)*. IEEE Computer Society, Washington, DC, 61.
- Jianjun Xu, Qingping Tan, and Wanwei Liu. 2010. Estimating the soft error vulnerability of register files via interprocedural data flow analysis. In *International Symposium on Theoretical Aspects of Software Engineering (TASE)*. 201–208.
- Jianjun Xu, Qingping Tan, and Huiping Zhou. 2011. Scheduling instructions for soft errors in register files. In *International Conference on Dependable, Autonomic and Secure Computing*. 305–312.
- Jun Yan and Wei Zhang. 2005. Compiler-guided register reliability improvement against soft errors. In *ACM International Conference on Embedded Software (EMSOFT'05)*. 203–209.
- S. Yoshimoto, T. Amashita, D. Kozuwa, T. Takata, M. Yoshimura, Y. Matsunaga, H. Yasuura, H. Kawaguchi, and M. Yoshimoto. 2011. Multiple-bit-upset and single-bit-upset resilient 8T SRAM bitcell layout with divided wordline structure. In *International On-Line Testing Symposium (IOLTS'11)*. 151–156.
- Xiangrong Zhou, Chenjie Yu, and P. Petrov. 2008. Compiler-driven register re-assignment for register file power-density and temperature reduction. In *45th ACM/IEEE Design Automation Conference*. 750–753.
- V. Zivojnovic, J. Martinez, C. Schlager, and H. Meyr. 1994. DSPstone: A DSP-oriented benchmarking methodology. In *International Conference on Signal Processing Applications and Technology*. 715–20.

Received July 2014; revised January 2015; accepted February 2015