Application-Guided Power Gating
Reducing Register File Static Power

Hamed Tabkhi, Student Member, IEEE, Gunar Schirner, Member, IEEE,

Abstract—Power and energy efficiency are on the top priority list in embedded computing. Embedded processors tailed out in deep sub-micron technology have a high contribution of static power to overall power consumption. At the same time, current embedded processors often include a large Register File (RF) to increase performance. However, a larger RF aggravates the static power issues associated with technology shrinking. Therefore, approaches to improve static power consumption of large RFs are in high demand.

In this paper, we introduce AFReP: an Application-guided Function-level Register-file Power-gating approach to efficiently manage and reduce RFs static power consumption. AFReP is an interplay of automatic binary analysis and instrumentation at function-level granularity supported by ISA and micro-architecture extensions. AFReP enables runtime power-gating of registers during un-utilized periods, while applications can fully benefit from a large RF during utilized periods. To demonstrate AFRePs potential for reducing static power consumption, we have enhanced a Blackfin processor with the AFReP technology. Using AFReP, the RF static power is reduced on average by 64% and 39% for control and DSP applications, respectively. At the same time, AFReP only induces a very minimal overhead of 0.4% and 0.6%.

Index Terms—Register File (RF), Power-gating, Application binary.

I. INTRODUCTION

POWER and energy-efficiency have a very high priority in embedded/mobile computing. Energy efficiency is important for increasing operational time of battery-powered devices. In addition, it is also a necessary precondition in order to create headroom for performance improvements. To enable a 1.5x-1.7x performance gain, a 2x-2.5x power overhead is required [1, 2] which poses significant challenges to designing of mobile embedded systems. Consequently, processor and chip vendors invest considerable design time to improve power efficiency of embedded processors. Several approaches, ranging from system-level solutions down to circuit-level techniques, have been introduced to reduce energy consumption of microprocessors. Chip-wide Dynamic Voltage and Frequency Scaling (DVFS), power-gating [3] clock-gating [4], and multi-thresholds transistor [5] are only few examples. In order to maximize efficiency, these general approaches need to be complemented by component-specific solutions that optimize characteristics and utilization of individual components.

Register Files (RFs) are significant power consumers within processor cores. Power breakdowns of embedded processors show that RFs are consuming 15%-36% of overall processor core power [6], [7], [8], [9], [10], and up to 42% of core datapath [11]. Power assessments considerably differ based on RF size, processor complexity, and also the running application. As one example, 20% of the core power (including data and instruction caches) in a Blackfin processor (in 120nm technology) is attributed to the RF itself [9]. In addition to high contribution to total core power, RFs are also identified as one of the main hot-spots [12], [13], [14], [15]. The high power density (power per area) in an RF can cause severe reliability issues such as transient faults, violation of circuit timing constraints, and reduce the overall lifespan of the circuit [12]. Therefore, while reducing RF power is by itself beneficial for reducing overall core power, it also helps to mitigate RF temperature pressure, allowing embedded processors to work in their peak performance point.

Static power is an increasing contributor to overall power consumption. With every technology scale, power supply and threshold voltage need to be scaled down accordingly [16]. Simultaneously, a linear scaling in threshold voltage results in an exponential increase in leakage currents and static power consumption. The ratio of static to dynamic power of the OpenSPARC processor [5] has increased from 40% in 70nm technology to 70% in 45nm [7]. Similarly for register files, static power has a considerable contribution in total power dissipation [18], [19] making it even more important than dynamic power. RFs are composed out of flip-flops requiring more transistors than regular SRAMs [20], [4], have larger fan-out requiring larger transistors [21], which both aggravate static power dissipation. For example, in TI C64x cores the RF leakage power attributes to 60% of core leakage without cache and to 15% of full System-On-Chip leakage [21]. With RF’s high contribution to the overall static power consumption, systematic approaches are needed to reduce RF static power.

Embedded processors are often designed with large RFs, composed out of heterogeneous register banks with special-purpose registers [22], [23], [24]. A large heterogeneous RF helps embedded processors to efficiently execute a variety of applications from control to DSP and media [25]. However, large RFs also aggravate static power consumption of the core. The challenge emerges how to support large RFs while minimizing their impact to static power consumption. Nonetheless, registers utilization considerably changes between applications and even within an application [26], [27]. This provides a great potential to reduce RF static power by

1The static power ratio greatly varies for example with design process (low-power vs. high-performance), junction temperature and core activity. [17] reported 55% and 75% for a Blackfin processor (including cache hierarchy) for low and fast processes respectively.

Hamed Tabkhi and Gunar Schirner are with the Department of Electrical and Computer Engineering, Northeastern University, Boston, MA, 02115. E-mail: tabkhi@ece.neu.edu, schirner@ece.neu.edu
Manuscript received March 1, 2013; revised August 28 and November 1, 2013; accepted November 10, 2013.
turning-off registers not contributing to the program execution. However, 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 (which will be written to next). Therefore, an application guided power-gating approach is needed. Furthermore, an automatic solution for managing RF static power based on RF utilization is highly desirable to enable wide adaption.

In this paper, we propose Application-guided Function-level Register file Power-gating (AFReP), an automatic binary instrumentation for runtime RF power-gating. AFReP is an interplay of application binary analysis, Instruction-Set Architecture (ISA) and Micro-architecture extension. The core of AFReP is application binary analysis and instrumentation which determines non-optimized registers at function-level granularity. Then, it instruments the binary for run-time power-gating of unused registers. The instrumented binary can be executed on an AFReP-enhanced ISA with the necessary micro-architecture extensions to support selectively power-gating of registers and runtime power-gating configuration. AFReP uses static binary analysis and thus is independent from RF simulation profiling and input data.

We demonstrate the power saving potentials offered by AFReP by enhancing and analyzing a Blackfin Processor, a DSP-RISC processor by Analog Devices Inc. [22]. We analyze execution of a variety of signal processing (DSPstone) and control (Mibench) benchmarks. Our results demonstrate an RF static power reduction by 64% and 39% on average for control and DSP applications with a 1.2% and 0.6% increase in runtime. We further reduce AFReP’s overhead by optimizing the common case of recursive function calls. By introducing an instrumented interface function that in turn calls the recursive function, the performance overhead is minimized to 0.4% for control applications. Overall, AFReP shows impressive results for reducing RF static power consumption with negligible performance overhead.

This paper is organized as follows. Section II overviews previous work in RF power-gating. Background and the register lifetime are presented in Section III and Section IV, respectively. Section V discusses the granularity of RF power-gating in our AFReP approach. Section VI and Section VII explain AFReP architecture extension and application binary analysis, respectively. The experimental results are demonstrated in Section VIII. Finally, Section IX concludes this paper.

II. RELATED WORK

With the increasing importance of static power, more attention rests on power-gating of individual components inside processors cores. Some efforts focus on profiling-based power-gating of functional units (such as ALU or barrel shifter) during their idle periods [28], [29], [30]. Similarly, [31] proposes a data-flow analysis together with instruction re-scheduling for runtime power-gating of the core’s functional units. However, these approaches do not utilize the opportunities of involving the register file (RF).

Reducing RF power consumption is important because (a) the RF itself considerably contributes to the core power and (b) the RF is one main processor hot spot [12]. Most existing approaches have focused on RF dynamic power reduction: either by decreasing frequency of read/write accesses to RF [12], [13], proposing multi-banks RF composition [32] or by clock-gating registers [4]. While important, these approaches ignore RF static power and its considerable share to overall power dissipation.

AFReP is an application-guided architecture-level solution targeting in-order embedded processors. To put our work in context, the next paragraphs summarize underlying circuit-level approaches for power-gating. We then briefly review RF power-gating approaches for out-of-order cores. Following that, we discuss approaches for in-order cores in detail and compare them to our AFReP approach.

A. RF Power-Gating at Circuit-Level

At circuit-level, RF power-gating can be roughly divided into two categories; sleep mode and drowsy mode:

- **In sleep mode**, static power caused by leakage currents will be almost eliminated. However, register content is lost when turned off [3].
- **In drowsy mode**, static power will be partially reduced. At the same time, register is able to hold its content [33].

In drowsy mode, registers are backed up by state-retention latches in order to hold their contents. However, the retentive latches decrease leakage power saving and increase area overhead. Sleep mode in contrast achieves higher static power saving. However, it must be paired with register access analysis to avoid power cycling a register in case that its content is required in future [34], [33].

Recently a circuit technique, Supply Switching with Ground Collapse (SSGC) [20], offers faster switching delay and lower overhead in comparison to earlier approaches. Nonetheless being a circuit-level approach, [20] does not provide any solution to activate the proposed circuit. Our proposed AFReP approach could utilize SSGC or other proposed circuit-level techniques in an application-guided fashion [34], [33].

B. Out-of-Order Cores

Some approaches attack RF static power at the architecture/micro-architecture-level for out-of-order cores. [35], [5], [36] target high-performance out-of-order cores with large physical RFs and register renaming. [36] monitors the reorder buffer to identify and turn-off unused physical registers but has a limited visibility depending on the depth of reorder buffer. Similarly, [35] proposes early register release, turning-off unused physical registers and thus saving power by reducing the number of physical registers in operation mode. Both [35] and [36] are HW-only solutions and cannot guarantee absence of future read accesses before the next write. Therefore, they rely on the drowsy mode using state-retention logic with its associated remaining static power consumption.

C. In-Order Cores

For embedded processors static power mainly has been discussed in context of the memory hierarchy. However, as embedded processor are increasingly equipped with larger RFs,
their leakage is becoming more important. In this context, [5] introduces a HW-only approach reducing RF static power of in-order multi-cores processor. It monitors cache stalls and puts the correspondent core’s RF into drowsy mode.

Embedded processors often utilize in-order cores without register renaming in micro-architecture. The almost transparent mapping between ISA and physical registers offers an opportunity to utilize application analysis and instrumentation to efficiently reduce RF static power. To the best of our knowledge, [37], [7], [6], [21] are the most relevant approaches aiming to reduce RF static power of embedded processors with some application involvement.

Y. Zhou et. al [37] target ASIP design and proposes a post compiler analysis to detect unused registers of an entire program and subsequently excludes unused registers from processor synthesis. The approach does not include run-time configuration and imposes a performance penalty to other applications. X. Guan and Y. Fei in [7] splits a RF into hot and cold regions, where registers in the cold region are in drowsy mode. Based on simulation profiling, this approach maps less frequently accessed ISA registers to the cold region leading to overall RF power reduction. While being an interesting approach, [7] introduces three concerns. First, it assumes to freely map all registers and hence is not applicable for heterogeneous RFs with special-purpose registers. Secondly, [7] relies on simulation profiling to analyze register accesses which is slow and input specific. However, variations in input data can significantly change register activities [38] limiting the benefits of this approach. Finally, [7] shares the limitations of all drowsy mode approaches with the remaining static power.

J. L. Ayala et. al [6] aim to reduce RF static power for the most frequent executed code (subsequently referenced as kernel). Through manual binary instrumentation, the kernel’s unused registers are switched into drowsy mode. The instrumentation relies on analysis from simulation profiling. As such, [6] shares the limitations stated earlier for drowsy mode and simulation profiling. D. Atenzia et. al introduce in [21] propose an extension over [6] which uses a coarser granularity for power-gating of a complete register bank. Similar to [7], it requires re-mapping registers so that the kernel’s unused registers are mapped to the same bank and switched to drowsy mode. It also assumes that ISA registers are freely mappable. Overall, [21] suffers from the same drawbacks as [6] and [7].

Comparing earlier approaches ([6], [7], [21]) to our proposed AFReP reveals three main differences. AFReP (a) uses sleep mode to completely turn off registers and thus saves more power, (b) operates on a function-granularity covering the whole of program, and finally, (c) uses static binary analysis instead simulation profiling. AFReP provides a methodological approach that inherits the flexibility from its function-level granularity. AFReP uses analysis and automatic instrumentation of the application binary independent from input data. Consequently, AFReP avoids long simulation time and ambiguity where register access may differ with input data. Additionally, the power-gated registers can be completely turn-off, minimizing static power dissipation. Finally, power-gating can be applied to entire execution, not only most executed segment of program.

A complementary approach was introduced by [30], in which unused bits within an instruction are used for power-gating of functional units. While [30] differs significantly, targeting functional units instead of RF, their approach of encoding power-gating information can be considered in the context of our work. The Blackfin offers a variable instruction length (up to 64bits). It’s ISA is efficiently encoded and does not have enough unused bits to capture the power-gating configuration (see Section VI-B) in an existing instruction. Nonetheless, the original 32bit call instruction could be augmented with a 64bit version that includes power-gating configuration. This would eliminate the overhead of added LD1 instructions (Section VI-B), except when limited through the memory interface. Nonetheless, AFReP has already a very low execution time overhead in the sub percent range (Section VIII-C) which in most cases is practically negligible.

This paper extends over our initial publication [39]. In this paper we provide a more comprehensive description of the AFReP approach with significantly added details about design decisions and tradeoffs (e.g. with respect to granularity of power-gating) and provide a more in-depth analysis of the efficiency of the AFReP. On top of that, we introduce two extensions to reduce the overhead induced by AFReP, namely (1) AFReP-ScratchOnly focusing on non-call preserved registers, as well as (2) minimizing the overhead for recursive functions.

III. BACKGROUND

Before introducing the AFReP approach, we will first discuss using heterogeneous register files in embedded processors, and then introduce one example in particular, the Blackfin core which we will use as a case study for our experimental results.

A. Heterogeneous Register File

Designing processor with larger Register Files (RFs) is a common approach to increase core performance. A large RF can potentially reduces accesses to the memory hierarchy. The trends to increase RF’s size differ between out-of-order and embedded/in-order cores. In out-of-order designs, the physical RF can be extended for register renaming supporting speculative execution [40][36][35]. This increases the physical registers, while the ISA registers remain constant. Conversely, one trend of embedded processors is composing a large and complex logical RF out of heterogeneous registers with specialized functionality [41], [23], [24]. Heterogeneous registers are part of Instruction-Set Architecture (ISA) and thus are exposed to the programmer/compiler. Embedded processors such as ADI Blackfin families [41], TI C6x series [24], and Qualcomm hexagon [23] processors are designed with a large RF. In the TI C6x series, for example, the RF size has been doubled from thirty two 32-bit registers in C62x/C67x series to sixty four registers in C64x+ series with special-purpose registers.

B. Case Study ADI Blackfin

As a case study for this paper, we chose the Blackfin processor [41]. Blackfin is a DSP-RISC, designed to support both control and signal processing applications [25]. The Blackfin’s core includes two 16-bit integer multipliers, two
40-bit integer accumulators, two 40-bit integer arithmetic logic units (ALUs), four 8-bit ALUs, and a 40-bit barrel sifter. The Blackfin core performs in-order processing and has a variable instruction length (up to 64bits) with up to four parallel operations.

Blackfin has a 38 x 32bits ISA Register File (RF) composed out of eight data registers (#0-7), eight pointer registers (#8-15), sixteen circular addressing registers (#16-31), and also six zero-overhead-loop (hardware-loop) registers (#32-37). The circular-addressing registers are used for fast data memory addressing when a constant pattern of data array is accessed over and over again. The zero-overhead-loop registers define loop boundaries for the sequencer, which then can issue instructions in a loop, while avoiding pipeline stalls at the loop boundaries [41].

In this paper, we refer to both circular-addressing and zero-overhead-loop registers as special-purpose registers due to their specialization. Although special-purpose registers can be used for general purposes, they are mostly employed in data streaming and DSP applications. A similar RF organization with special-purpose heterogeneous registers is considered in other embedded processors such as Qualcomm hexagon and TI C6xx series [23], [24].

IV. REGISTER LIFETIME ANALYSIS

During program execution, register lifetime can be divided into two distinct periods: active and passive. In an active period the register is utilized, meaning that the register's content contributes to correct architectural state of program or in other words it contributes to program execution. In contrast, in a passive period, the register is not utilized, thus the register does not have any contribution to execution. An active periods starts by a write operation, and extends until its last read. Conversely, a passive periods starts from the last read and continues until the following write. Within that period the register content is irrelevant for program execution.

In this paper we will use register utilization as a metric of how long registers or register groups are utilized. The register utilization can be calculated by the duration of all active periods over the total duration of entire program’s execution. RF utilization is an average of individual register’s utilization during program execution. Further details for calculating RF utilization is provided in [27].

To estimate the RF power optimization potential, we investigate into register utilization over different application types. We chose six benchmarks from Mibench suite [42], and four benchmarks from DSPstone suite [43] representing control and DSP intensive operations. Fig. 1 shows the average utilization of different register categories of the Blackfin processor. The results are gathered from runtime profiling of both control and DSP benchmarks, compiled with gcc with -O3 optimization level. The profiling reveals that even for optimized applications many registers are not utilized, therefore their content does not contribute to the program execution. On average, data register utilization is 60% for DSP benchmarks and 44% for control benchmarks. This is particularly evident for control applications where the special-purpose registers including zero-overhead-loop and circular-addressing registers are seldom used (less then 10% utilization in control benchmarks). Non-utilization of special-purpose registers stems from two reasons. First, compilers may not detect code patterns to benefit from special-purposed registers in some applications. Therefore, many embedded programmers handcraft their applications in assembly to freely utilize all resources. Secondly, some applications may inherently not require all resources and they work with a limited set of registers.

For both control and DSP benchmarks, we observe a variation in RF utilization and also lower utilization of special-purpose registers. This provides an opportunity for RF static power saving by power-gating of registers during their passive periods.

V. GRANULARITY OF POWER-GATING

In a large set of applications, many registers have a low utilization (see Section IV), which happens particularly in control applications. Consequently, we propose Application-guided Function-level Register file Power-gating (AFReP). AFReP reduces RF static power significantly by automatically detecting register passive periods and then power-gating them.

Passive registers could be power-gated at application, function, loop, basic block, or even instruction levels. Application-level is the most coarse-grain whereas the instruction-level is the most finest granularity to manage the RF power-gating. There is a trade-off between the amount of RF power saving and the
configuration overhead, since at least one extra instruction is required for each configuration change. As granularity of power-gating moves from coarse application-level to fine instruction-level, more and smaller passive periods can be covered in cost of more frequent configuration changes of RF power-gating. An application granularity conversely leads to negligible additional instructions, but would be too coarse and cannot provide enough flexibility. On the other extreme, an instruction or basic block granularity could power-gate shorter passive periods but would be too fine, resulting in too many configuration changes and thus too much performance overhead. Therefore, a medium granularity is desired that balances power saving achievements and overhead due to inserted configuration instructions.

To guide our granularity selection, we will look into the length of passive periods. Fig. 3 shows the cumulative contribution of passive periods with reducing length to the overall RF unutilization. Fig. 3(a) focuses on control benchmarks and Fig. 3(b) on DSP benchmarks. Long periods of inactivity (>10,000 cycles) have a high contribution to the total passive periods especially in control benchmarks (from 47% for Basicmath to 85% for Quicksort). For Strsearch, passive periods of 8000 cycles and longer contribute to >90% of the total passive time of registers. In DSP benchmarks, the contribution of long passive periods is somewhat lower (from 25% for FIR2 to 49% for AES) but still shows a significant potential for power saving. For DSP benchmarks, a significant increase in contribution to total passive time is by very small periods of less than 10 cycles, which we however do not target due to the configuration overhead.

Sharp edges in both of benchmark groups demonstrate that passive registers increase at the boundary of some loops with many iteration or function calls. In general, long passive periods appear either in functions or in loops with many iterations. We have observed that longer periods (periods longer than 10000 instructions) significantly contribute to RF unutilization (70% on average for control benchmarks). For DSP benchmarks, the contribution of long passive periods is around 40% on average. Both loop and function-levels have reasonable levels of granularity to utilize passive periods for power saving. However, loop-level instrumentation poses additional complexities and overheads. 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. Our AFReP approach therefore operates on function-level granularity to manage RF power-gating.

For each function, our AFReP detects all potentially active registers within that function irrespective of a potential execution. Therefore, it provides an upper bound of active registers for that function. The remaining registers are candidates for static power saving (can be power-gated). To illustrate the differences, Fig. 4 compares the number of active registers at instruction-level (i.e. actual register usage) with our function-level estimation during the execution of FFT benchmark on Blackfin core. Function-level estimation provides an upper bound for number of registers that could be active within a function body. Therefore, the number of active registers at instruction-level is always below that in function-level estimation. The sharp edges in function-level estimations indicate function calls, when estimated RF activity is updated. In Fig. 4, the space above the function-level estimation shows the number of passive registers detected by our approach - demonstrating potential for power-gating. A significant power saving potential exists for FFT with at least 12 and up to 20 unused registers.

To quantify how much power saving potential is not realized due to our selection of function-level granularity, we measure the distance between the instruction-level and function-level estimation for a set of DSP and control benchmarks. Fig. 5 shows the average number of active registers both in instruction-level and function-level estimations. Register utilization as well as the difference between instruction- and function-level is application-specific. On average function-level estimation is
conservative by 5 registers for DSP applications, and 3 registers for control applications. Only FIR2 shows a larger difference with 15 registers. Overall, most passive registers can be detected by function-level estimation. The results confirm our selection of function-level granularity. It is able to express a large potential of passive registers while at the same time having an low anticipated performance overhead. In addition, the function-level granularity simplifies the analysis and instrumentation.

VI. AFRεP ARCHITECTURE EXTENSION

This section discusses RF micro-architecture and architecture extensions to enable runtime power-gating of individual registers. We first discuss AFRεP micro-architecture extension followed by the ISA extension to expose control to the software side. Lastly, we evaluate the effects of pipeline execution on the micro-architecture and ISA extensions.

A. Micro-Architecture Extension

Fig.6 shows a conceptual diagram of our proposed power-gating circuit. For simplicity, we limit the illustration to a four word RF (including R0, R1, SR0 and SR1). One possible implementation would be adding an additional PMOS transistor, called sleep transistor, to the power supply path. We add one additional register, named RFC (Register File Configuration), offering configurability for RF power-gating. We assume RFC is an ISA register which is exposed to programmer/compiler. The gate of each PMOS is driven by a single bit from the newly introduced RFC register, providing fine grain control over PMOS transistors. As a bit in the RFC is set to 0, the corresponding PMOS will be turned on, powering the corresponding register. Vice-versa, setting 1 in a bit of the RFC, powers off the corresponding register.

![Figure 6: Micro-architecture extension for Power-gated RF.](image)

Although adding the RFC itself will increase power, it provides the ability for runtime power-gating configuration which will ultimately lead to more power saving. The RFC length is equal to the number of registers in the RF.

B. ISA Extension

To utilize RF power-gating circuit, the running program has to control the newly added RFC. For this, the Instruction Set Architecture (ISA) has to be extended to support associated operation to the RFC. ISA needs to be extended in two ways: loading the RFC and integrating RFC into sub routine handling.

We extend Blackfin ISA with a new Load-Immediate Instruction (LDI) for loading an immediate value to the RFC. This bitmap value determines the set of registers to be turned-off. Fig. 7 shows an example for LDI using our four register word simplified architecture. Loading 1001b to the RFC will turn-off R0 and SR2, while R1 and SR0 are in operational mode (Fig. 7). Depending on length of RFC and target architecture, multiple LDI instructions are required, in case that the immediate section in the instruction is too short for capturing the whole bitmap. We consider the RFC to be a call-preserved register. Briefly, we see the RFC to be similar to the FP (Frame Pointer) register in respect to call preserve rules and function-call conventions. Therefore, the RFC needs to be stored into stack during a function call and restored from stack upon a return. In result, ISA also needs to support RFC stack operations.

![Figure 7: LDI for loading immediate value to RFC.](image)

To support RFC stack operations, the ISA can either be extended by new instructions, or the required operations can be embedded into existing instructions. Although the functionality would be same in both cases, the code size overhead is lower when embedding the RFC push/pop to currently available instructions. For our case study of the Blackfin core Link/Unlink instructions already push/pop both SP and FP registers. To avoid the overhead of additional operations, we expand Link/Unlink instructions to handle RFC stack operations as well.

C. Switching Delay and Energy Overhead

One challenge in RF power-gating is the additional delay due to powering registers on/off. At circuit-level, the delay depends on the characteristics of the sleep transistors and also on the power-gating circuit implementation. The timing requirements of the circuit-level are defined by the pipeline organization of the processor core.

Two potential critical paths exist for RF power-gating. First, a Write-after-Power-on; here a LDI instruction powers a register on, which is directly followed by a write instruction to the same register. If the register was not completely powered-on before the write operation, an error might occur. The second critical path is a Read-before-Power-off; when a register read instruction is followed by a LDI instruction. This can cause an error when the register is powered-off too early while the previous read operation has not finished yet.

In order to simplify the discussion of the two critical paths, we illustrate them on a simple five stage MIPS-like pipeline.
Fig. 8 illustrates both critical paths in one figure. First, a LDI instruction is turning-on R0 followed by a write instruction to R0. This illustrates the Write-after-Power-on case. Second, the same LDI, which turns-off R1, is followed by a read instruction from R1. This shows the Read-before-Power-off case.

In MIPS pipeline, as highlighted in Fig. 8, register write and direct/immediate load to register occur at WriteBack stage. For the typical power-gating circuit, power-on latency varies between one to four processor clock cycles [33]. Since the RFC is not subject to general RF accesses and also to relax the timing requirements, configuring the RFC by LDI can take place in the Execute stage. In result, the sleep transistor has three clock cycles to power-on R0 (in Fig. 8, from LDI instruction at Execute to the following write instruction at WriteBack). Effectively, the power-on latency can be masked by pipeline stages. In our case study of the Blackfin core, the distance between instruction decode and RF write back is six clock cycles. This additional time gives the circuit designer opportunity to minimize the overhead of the sleep transistor.

In MIPS-like pipeline, a register read occurs in the execute stage. Fig. 8 shows that while the read instruction from R1 is in the execute stage, the following LDI instruction is still in decode and thus has not yet powered-off R1. Therefore, Read-before-Power-off sequence would not disturb the functionality, since the register read occurs before the execute stage, assuming power-gating would be applied at execute.

VII. BINARY ANALYSIS AND INSTRUMENTATION

With the hardware extensions of Section VI in place, we present in this section our application binary analysis for detecting unused registers and binary instrumentation for inserting instructions to perform the run-time power-gating.

AFReP is based on a static approach for binary analysis and instrumentation, making it independent of simulation-based profiling. We chose static binary analysis as it yields worst-case register usage (most used) avoiding ambiguity where register access depends on input data. Our analysis operates as a post compiler stage, after compiler optimization and library linking. In this way, the analyzer achieves a comprehensive insight of register accesses in the program, including any 3rd party libraries that might only be available in binary form.

Fig. 9 shows the flow of our AFReP approach in two main stages: Binary Analysis and Power-Gating Instrumentation. The Binary Analysis is a composition of Function-Call Graph Extraction, Register Access Analysis and Dependency Analysis. At first, all callable functions are detected and a function call graph is created. In the next step, the Register Access Analysis parses the body code of each function to determine active and passive registers. Following that, the Dependency Analysis traces register dependencies between callee and caller to identify the register are required to be preserved (stored/restored on the stack). Finally, Power-Gating Instrumentation instruments application according to the gathered results from the binary analysis and creates a low power binary including the power-gating instructions.

Fig. 10 illustrates the steps of our flow with an example based on a simplified four word RF. The input, (Fig. 10 a) is a pseudo assembly generated out of flat application binary. Fig. 10 b) shows the function-call graph with identified active/passive registers within each function. Fig. 10 c) illustrates the dependency analysis. Finally, Fig. 10 d) shows the instrumented application assembly with RF power-gating instructions. In the following subsections, we will discuss the Fig. 9 in detail and move along with the example shown in Fig. 10.

A. Function-Call Graph

Our binary analysis and instrumentation operates on the assembly level. In this paper, we use objdump to generate assembly code from a flat Blackfin binary. Our Function-Call Graph Extraction parses the assembly, detects function-calls and extracts the corresponding call-graph. The root function is mapped to the Main function in C programming. Therefore, we ignore initialization functions such as the C-RunTime (CRT), which execute before Main.

As an example, Fig. 10 a) presents a pseudo assembly code. The example’s application binary comprises of only four callable functions: f0, f1, f2 and f3. Function f0 calls both f1 and f2, function f1 calls f3. Fig. 10 b) presents the output of Function-Call Graph Extraction which graphs the function call-chains of the application. In Fig. 10 b), we assume that f0 is the Main function.

B. Register Access Analysis

As discussed earlier, AFReP operates at function-level granularity. Therefore, AFReP is independent of execution trace and register access within each function body. Basically, AFReP assumes that each function has its own set of working registers (active registers). The working registers are those who are potentially accessed anywhere within the function’s body. As long as a register is referenced somewhere inside function body, regardless of runtime conditions, it is counted...
as a potential active register. Therefore, passive registers (unutilized registers) are limited to the registers which are out of function’s working register set. Although, this approach is somewhat conservative, it makes power-gating independent of input data and avoids simulation ambiguity. Registers which are not referenced thus passive/idle in function body, can be turned off and their power can be gated. By limiting the power-gating to the registers out of function’s working register set, AFReP guarantees un-altered functionality of the program independent of input data, and the path executed within the function body.

Fig. 10 b) also shows the output of Register Access Analysis, listing Active/Passive registers. For example, in f0, R0 and SR1 are declared passive since they are not referred inside of the f0’s body. Conversely, R1 is an active registers in f3.

C. Dependency Analysis

Power-gating of passive register has to adhere to the calling conventions defined in the Application Binary Interface (ABI). Calling conventions govern how function arguments are passed and how registers are accessed across a sequence of function calls. Function-call conventions roughly divide registers into three groups: dedicated, call-preserved and scratch registers.

The content of dedicated registers, e.g. FP (Frame Pointer) and SP (Stack Pointer), must be always valid for every function. We exclude dedicated registers from power-gating since they always have to be active. Instead, we focus on call-preserved and scratch registers.

The content of call-preserved registers need to be preserved across function calls. This means, their content has to be saved by callee before they are used, and restored back upon return from callee. In result, call-preserved registers appear unchanged to the caller. Typically, registers used for keeping global data and memory addresses and most of general-purpose registers belong to call-preserved registers.

The content of scratch registers is not preserved across function calls, meaning that their contents does not need to be saved and restored in a sequence of call and return instructions. Typically, a subset of general-purpose registers and most of special-purpose registers belong to the scratch category.

In Blackfin processor, our case study, registers (#4-7) and (#11-15) are call-preserved registers and the remaining which mostly belong to Circular-Addressing and Zero-Overhead-Loop registers are scratch registers. Also other architectures have a combination of call-preserved and scratch registers. To look at other architectures, most of the Vector Floating-Point (VFP) registers in ARM-based architectures belong to scratch category [44]. Similarly in TI C64xx/C32xx series, half of general-purpose registers are configurable to use as circular-addressing and belong to scratch register category [24].

Section VII-B showed how passive registers are detected for each function. Passive registers are good candidates for power-gating since their contents will not be used in current working function (Callee function). However, by power-gating of passive registers in the callee, there is a potential for data loss if their contents is already used by the caller function. This data loss is limited to the call-preserved registers (as the content of scratch registers is not guaranteed in the first place). To avoid this problem, the Dependency Analysis applies one rule: “the content of a passive call-preserved register in a callee that is active in at least one caller function has to be preserved”. In result, passive call-preserved registers in the callee will appear unchanged for the caller function.

Algorithm 1 represents the procedure of detecting registers required to be preserved. The Dependency Analysis uses the information from directional Function-Call Graph Extraction (see Section VII-A) and Register Access Analysis (see Section VII-B). Consequently, four lists are generated for every function:

- **Parent**: Direct callers of the function.
- **Active**: Active registers of the function.
- **Passive**: Passive registers of the function.
- **Stack**: Preserved registers of the function.

For each passive register of every functions, the algorithm checks all direct parents of that function. If a passive register is active at least in one of the parents and also the register belongs to the call-preserved category, its content has to be preserved. Then, the identified register is added to the stack list of the function. As each function maybe called from different callers, the callee has to preserve all passive call-preserved registers that used by any caller functions. Please note, each function only needs to keep track of its own direct parents (caller functions). If a parent further up in hierarchy power-

```
Algorithm 1 Procedure of detecting preserved registers
1: for F belongs to Functions do
2:   for R belongs to Passive(F) do
3:     if R belongs to Parents(F) do
4:       if R is Call-Preserved and R belongs to Active(P) then
5:         add R to Stack(F)
6:       end if
7:     end for
8:   end for
9: end for
```
gates an un-used call-preserved register, it in turn will add the necessary stack operations.

Relating back to our example in Fig.10, we assume that registers $R0$ and $R1$ are call-preserved and registers $SR0$ and $SR1$ are scratch registers. Fig.10 c) presents the result of dependency analysis. Since, $R1$ is a passive register in function $f1$ while it is active in function $f0$, its content should be preserved in $f1$. In contrast, there is no need to preserve the content of $R0$ in $f1$ because it is a passive register in caller function $f0$. In function $f2$, although $SR0$ is active in caller function $f0$, it does not need to be preserved since $SR0$ is a scratch register.

The additional stack operation for pushing/popping call-preserved registers causes some overhead. One approach to avoid this overhead is to ignore call-preserved registers in power-gating and only focus on scratch registers (we hence refer to this option as AFReP-ScratchOnly). However, limiting the scope of power-gating will lead to less overall RF static power saving. Section VIII-D presents a cost and benefits analysis for AFReP-ScratchOnly.

### D. Power-Gating Instrumentation

The previous steps have identified passive registers for power-gating and those for stack preservation. The next step is to instrument the analyzed data to the application binary for the purpose of actual RF power-gating. As illustrated in the AFReP flow (Section VII), we developed the Power-Gating Instrumentation tool to instrument application binary to manage RF power-gating at function-level granularity. The Power-Gating Instrumentation tool inserts (a) LDI for configuring RF power-gating, and (b) extra stack operations for identified call-preserve registers.

Fig.11 shows a function caller/callee dependency: $f()$ is the caller function which calls $g()$ being the callee. Both $f()$ and $g()$ functions have their own sets of passive registers. Power-gating of passive registers is controlled by inserting LDI instructions in the prologue of each function (see Fig.11). LDI loads an intermediate value to the RFC register, determining the registers for power-gating. Before updating the RFC, however, an extra push is required in order to preserve the caller’s RFC. In the epilogue also, before returning to the caller, the caller’s RFC is popped, restoring the previous configuration. In that, the RFC follows call preserve semantics (like FP). As a function gets called in run-time, the instrumented prologue is executed which configures for the functions power-gating state. Reaching an instrumented epilogue restores the previous power-gating state.

Inserting instructions may require updating jump addresses in the application binary. (Note, functions with multiple returns have multiple epilogues). We employ a three phase mechanism for updating target addresses of affected branch/jump instructions. In a first stage, all jumps/branches, including both absolute jumps, relative and indirect branches, are identified building a linear list of branch/jump sources and their potential destinations. Since new instructions are only added into 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. In second stage, the based addresses of branches and absolute jump targets are updated according to number of instrumented instructions in prologue of current function and prologues/epilogues of all prior functions in the code section of program. In a third pass, the index addresses for relative branches are also updated in cases of multiple returns or indirect jumps within multiple functions in case that relative addresses are affected.

The same method can be expanded for recursive function calls when both caller and callee are same. For dynamic function calls, the address of callee is determined at runtime. Here, the caller has to preserve all active call-preserved and scratch registers to prevent from any data loss. In case of interrupt or any exception, the power-gating should be disabled for the ISR (Interrupt Service Routine).

Fig. 10 d) of our running example presents the instrumented binary. The LDI instruction is inserted in prologue of every functions. Also, extra push/pop instructions are inserted in epilogue/prologue of $f2$, to preserve the register content. For instance, inserting LDI 1001 means that the passive registers $R0$ and $SR1$ are power-gated during the execution of $f0$. Similarly, inserting LDI 1101 in prologue of $f3$ shows that only $R1$ should be in operation mode and the remaining registers are power-gated. In addition, the Push and Pop instructions in $f2$ present preserve operations for register $R1$.

### E. Recursive Function-Calls

Instrumenting the binary at function-level granularity leads to some performance overhead depending on the frequency of function calls and the number of registers need to be preserved. This performance overhead can be more pronounced in recursive function-calls when a single function continuously calls itself. In this case, caller and callee are identical and thus have an identical RFC configuration. LDI instructions updating the RFC will not have any effect since re-writing identical values, only introducing extra overhead.

To reduce the performance overhead, the additional LDI/stack instructions appearing in the recursive function calls need to be eliminated. To achieve this, we propose an interface function between the parent caller and the recursive callee function. Therefore, instead of calling the recursive function directly, the parent function calls the interface function which is instrumented according to the register activity of recursive callee function. Consequently, the instrumented interface function only calls the unmodified recursive callee while the power-gating configuration have been already set. In result, when the recursive function calls itself again, no additional power-gating instructions and stack operations are executed.

![Figure 11: Function-call support in AFReP.](image-url)
Fig. 12 illustrates the insertion of an interface function for a recursive call. Function \( f() \) is the parent function and \( g() \) is the recursive callee function. This recursive pattern can be statically identified by parsing the body code of function and identifying a call instruction that point to its own function. After detecting recursive-call patterns, the instrumentation tool inserts the additional function, \( g'(\cdot) \), as an interface between \( f() \) and \( g() \). Epilogue and prologue of \( g'(\cdot) \) are instrumented with LDI and stack operations in respect to \( g() \)‘s register activities. In its body, \( g'(\cdot) \) calls \( g() \). In result, \( g() \) holds same identical RFC configuration as \( g'(\cdot) \) function, without additional LDI instruction. On recursive functions, the overhead is significantly reduced by introducing an instrumented interface function and keeping the recursive function un-instrumented.

VIII. EXPERIMENTAL RESULTS

In this section, we first explain the simulation setup for evaluating our AFReP approach. Then, we focus on RF static power-reduction in AFReP and compare it with most closing approach (Kernel-Based [6]). Following that, we investigate the performance overhead associated with AFReP without special handling of recursive functions. We then evaluate the effect of interface function in reducing the performance overhead of recursive function-calls. Finally, we will consider an option of AFReP, "AFReP-ScratchOnly", which operates on a limited set of registers.

We target the Blackfin core [41], a DSP/RISC-like processor with an ISA and micro-architecture that is tailored for both control and DSP applications [25]. In full operating mode, Blackfin core runs at 600MHz with 1.2 Vdd. Blackfin has a 38 x 32bits ISA register file. The baseline architecture configuration of Blackfin’s has been provided in Section III-B. For our experiments, we developed an Instruction-Set Simulator (ISS) of the Blackfin core based on the Trap-ADL [45]. Both Blackfin ISS and the ISA have been extended to support the AFReP-enhanced micro-architecture with the RFC register (Section VI-A), and also the LDI instruction (Section VI-B) as well as expanded Link/Unlink instructions.

In our implementation, the RFC has a length of 34bits, even though the core consists of 38 registers. Two groups of the hardware-loop registers can be power-gated with one bit per group as either all registers in a group are used, or none. Two 32bit LDI instructions are required for loading a bitmap value into RFC. We use an unmodified gcc compiler (gcc), since AFReP is based on the binary-level analysis. As described in Section VII, AFReP operates as post-compiler stage after all compiler optimization and library linking. We use gcc (ld) tools to read and operate on the binary. By using obidamp tool, we can generate assembly code out of binary and then feed it into our AFReP toolchain. We select benchmarks from MiBench [42] and DSPstone [43] as real workloads for control and signal processing applications, respectively. The benchmarks are compiled with gcc (-O3) so that the compiler optimization can find opportunities to best utilize registers.

To evaluate the AFReP energy overhead we focus on its two main contributors: the newly introduced RFC register and the additional sleep transistors. We establish an analytical power model based on the numbers reported by [7] for 70nm technology (AFReP is scalable to any technology).

The static power of the RFC register is equal to the static power of a 34-bit register. Given the Blackfin’s RF, adding the RF static power increases the total RF static power consumption by less than 3%. The power of the sleep transistors depends on the transistor feature size, which in turn depends on the time pressure of power-gating. The pipeline of the Blackfin processor offers a large slack time of six cycles for the off/on transition (see Section VI-C). This allows for a small transistor feature size and thus small power impact. The overhead of the sleep transistors in comparison to the RFC overhead is negligible. Nonetheless, following static power consumption reported in [34], we assume that even after full power-off, only 95% of the leakage power can be eliminated. The remaining 5% (worst case) are attributed to sleep transistors and residual static power. Realistic benefits of AFReP may be even better, for example [47] demonstrates that sleep mode power-gating reduces leakage by more than two orders of magnitude.

To reflect the impact of our proposed micro-architecture extensions, we have annotated the power overhead of RFC and sleep transistors (in accordance to [34], [7]) into our AFReP-enhanced ISS.

A. RF Static Power Reduction in AFReP

To evaluate the amount of power savings with our AFReP approach, we compare AFReP with the most closest related work approach [6]. The details of [6] have been already discussed in Section II-C. In a nutshell, [6] applies drowsy state to power-gate unused registers of the most executed segment of code. We refer to the work in [6] as Kernel-Based for the subsequent discussion. For realizing the Kernel-Based idea of [6], we manually profiled the application to determine the function contributing the most to execution time (kernel). Then, we manually added the LDI instructions only to the prologue and epilogue of the kernel function. In the drowsy state of the kernel-based approach, the power-gated registers are in state-retention mode where the register can hold its content with cost of additional power overhead. [7] shows that the static power in drowsy state is reduced to 20% of a register in operational state.

Fig. 13 shows the RF static power reduction of AFReP and Kernel-Based approaches. Results are obtained by executing the

---

2 Note, [46] evaluated the total power ratio including core and cache. We assume the same ratio to hold true for the static power of the RF.

3 Other modeling tools such as Wattch could be used estimate the power overhead of our proposed AFReP micro architecture extensions. However, since Wattch is targeted for SimpleScalar it would require to first develop a Blackfin model within SimpleScalar, which is impractical for the purpose of this work.
benchmarks on our power-annotated Blackfin ISS. 0% savings is equivalent to execution on the base-line processor without any extension. In addition to comparing AFReP and Kernel-Based approaches, Fig.13 also shows the power saving for AFReP-ScratchOnly, which we will discuss later in Section VIII-D.

Fig. 13 demonstrates that AFReP reduces the RF static power by 64% for control applications, on average. AFReP achieves a 39% RF static power reduction for DSP benchmarks. For all benchmarks some power savings were achieved. In other words, the benefit of power gating outweighs the added overhead of our micro-architectural extensions. We observe that the amount of RF static power saving varies across different benchmarks based on their RF utilization. The minimal observed reduction is 7% for FIR from the DSP section. AFReP saves up to 70% of RF static power of Quicksort, StrSearch and BitCnt from control benchmarks.

In comparison, the Kernel-Based approach achieves much less power reduction. On average, it reduces RF static power by 38% and 20% for control and DSP benchmarks. For some benchmarks such as Matrix, BitCnt and CRC, both AFReP and Kernel-Based yield almost similar RF power savings. At the same time, other benchmarks significantly benefit more from AFReP. For the benchmarks such as FFT, Wavelet and BasicMath, AFReP achieves much higher RF power saving (more than three times) than Kernel-Based approach.

To analyze the underlying reasons for the variances in power-saving, we captured, each function’s contribution to total execution time with blue bars in Fig. 14. In Fig. 14, we list the Kernel, Second function and Third function and Others, ordered based on the contribution to execution time. Also the share of other functions to execution time presented separately with name of Others. The sum of all other functions with lower contributions is presented as Others. Left of the execution time contribution, Fig. 14 shows the contribution of these functions to the overall RF static power saving with green bars.

Fig. 14 shows that in few benchmarks, including Matrix, BitCnt and CRC, only a single function is dominating the total execution. In result, AFReP cannot improve the power-savings by instrumenting functions that almost do not contribute to execution time in comparison to Kernel-Based approach. In these cases it only improves power savings by using sleep mode instead of drowsy. In contrast, in most of the benchmarks, such as Wavelet, AES, FFT and MDRall, more than one function significantly contributes to program execution. Since AFReP automatically instrument all functions, it leads to much more power saving than [6] – twice RF power saving on average.

It is interesting to note that in benchmarks with a wider spread of functions to execution time, the kernel (most contributing function) has fewer passive registers, leading to less of a power saving opportunity. This puts the Kernel-Based approach [6] at a further disadvantage. For example, in Wavelet, only 14 registers can be power-gated in Kernel while average of passive registers for Others is 25 registers. While in Wavelet the kernel function attributes to 47% of the total execution time, it only contributes to 33% of the power savings.

B. AFReP Overheads

The advantages of power-savings come at a low cost of some performance overhead, code size increase, as well as a minimal area increase. Fig.15 shows the performance overhead of AFReP due to additional instructions for RFC initialization (LDI instruction) and stack operations for preserving the necessary call-preserved registers. We measure performance overhead by instrumenting functions that almost do not contribute to execution time in comparison to Kernel-Based approach. In these cases it only improves power savings by using sleep mode instead of drowsy. In contrast, in most of the benchmarks, such as Wavelet, AES, FFT and MDRall, more than one function significantly contributes to program execution. Since AFReP automatically instrument all functions, it leads to much more power saving than [6] – twice RF power saving on average.
overhead through execution on Blackfin ISS both for original and instrumented (AFReP-enhanced) binary. The performance overheads shown in Fig.15 vary considerably depending on the granularity of functions and the frequency of function calls. On average, we observe a performance overhead of 1.2% for the control benchmarks and 0.6% for DSP benchmarks. Frequently called small functions like in FFT and MDRall, or recursively called such as in Quicksort and StrSearch, yield a measurable performance overhead. In contrast, CRC, BitCnt and Matrix with few function calls shows practically no overhead.

To examine the reasons for variations in performance overhead, we analyze the share of functions to the performance overhead. For benchmarks with a measurable overhead, Fig. 16 shows the contribution of functions to performance overhead (right bar in red color) in comparison to the function’s power saving contribution (left bar in green). In benchmarks such as Wavelt, FFT and MDRall, the most contributing function to power saving has much less contribution to the performance overhead. In this benchmarks, functions with much fewer contribution on power saving (Others) are highly contributing to the performance overhead. In contrast, in some benchmarks (QuickSort and StrSearch) a single function dominates both in power saving and performance overhead. We analyzed these benchmarks and found that their Kernel is a recursive function call. This motivated us to optimize recursive function calls which the solution has been proposed in Section VII-E and the results are described in the next section.

Table I: AFReP code-size overhead

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>DSP</th>
<th>Wavelt</th>
<th>Matrix</th>
<th>FFT</th>
<th>FIR2</th>
<th>AES</th>
<th>Jpeg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead(%)</td>
<td>1.8</td>
<td>6.2</td>
<td>1.6</td>
<td>4.7</td>
<td>2.0</td>
<td>2.6</td>
<td></td>
</tr>
<tr>
<td>Control</td>
<td>StrSearch</td>
<td>BitCnt</td>
<td>MDRall</td>
<td>QuickSort</td>
<td>BasicMath</td>
<td>CRC</td>
<td></td>
</tr>
<tr>
<td>Overhead(%)</td>
<td>2.5</td>
<td>4.9</td>
<td>2.0</td>
<td>4.1</td>
<td>1.5</td>
<td>4.4</td>
<td></td>
</tr>
</tbody>
</table>

Table I quantifies the code size overhead due to the added power-gating instructions. We only consider the text section which is the only modified section. Both DSP and control benchmarks exhibit a low increase in text size (3.1% and 3.2% respectively). Benchmarks consisting of many small functions (e.g. Matrix) grow slightly more in text size. Overall, the added cost due to code size increase is minimal and the benefit of reducing static power clearly dominates.

Finally, AFReP introduces some minimal area overhead.

The additional area within the RF stems from the added RFC register and added power gating transistors. In case of Blackfin, the RFC register increases RF capacity by 2.8% and should similarly increase area. Using transistors count as an indication for area increase, the added power-gating transistors have even smaller impact as they count for less than 0.5% of the RF transistors (assuming 6 transistors per bit). Overall, the increased area within the RF is dwarfed by the overall core area. When comparing to [6], AFReP introduces much smaller area overhead. Both approaches require additional register to control power-gating. However, AFReP uses power-off power-gating and thus is not subject to the area increase due to state-retention latches, which are required by [6].

C. Recursive Function-Calls

For benchmarks with recursive calls we have seen that they have (a) a higher performance overhead which (b) is dominated by the recursive function itself. As a re-configuration of power-gating is not necessary between recursive calls, this offers opportunity for reducing the overhead. Recursive function-calls can be easily detected statically by analyzing application binary. The performance overhead can be avoided by inserting an interface function between the caller and recursive callee, described in Section VII-E.

![Figure 17: Performance overhead before and after detecting recursive function calls.](image)

Fig.17 illustrates the performance overhead of Quicksort and StrSearch before and after inserting interface function. Clearly, a considerable reduction can be achieved by the interface function. For Quicksort the performance overhead decreases from 1.2% down to 0.2% and for StrSearch the overhead is almost eliminated (to less than 0.05%). Fig.17 also presents the average performance overhead for all control benchmarks without and with optimization for recursive functions. By reducing the overhead of recursive function-calls, the average performance overhead of all control benchmarks reduces from 1.2% down to 0.4%.

D. AFReP ScratchOnly

As a further optimization option we look at AFReP being applied only to scratch registers. The performance overhead is caused by LDI instruction to configure RF power-gating and by additional stack push/pop instructions to preserve the content of some call-preserved registers. By excluding the call-preserved registers from RF power-gating, additional stack operations

4Note that also the control logic for decoding instructions increases minimally. However, reliably estimating the area requires a more detailed RTL design, which is outside of the scope of this work.
can be avoided reducing the AFReP performance overhead. However this limits the power saving potential.

AFReP-ScratchOnly is a reduced version of AFReP which only focuses on scratch registers. Fig.13 and Fig.15 also present the RF power saving and the performance overhead of AFReP-ScratchOnly approach. By focusing on scratch registers, AFReP-ScratchOnly achieves a slightly lower power savings of 55% (vs. 64%) for control and 33% (vs. 39%) for DSP applications. Still, AFReP-ScratchOnly achieves higher power savings than the kernel-based approach. The benefits of AFReP-ScratchOnly become apparent when looking at the performance overhead. It is much lower than when considering all registers with 0.3% (vs. 0.4%) for control and 0.25% (vs. 0.6%) for DSP applications.

Overall, AFReP-ScratchOnly reduces the already very low performance overhead further, however at the loss of less power saving. With this, AFReP-ScratchOnly (in contrast to AFReP), has to be carefully explored before applying it to a processor.

IX. CONCLUSIONS

In this article, we introduced AFReP: an Application-guided Function-level Registerfile Power-gating approach to mitigate the power consumption of register files. AFReP utilizes the fact that in many applications not all registers are utilized all the time, while still dissipating static power. Our approach performs a static binary analysis to determine register lifetimes on a function-level granularity. AFReP then automatically instruments the code to configure a register-power-gating circuit during execution time. Our AFReP approach is supported by a micro-architecture extension for power-gating individual registers, and an ISA extension to enable the run-time control.

We applied AFReP to a Blackfin processor, extending the ISA and adding power-gating to its 38 registers. We validated the effectiveness using MiBench and DSPBench benchmarks, executing on an extended ISS. The simulation results demonstrate an RF static power reduction by 64% and 39% on average for control and DSP applications. Performance overhead is very low with 0.4% and 0.6% for control and DSP benchmarks. AFReP helps embedded cores to get larger RFs to achieve higher performance and with less concern about static power overhead. With future embedded processors using even larger register files and being built with smaller feature sizes, the impact of AFReP will even further increase.

REFERENCES


[22] Analog Devices Inc. (ADI), ADSP-BF52x Blackfin Processor Hardware Reference (Volume 1 of 2).


