Application-Specific Power-Efficient Approach for Reducing Register File Vulnerability

Hamed Tabkhi, Gunar Schirner
Department of Electrical and Computer Engineering, Northeastern University
Email: {*tabkhi, schirner*}*@ece.neu.edu*

***Abstract*—This paper introduces a power efficient approach for improving reliability of heterogeneous register files in embedded processors. The approach is based on the fact that control applications have high demands in reliability, while many special-purpose register are unused in a considerable portion of execution. The paper proposes a static application binary analysis which is applied at function-level granularity and offers a systematic way to manage the RF's protection by mirroring the content of used registers into unused ones. The simulation results on an enhanced Blackfin processor demonstrate that Register File Vulnerability Factor (RFVF) is reduced from 35% to 6.9% in cost of 1% performance lost on average for control applications from Mibench suite.**

I. Introduction

Soft errors caused by high energy particle strike are exponentially increasing with shrinking feature size, . Register File (RF) as a key component in the processor's performance has also a significant influence over the processor's reliability [1]. An investigation on ARM processors shows that more than 50% of errors affecting the correct state of processor comes from faults in the RF [1]. Access frequency to RF is quite high and errors are mostly exposed to the outside. Consequently, an error in RF easily propagates to data path and whole system leading to system crash or silent data corruption [2].

At the same time, RF is already one of the main sources of energy dissipation in embedded processors, consuming 15%-36% of the total processor power [3]. ECC (Error Correction Code) or fully duplicated RF may not be applicable to the entire RF due to their huge power overhead. Different RF caching solutions and partial protected RFs have been proposed in order to protect just the most vulnerable register words [4][1]. However, adding ECC or duplication even for part of RF still imposes a considerable amount of static and dynamic power to the processor. For instance, the required energy for an ECC checking is three times higher than a register accesses [5], while it can correct just a single bit flip.

In the recent years, processors are designed with larger register files to reduce the number of references to memory thus increasing performance. One trend of embedded processors is composing a complex register file out of heterogeneous register banks with specialized functionality [6]. Heterogeneous RFs aid embedded processors to efficiently support variety of applications from control to media applications. Heterogeneous RFs are currently implemented in embedded processors like ADI Blackfin families [6].

978-3-9810801-8-6/DATE12/©2012 EDAA

In this paper, we propose an application-based approach for improving the register file vulnerability against soft errors. The approach relies on the fact that control applications are more critical in terms of reliability while at the same time, they do not use all of the heterogeneous registers. The paper proposes an static application binary analysis which is applied at function level granularity and offers a systematic way to manage the RF's protection according to runtime conditions by mirroring the content of used registers into unused ones. The simulation results on Blackfin processor show that, for control applications the RF Vulnerability Factor is reduced from %35 to %7 on average with negligible performance overhead.

The rest of this paper is organized as following. Section 2 discuses the previous work. Section 3 presents background information of the paper. Section 4 introduces our proposed approach. The implementation and simulation results are presented in Section 5. Finally, Section 6 concludes this paper.

II. Previous Work

As ECC and duplication are too costly, current approaches try to protect register file partially. Overall, most of the previous approaches are hardware only solutions [2][7][1][4]. [1][4] propose a register caching mechanisms to protect just most frequent register words. They keep an extra copy of most recently accessed values protected either by embedding SEU-tolerant latches [4], or with CRC code [1] providing the correction capability. In these approaches, although RF protection can be applied efficiently, still the power overhead for register caching logic is considerable. In high-performance processors, some approaches take the advantage of the large physical RF using for register renaming [2][7]. [7] selectively applies ECC to those registers containing useful data. Alternatively, [2] duplicates the content of live registers to those micro-architectural unused ones. These approaches are not applicable to embedded processors who typically do not use register renaming.

In contrast to hardware-only solutions, recent approaches try to take application analysis into account [3][8][5]. It was shown that just by instruction rescheduling, RF Vulnerability can be reduced up to 30% [5]. [8] uses RF profiling information to detect the register words with long vulnerable period and stores their values into L1 cache assuming that L1 cache is already protected by ECC. Recently, Compiler-based approaches are considered to protect RF [3]. [3] presents a

static RF vulnerability estimation can be applied as a part of compilation process using for RF vulnerability optimization with this assumption that a part of RF is already protected.

III. Register Activity

The Register File Vulnerability Factor (RFVF) has become a common metric to evaluate the RF reliability. RFVF derives form Architecture Vulnerability Factor(AVF), and calculates the proportion of time that the RF is susceptible to soft errors [8]. At any point of time, a register is vulnerable if an error in the register potentially can affect the correct architectural state of program. For a regular register without any protection, the vulnerable period starts by a write operation, and extends until the last read. In the duration between the last read until the next write, the register is unused, thus it is also invulnerable. RFVF is equal to the average of vulnerability factors of all register words. RF's reliability improves throughout RFVF reduction either by shortening length of the vulnerable periods or by protecting vulnerable registers.

As a case study, we chose the Blackfin processor designed for supporting both control and signal processing applications [9]. Blackfin has a 32x32 bits ISA RF composed of eight data registers *(#0-7)*, eight pointer registers *(#8-15)*, and sixteen circular addressing registers *(#16-31)* counted as special-purpose registers, such as circular addressing registers [6]. Special-purpose registers mostly are employed in data streaming applications, e.g. accelerating computation through fast and continuous data memory addressing. Fig.1 shows the average utilization of different registers in the Blackfin's RF for two classes of applications. The results are gathered through runtime profiling of six and four optimized *(gcc-O3-compiled)* benchmarks from Mibench and DSPStone suites [10][11], respectively. The profiling highlight that in signal processing programs, all the special-purpose registers, specially circular addressing registers, are highly utilized. In contrast, in the control applications, the special-purpose registers are seldom used and do not have effective contributions to program execution. The low utilization of special-purpose registers has two reasons. First of all, compilers may not detect code patterns to benefit from heterogeneous registers in a large set of applications. Therefore, many embedded programmers handcraft their applications in assembly to freely utilize all resources. And secondly, many applications inherently do not require all resources. In contrast, the general-purpose registers are fairly active in the control applications thus these registers are highly vulnerable to the transient faults. As control applications work with a limited set of registers, there is an opportunity to utilize the existing unused registers in order to protect the vulnerable general-purpose registers.

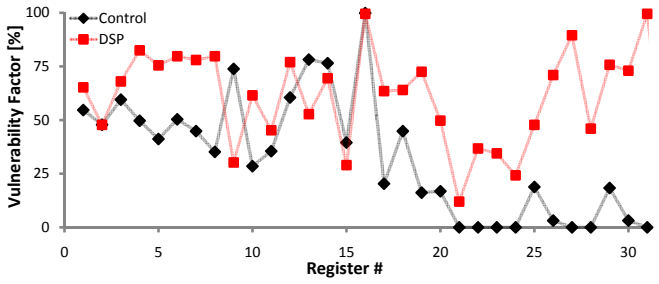

Fig. 1. Average register utilization in Blackfin's RF.

IV. Approach

Control applications have high demands in reliability. In these applications, while general purpose registers are highly used thus vulnerable, many special purpose register are unused in a considerable portion of execution. Consequently, through utilizing unused registers and mirroring the content of vulnerable registers, the overall vulnerability of the RF can be improved significantly, leading to an improved reliability of applications without imposing a significant overhead. Although, this approach can be applied to any sort of register file, it is particularly efficient for heterogeneous RFs with different register usage patterns.

A. Design Tradeoffs

The register mirroring can be done at different levels of granularity; application, function, loop, or basic block. An application granularity would be too coarse grain and cannot provide enough flexibility, while basic block would be too fine, resulting in too many configuration changes. Our profiling results show the considerable portion of unused periods are higher than *8000* instructions long (*65%*), appeared in either functions or loops with many iterations. Although, both offer a reasonable level of granularity, the loop-level would impose additional complexities. Detecting the loops with many-iterations and unused register periods in the loop are data dependent in many cases, requiring additional analysis. In contrast, functions provide isolation over register accesses in the program code. Moreover, the predefined rules over register access within functions simplify tracking the register dependencies. Consequently, we use function-level granularity is used for detecting unused periods.

In the hardware layer, different register mapping policies can be applied with different flexibilities. The register mapping defines the number of register candidates that can be used as backup. In our preliminary estimation, we observed the most significant drop in RF vulnerability from no mirroring (>50%) to *Direct* mapping (about 12%). *4-ways Associative* mapping only slightly improves the RFVF over *Direct* mapping (down to about 6%)[1]. In contrast, the hardware complexity required to support associative mapping is significantly higher [2] In order to achieve a considerable improvements in RFVF with minimum overhead, we choose *Direct Mapping* when one backup candidate for each vulnerable register.

B. Instruction Set Architecture (ISA) Extension

In order to provide control over RF configuration, the ISA needs to be extended with a new register called *Map Register (MR)*. We consider *MR* as an architectural register. The width of *MR* is equal to the number of registers in RF which are in register mirroring scope.

[1]Obtained estimation results are omitted due to the space constraints.

[2]Due to space constraints, the micro architecture implementation details are not scope of this paper.

Additionally, the ISA has to be expanded for configuration and backup/restore. First, a new instruction is required for loading the *Mapping Register (MR)* with an immediate value to configure the register backup. *MR*'s bitmap value determines the set of registers utilized for backup. Second, to support function calls, the *MR* needs to be pushed to and popped from stack. We consider the *MR* stack operations similar to Frame Pointer (FP) in respect of call preserve rules. *MR* stack operations can be supported by dedicated new instructions or considered as the part of current instructions. Even though the basic operation is the same in both cases, the code size overhead is lower when embedding *MR* push and pop operations into current instructions. For example, in the Blackfin ISA, Link/Unlink instructions stores/restores FP and SP registers. For our experiments, we expanded the Link/Unlink instructions to handle *MR* as well.

C. Application Analysis and Instruction Insertion

Application analysis statically detects unused periods for each register. In this way, the analyzer gains a comprehensive insight of register accesses in the program. Our proposed static analysis does not require simulation profiling thus it avoids long simulation-based analysis time and avoids simulation ambiguity where register accesses may differ with input data. Our application analyzer operates on program binary as a post compiler stage, after all compiler optimizations and library linking. As discussed, register mirroring is managed at function-level granularity. Fig. 2 shows the instrumentation flow of the proposed approach in two main stages: Binary Analyzer and Instruction Insertion. The binary analyzer detects register unused periods cross function calls. It is a composition of function call graph generator, register profiling, and dependency analysis. At first, callable functions are detected and a function call graph is created. Next, the register profiler parses each function body detecting register accesses. Following that, the dependency analyzer traces the register dependency between callee and caller. Dependency analyzer uses the function call graph 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-registers. Finally, the instruction insertion utilizes the analysis results to generate a new augmented binary. While inserting new instructions it also updates relative addresses that are due to inserting new instructions.

Each function has a set of active registers accessed within the function's body. As long as a register is referred inside the body of a function, regardless of runtime conditions, counted

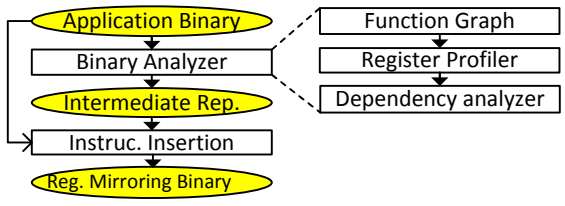

Fig. 2. Register mirroring instrumentation flow.

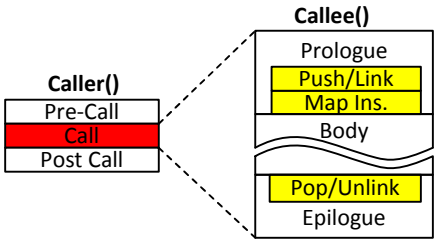

Fig. 3. Function call support.

as a used thus vulnerable register. The remaining registers, which are not referenced thus unused/idle within the function, can be used for register mirroring. Based on the direct mapping, any active register can only be duplicated where its corresponding backup register is unused. For every callable function, the profiler creates two lists of used and unused registers. For all instructions, a *Map* instruction is inserted in the prologue section (see Fig.3). The *Map* instruction loads an intermediate value to the *MR* enabling the register mirroring. Before updating the *MR*, an extra push is required in order to preserve the callers *MR*. In the epilogue, before returning to the caller, the callers *MR* is popped restoring the previous configuration, following call preserved semantics (like FP).

Registers can be divided based on how they have to be preserved over function calls: call-preserved, and scratch registers. The content of call-preserved registers needs to be preserved (saved by callee before they are used). Therefore, they appear unchanged to the caller. In contrast, the content of scratch registers does not need to be saved and restored. The scratch registers are not preserved across function calls. In register mirroring, there is a potential data loss if call-preserved registers are used for register mirroring while theirs contents are already used in caller function. In order to maintain a value of call-preserved registers, its content has to be stored and restored from stack memory. Every function has four lists; *Parent* (caller functions), *Used*, *Unused*, and *Stack* including the registers that need to be preserved. For each unused register of every callable function, the algorithm checks all direct parents of the function. If the unused register is used 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, each function only needs to keep track of its direct parents (caller functions) and does not care the descendant functions.

For preserved registers, extra pushes are inserted into the callee prologue before loading *MR*. Pops are inserted in epilogue after restoring the previous *MR* content. (Note: functions with multiple returns have multiple prologues). The same method applied for recursive function calls when both caller and callee are same. For dynamic function calls, since the address of callee would be determined in runtime, caller has to preserve all active call-preserved registers to prevent from any data loss. Register mirroring can be disabled for ISRs (Interrupt Service Routines). Please note, our dependency analysis only extends to call-preserved registers. Most of the

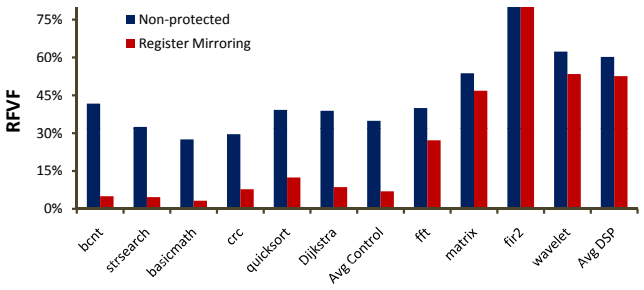

Fig. 4. Register File Vulnerability Factor with different register protection

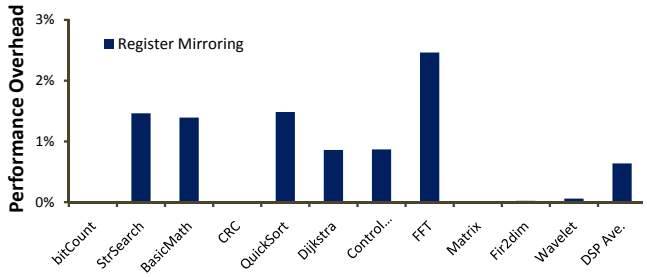

Fig. 5. Performance overhead of register mirroring

special-purpose registers, such as circular addressing registers are scratch registers [9][6]. These registers can be use in the callee function without any concern regarding their contents. This fact motivates us to focus on special-purpose registers for mirroring (minimizing additional stack operations) rather than using general-purpose registers.

V. Experimental Results

To evaluate our approach, we developed an Instruction-Set Simulator (ISS) for Blackfin processor [6] based on the Trap-ADL [12]. ISA and Blackfin ISS were extended in order to support the mphMap Register (MR) instructions outlined in Section IV-B. In our version, the general-purpose registers (#0-15) can selectively be protected in direct mapping to special-purpose registers (#16-31) depending on configuration. Benchmarks are selected form MiBench [12] and DSPstone [11] as workloads for control and signal processing applications, respectively. The benchmarks are compiled with gcc (-O3).

Fig. 4 shows the RF Vulnerability Factor (RFVF) calculated through ISS simulation for different register protection scenarios. In this figure, we compare RVFV reduction from the non-protected scenario, with our register mirroring. On average, the RFVF of control applications is considerably reduced (from 35% to 6.9%). DSP applications use the special-purpose registers much frequently, allowing only little room for our register mirroring. However, since we assume that DSP applications have a much lower demand on reliability than control applications, this can be acceptable. Our register mirroring prevents from costly ECC generating and checking in every read/write accesses which would significantly increase power consumption. Instead, our approach only duplicates the write operations and adds extra parity checking, with a simple comparator in the read access. Additionally, register mirroring takes the advantage of current unused registers leading to lower area, thus lower static power.

Finally, Fig. 5 shows the performance overhead of our approach due to executing additional instructions for RF configuration and stack operations. The results vary significantly with the application's function granularity. Frequently called small functions like in FFT yield a considerable overhead of 2.5%. In most benchmarks, however, the overhead is negligible in comparison to overall execution time (less than 1% on average for control applications). BitCount and CRC with few functions show negligible overhead compared to the recursively called Quicksort (1.5% overhead).

VI. Conclusion

In this paper, we introduced a power efficient approach for improving reliability of heterogeneous register files (RFs) in embedded processors. Control applications have high demands in reliability, while many special-purpose register are unused in a considerable portion of execution. Through an static analysis over program binary, approach utilizes the unused registers for protecting vulnerable registers, thus reducing the overall RF vulnerability. The simulation results on an enhanced Blackfin processor demonstrate that Register File Vulnerability Factor (RFVF) is reduced from 35% to 6.9% in cost of 1% performance lost on average for control applications from Mibench suite.

References

[1] J. A. Blom, S. Gupta, S. Feng, and S. Mahlke, "Cost-efficient soft error protection for embedded microprocessors," in *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, ser. CASES '06, 2006, pp. 421–431.

[2] G. Memik, M. Kandemir, and O. Ozturk, "Increasing register file immunity to transient errors," in *Proceedings of Design, Automation and Test in Europe*, march 2005, pp. 586 – 591.

[3] J. Lee and A. Shrivastava, "Static analysis of register file vulnerability," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, no. 4, pp. 607 –616, april 2011.

[4] M. Fazeli, A. Namazi, and S. Miremadi, "Robust register caching: An energy-efficient circuit-level technique to combat soft errors in embedded processors," *Device and Materials Reliability, IEEE Transactions on*, vol. 10, no. 2, pp. 208 –221, june 2010.

[5] J. Yan and W. Zhang, "Compiler-guided register reliability improvement against soft errors," in *Proceedings of the 5th ACM international conference on Embedded software*, ser. EMSOFT '05, 2005, pp. 203–209.

[6] "Blackfin processor programming reference mannual," *Analog Devices Inc.*, September 2008.

[7] P. Montesinos, W. Liu, and J. Torrellas, "Using register lifetime predictions to protect register files against soft errors," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN '07.*, june 2007, pp. 286 –296.

[8] J. Lee and A. Shrivastava, "A compiler optimization to reduce soft errors in register files," in *Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, ser. LCTES '09, 2009, pp. 41–49.

[9] D. Katz, T. Lukasiak, and R. Gentile, "Understanding advanced processor features promotes efficient coding," *Technical Report, Analog Devices Inc.*, 2009.

[10] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of IEEE International Workshop on Workload Characterization, WWC-4*, dec. 2001, pp. 3 – 14.

[11] V. Zivojnovic, J. Martinez, C. Schlager, and H. Meyr, "Dspstone: A dsp-oriented benchmarking methodology," in *Proceedings of the International Conference on Signal Processing Applications and Technology*, 1994, pp. 715–720.

[12] L. Fossati, "Leon2/3 systemc simulator: User manual." *Technical Report, Politecnico di Milano (Italy), http://code.google.com/p/trap-gen/.*