AFReP: Application-guided Function-level Registerfile Power-gating for Embedded Processors

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

ABSTRACT

With shrinking CMOS feature size, static power is growing significantly and power density has emerged as an increasing concern. At the same time, one trend of embedded processors is toward larger Register Files (RFs) which further increases static power dissipation and aggravating the issue. This paper introduces an Application-guided Function-level Register file Power-gating (AFReP) that reduces static power of RFs in embedded processors. Our AFReP approach is based on a automatic analysis of register lifetime in the application binary, followed an automatic binary instrumentation for runtime RF power-gating. The instrumented code executes on a processor with ISA and micro-architecture extension for power-gating control over individual registers. Our application binary analysis/instrumentation operates at function-level granularity, automatically gating the registers that do not contribute to program outcome. Our experimental results using an AFReP-enhanced Blackfin processor demonstrate average RF static power reduction by 60% and 52% for control and DSP applications from Mibench and DSPstone suites, respectively. The added instructions for run-time power-gating increase execution time by only 1% on average.

1. INTRODUCTION

In the recent years, embedded processors are designed with larger register files to reduce the number of references to memory thus increasing performance. High-performance processors, on one hand, use extended physical RFs for supporting speculative execution and register renaming [11][19]. For embedded processors, on the other hand, one trend is composing a larger RF out of heterogeneous registers with specialized functionality. Embedded cores such as TI C64x+ [2], ADI Blackfin [1] and Qualcomm Hexagon [3] are design with large RF. In the TI series for example, the RF size has doubled from thirty two 32-bit registers in C62x/C67x series to sixty four registers in C64x+. Large RFs aid embedded processors to support application variety from control to media applications [12], however, they impose additional power overhead.

RF significantly contributes to processor's power. The power breakdowns of embedded processors reported in [5][8] show that the Register File (RF) consumes 15%-36% of overall power of the processor. The results differ based on the size of RF, processor complexity, and running application. As an example, 20% of processor core power (including data and instruction caches) in Blackfin processor is attributed to the RF itself in 120nm technology [15].

With shrinking transistor size and consequently increasing the leakage currents, the static power increases in contribution to RF power dissipation, making static power even more important than dynamic power. While ratio of RF static to dynamic power was 40% in 70nm technology [8], it increased to 70% in 45nm according to the reports for OpenSPARC processor [18]. Additionally, RFs are composed out of flip-flops requiring more transistors than regular SRAMs [13][14], as well as larger fanout demands drive the need for large transistors [4], which both aggravate static power dissipation. [4] shows for the TI C64 that the leakage energy of RF is about 60% of core without cache memory and 15% of full system on chip.

In addition to significantly contributing to power consumption, RFs are also one of the main hot spots [17][22]. 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 the life span of the circuit [17]. Therefore, RF static power reduction not only is beneficial for additional energy saving, but is also is necessary for mitigating RF temperature pressure, allowing embedded processors to work in highest possible performance.

We have observed in experiments (Section 3), that RF utilization considerably changes in different applications and within a application. There is a potential for RF static power saving by turning-off registers that do not contribute to the program execution. However, the executing 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 instrumenting the code for power-gating is highly desirable to enable wide adaption.

In this paper, we propose an automatic binary instrumentation for runtime RF power-gating, called Application-guided Function-level Register file power-gating (AFReP). Our AFReP approach is based on static application binary analysis that determines non-utilized registers for each function. It then automatically instruments the binary for run-time power-gating of unused registers. The instrumented binary can be executed on an AFReP-enhanced ISA and with the necessary micro-architecture extensions. Our simulation results on an AFReP-enhanced Blackfin processor demonstrate average RF static power reduction by 60% and 52% on average for control and DSP applications from Mibench and DSPstone suits with negligible performance overhead.

The remainder of this paper is organized as follows. Section 2 will discuss previous work. The register lifetime analysis will be

IEEE/ACM International Conference on Computer-Aided Design (ICCAD) 2012, November 5-8, 2012, San Jose, California, USA

presented in Section 3. Section 4 explains our proposed AFReP approach. The implementation and simulation results will be presented in Section 5. Finally, Section 6 concludes this paper.

2. RELATED WORK

Reducing the RF power is important having a considerable share of processor power, and more importantly, because it is one of the main hot spots [17]. Many researches have focused on reducing RF dynamic power consumption by decreasing the frequency of access to RF [17, 22] or partial clock-gating RF [14]. However all of the above did not address the RF static power and its crucial involvement in total power dissipation.

Recently, some approaches address RF static power in high-performance processors where a large physical RF are used for supporting register renaming [11, 18, 19]. [18] monitors cache stalls to put RF in a low power state, called "*drowsy state*". In the *drowsy state*, static power dissipation is reduced at cost of increased latency, while the RF still keeps its current state. [19] proposed a hardware approach for monitoring the reorder buffer and putting the potential unused physical registers to drowsy state. In contrast, [11] proposes early register releasing to reduce the number of physical registers by completely turning off the unused physical registers.

In embedded processors, static power has been mainly discussed at higher level of the memory hierarchy – specially on caches. Fewer approaches have discussed RF static power. As embedded processor have started to get larger RF, its leakage is becoming more important. Recently, [13] applied a circuit technique, called supply switching with ground collapse (SSGC), to the register file. The proposed circuit overcomes some limitation of traditional power-gating circuit, however, [13] did not provide any solution to activate the proposed circuit. Our proposed AFReP approach can utilize these circuits in an application-guided fashion.

To the best of our knowledge, [8, 5] and [4] are the only approaches attempting to reduce RF static power of embedded processors with some application involvement. [8] splits the physical RF into hot and cold regions, where in cold regions registers are in drowsy state. Based on simulation profiling, this approach maps less frequently accessed ISA registers to the cold region leading to overall RF power reduction. There are three main shortcoming in [8]. First, it assumes to freely map all registers – hence it is not applicable for heterogeneous special purpose registers. Secondly, the RF access information are gathered from simulation profiling which is slow and input specific. Investigation shows that variation in input data can change register activities significantly [20]. Finally, registers in *drowsy state* hold their contents, therefor leakage current is not completely eliminated. This leads to less power saving with larger area overhead.

A different approach has been proposed in [5]. Similar to [8], [5] suffers from drawbacks imposed by drowsy state and simulation profiling. This approach tries to reduce RF static power for the most frequent executed segment of program (subsequently referenced as *kernel*). The kernel's unused registers are switched into drowsy state. This approach is based on manual instrumentation which relies on simulation profiling. [4] is an extension over [5], where the control over power-gating is in register bank granularity. All kernel's unused registers would be mapped in [4] to the same register bank and switched to drowsy state. It again makes the assumption of ISA registers being freely mappable. In addition, [4] suffers from the same drawbacks of [5] and [8].

In contrast, our AFReP approach proposes a static analysis and automatic instrumentation over application binary where it is independent from input data. Consequently, it avoids long simulation time and simulation 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.

3. REGISTER LIFETIME ANALYSIS

During program execution, register lifetime can be divided into two distinct periods, active and passive. In an active period the register's content contributes to correct architectural state of program. In contrast, in a passive period, the register does not have any contribution to correct 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. A register can have multiple passive and active periods over time. Register utilization can be calculated by dividing the length of all active periods over the entire execution. Fig. 2 highlights register lifetime for *R7* during execution of nine instructions.

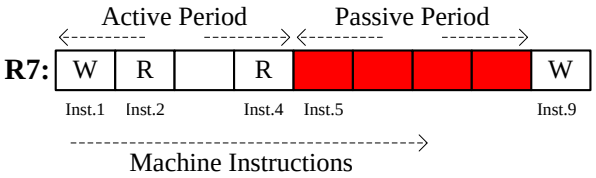

Figure 1: Passive and Active periods.

As a case study for this paper, we chose the Blackfin processor designed for supporting of both control and signal processing applications [12]. Blackfin has a *38 x 32bits* ISA register file composed out of eight data registers *(#0-7)*, eight pointer registers *(#8-15)*, and sixteen circular addressing registers *(#16-31)*, and also six zero-hardware-loop registers *(#32-37)*. Circular-addressing registers, and zero-hardware-loop registers are counted as special-purpose registers where they can be used for fast and continuous data memory addressing or continuous execution of instruction loops reducing pipeline stalls. A similar RF organization is considered in other embedded processors such as TI C64x series [2]. Although special-purpose registers can be used for general purposes, they are mostly employed in data streaming and DSP applications.

Fig.2 shows the average utilization of different registers in the Blackfin's RF for two classes of applications. The results are gathered from runtime profiling of six and four optimized *(gcc-O3-compiled)* benchmarks from Mibench and DSPStone suites [10][23]. The profiling highlights that even for highly optimized applications many registers are unutilized. This is particularly evident for control applications where the special-purpose registers are seldom used. 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 and they work with a limited set of registers. Fig. 3 shows the average contribution of passive periods with increasing length over the sum of all passive periods, for the same benchmark. The results demonstrate that a considerable portion of passive periods (65%) belongs to the periods with long duration (>8000 instructions).

The above observations show that there is an opportunity for RF static power saving through gating the power of registers during their passive periods.

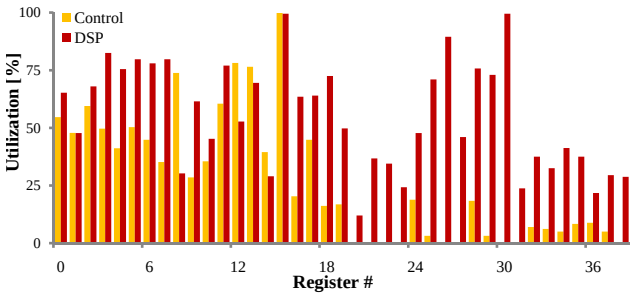

Figure 2: Average register utilization in Blackfin's RF.

4. APPROACH

In a large set of applications, many registers have a low utilization. This happens particularly in control application when there is not too much potential for taking the benefits of all registers. Consequently, we propose Application-guided Function-level Register file Power-gating (AFReP). AFReP can reduce the RF static power significantly by automatically detecting register passive periods and thus power-gating them.

The heart of AFReP is an automatic binary analysis and instrumentation. Binary analysis detects the register's passive periods and applies power-gating for them by instrumenting the original binary. For this purpose, the RF micro-architecture needs to be extended to allow selectively turning-off RF words. Additionally, an enhancement in the Instruction Set Architecture (ISA) is needed as a communication bridge between the program and the extended RF. The remaining of this section discusses these components individually.

4.1 Granularity of Power-Gating

Theoretically, passive registers can be power-gated at application, function, loop, or even basic block levels. There is a trade-off between the RF power saving and the configuration overhead. An application granularity would be too coarse and cannot provide enough flexibility. On the other extreme, a basic block granularity would be too fine, resulting in too many configuration changes. Our profiling results (see Fig.3) show the considerable portion of passive periods with higher than *8000* instructions long (*65%*), appeared either in functions or in loops with many iterations. Although, loop and function-level have reasonable levels of granularity, loop-level poses additional complexities. Detecting the 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

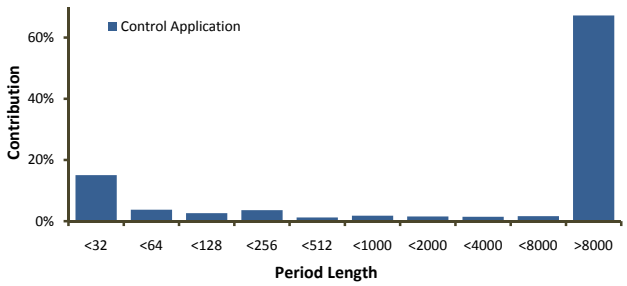

Figure 3: Contribution to overall passive time over increasing length of passive period.

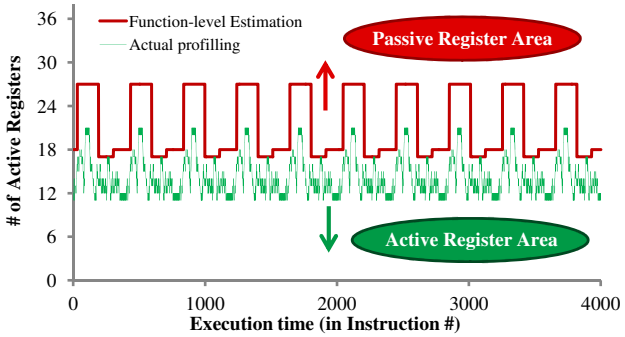

Figure 4: Actual and estimated RF activity for Dhrystone benchmark.

operates on function-level granularity, which is used for detecting passive periods.

For each function, our AFReP detects all potentially active registers within that function irrespective of the call path. Therefore, it provides an upper bound of active registers for that function and the remaining registers can safely be turned-off (power-gated). Fig. 4 shows the RF activity for the Dhrystone benchmark. It shows both, the actual used registers (determined through lifetime analysis of executed code) in green and the upper-bound detected by our AFReP in red. The edges in the red line are function calls. With each function call, the estimated bound for active register is updated depending on number of active registers in the called function. Please note that, the actual register activity is always below the estimated activity. For each function, the space below the green line presents the number of active registers (which can not be turned off at any time). The space above the red line shows the corresponding number of passive registers – the potential for power-gating. During this section of Drystone execution, a significant saving potential exist turning off up to 20 registers.

4.2 Micro-Architecture support

Our AFReP approach relies on a power-gating circuit in the RF micro-architecture. A straight-forward implementation adds an extra transistor, called sleep transistor, into power supply path for each register. The sleep transistor provides control over power-gating operation. In this way, the leakage currents in the "turned-off" state of the power gated register is almost eliminated. However, the register will lose its content.

An state-retentive alternative power-gating uses a "drowsy" state [6], where the register keeps its content by adding retentive latches to all individual bits of the registers. The state-retentive properties make this approach attractive [18, 19, 8, 5, 4] for reducing RF static power, as the surrounding approach does not need to detect the usefulness of registers' content. Thus, no register life time analysis is needed. However, the retentive latches for the "drowsy" approach reduces the leakage power saving potential and increases area overhead in comparison to [21][16].

In contrast, our AFReP approach can detect the passive periods of each register through the binary analysis. In result, AFReP takes the benefit of complete power-gating method to send the passive register to Turned-off state leading to more power saving. Our AFReP approach can be applied to different power-gating circuit-level approaches, such as [13] as well.

Fig.5 shows a simple example of power-gating circuit for a four words RF. PMOS transistors (sleep transistors) are added between each register word and supply voltage. The gate of each PMOS is driven by a single bit from a new register called *RFC (Register File Configuration)*, providing fine grain control over PMOS transistors.

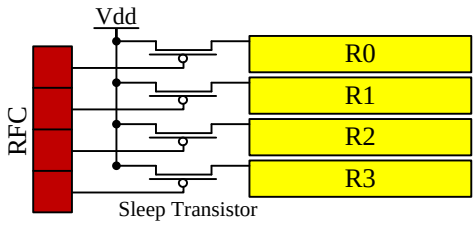

Figure 5: Micro-architecture extension for Vdd-gated RF.

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. Although adding the *RFC* itself will increase power, it provides the ability for runtime configuration which will ultimately lead to more power saving. The *RFC* length is equal to the RF length.

4.3 ISA Extension

To utilize RF power-gating circuit, the running program has to control the *RFC*. For this, the Instruction Set Architecture (ISA) has to be extended in two ways.

First, a new instruction (*Config.* Instruction) has to be expanded for loading an immediate value to the *RFC*. This bitmap value determines the set of registers to be turned-off. Assuming in Fig.5, loading *1001b* to the *RFC* will turn-off *R0* and *R3*, while *R1* and *R2* are on. Depending on length of *RFC* and target architecture, single or multiple *Config.* instruction can be required.

Second, to support function calls, the *RFC* needs to be pushed to and popped from stack (please see the next section for function call support). Briefly, we consider the RFC to be similar to FP (Frame Pointer) in respect to call preserve rules and function-call conventions. For *RFC* stack operations, either add a new instruction can be added, or it can be embedded into the current instructions (e.g. operating on FP). Although the operation is the same in both cases, the code size overhead is lower when embedding the *RFC* push and pop operations to the current instructions. For example, in the Blackfin ISA, Link/Unlink instructions stores/restores FP and SP registers. Link and Unlink instructions can be expanded to handle RFC as well.

4.4 Binary Analysis and Instrumentation

The binary analysis determines the passive periods for individual registers of an RF. The proposed analysis and instrumentation method is static and thus independent from specific input data. Statically operating on the application binary avoids long simulation-based profiling and also simulation ambiguity where register access may differ with input data. Moreover, our analysis operates as a post compiler stage, after all of the compiler optimization and library linking has been completed. In this way, the analyzer achieves a comprehensive insight of register accesses in the program.

As discussed earlier, our AFReP approach manages RF power-gating at function-level granularity. Basically, each function has its own set of active registers, which are 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 an active register. Although, this approach is somewhat conservative, it makes power-gating independent of input data and avoids simulation ambiguity. The remaining registers, which are not referenced thus passive/idle, can be turned-off and power can be gated. The function's passive registers are set through a *Config* instruction in the prologue of each function (see Fig.6). It loads an intermediate value into *RFC* enabling the register power-gating. Before updating the *RFC* however, an extra push is required in order to preserve

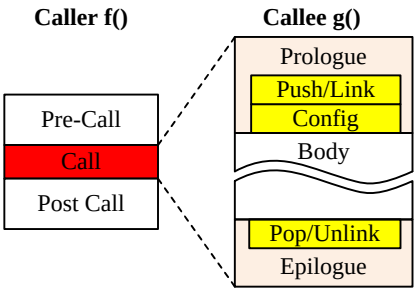

Figure 6: AFReP Function call support.

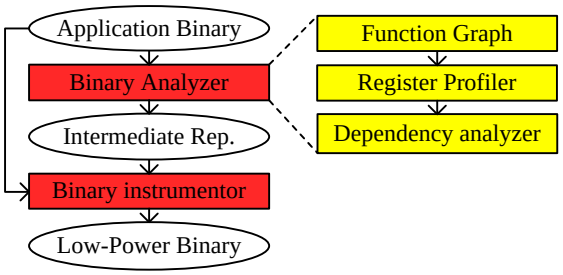

Figure 7: AFReP design flow.

the caller's *RFC*. In the epilogue, 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).

Fig. 7 shows the design flow of our AFReP approach in two main stages: *Binary Analyzer* and *Binary Instrumentation*. The binary analyzer is a composition of function call graph generator, register profiler, and dependency analyzer. At first, all callable functions are detected and a function call graph is created. In the next step, a register profiler parses the body code of each function to determine active and passive registers for each function. Following that, the dependency analyzer traces the register dependency between callee and caller. The dependency analyzer uses the function call graph, and register accesses information to identify registers for power-gating. It also identifies registers that need to be preserved (stored/restored on the stack). These are call-preserved registers that will be turned off in the callee. Their content is saved on the stack to avoid data loss during the power-gating. Finally, the instruction instrumentation inserts the instructions for configuring the RFC and for the additional stack operations. While instrumenting new instructions, it also updates relative addresses shifted due to inserting new instructions.

4.4.1 Register Dependency Analysis

Our AFReP works in function-level, therefore power-gating of passive register should be in respect of calling conventions. Calling conventions control how function arguments are passed and return values retrieved cross a sequence of function calls. Calling conventions divide registers into two groups: call-preserved, and scratch registers. The content of call-preserved registers needs to be preserved (saved by callee before they are used). Therefore, the registers appear unchanged for the caller. Registers used for keeping global data and memory addresses belong to call-preserved registers. In contrast, the content of scratch registers does not need to be saved and restored. Scratch registers are not preserved across function calls. Many special-purpose registers like those used for circular addressing or zero hardware loop implementation belong to this group. In Blackfin processor, registers *(#4-7)* and *(#11-15)* belong to call-preserved category and the remaining are scratch registers.

Throughout a period of power-gating of the call-preserved re-

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 Stack(F)
6:       end if
7:     end for
8:   end for
9: end for
```

gisters, there is a potential data loss for those call-preserved registers that are power gated when theirs contents would be later used in caller function. To avoid this problem, dependency analyzer 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. Therefore, the extra stack pushes should be inserted into the callee prologue before setting the *Config* Instruction. Stack pops are inserted in epilogue after restoring back the previous *RFC* content (Note, functions with multiple returns have multiple epilogues).

Algorithm1 shows the procedure of detecting preserved registers. The dependency analyzer depends on directional function-call graph and register activity information. Every function has four lists; *Parent* (caller functions), *Active*, *Passive*, and *Stack* presenting 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 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. As each function maybe called from different callers, the callee has to preserve all passive call preserved registers that used by any caller functions in order to power-gate them. Please note, each function only needs to keep track of its direct parents (caller functions) and does not care the descendant functions.

Fig.8 shows an example of register dependency in the AFReP binary instrumentation flow. The input is the application binary (generated by compiler), and output is instrumented binary. The *Config.* instruction is inserted in prologue of each function in order to configure RF power-gating circuit in respect to register activity of the function. In this figure, we assume that registers *R0 and R1* are from call-preserved and registers *R2 and R3* are scratch registers. As an example, *R1* is a passive register in function *f1* while it is active in function *f0*. Therefor, its content should be preserved in *f1* by stack posh/pop operations. In contrast, there is no need to preserve the content of *R3* in *f1* because it is a passive register in caller function *f0*. In function *f2*, although *R2* is active in caller function *f0*, it does not preserve since it belongs to scratch register type.

The same method can be expanded 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, 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).

4.5 Switching Delay and Energy Overhead

One challenge in power-gating is the additional delay imposed to the execution for powering-on/off of registers. The delay overhead depends on the characteristics of sleep transistor and detail of power-gating implementation.

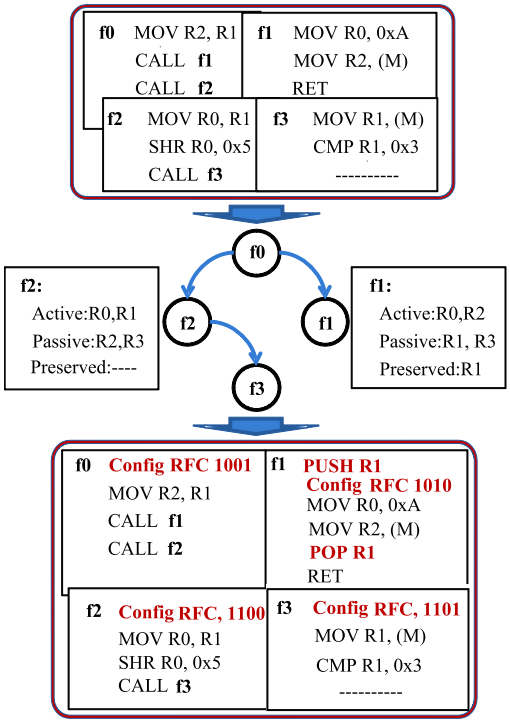

Figure 8: Application binary instrumentation in AFReP design flow.

The critical path in our approach occurs, when a *Config* instruction powers on a register, 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. Fig. 9 highlights this scenario in a simple five stage MIPS-like pipeline. The RFC update and register write instructions are back-to-back.

In MIPS pipeline, register writes occur in the Write Back (WB) stage. Also, loading registers with an immediate operand usually occurs in WB. To relax the timing requirements, configuring the RFC with an immediate value can take place in the execution stage. The RFC is not subject to general RF accesses. In result, the *Config.* instruction has three clock cycles to powering-on R0 before a consecutive write operation reaches WB.

Studies report the power-on latency to vary between one to four processor clock cycles [16]. As Fig. 9 shows, this latency can be masked by pipeline stages. In the Blackfin core of our studies, the distance between decode and RF write back stages is six clock cycles. Therefore, designer can create an appropriate balance where the powering-on energy of the register is minimized, while it is fast enough to turn-on the register as the write time.

Another issue may raise when a register read instruction is followed by a *Config.* instruction. This can cause an error when the register is powered-off too early before the previous read operation has finished. In MIPS pipeline, register read occur in execution

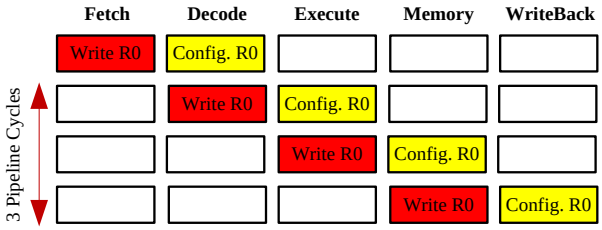

Figure 9: *Config.* instruction in a five stages MIPS-like pipeline.

stage. At that time a consecutively following *Config.* instruction would still be in the decode stage and thus has not yet powered-off the register. The powering-off latency is not on the critical path and register power-gating can be done in background without any effect on execution.

5. EXPERIMENTAL RESULTS

In this paper, we target the Blackfin core [1], a DSP/RISC processor. In full operating mode, Blackfin core tuns at 600MHz with 1.2 Vdd. Blackfin has a *38 x 32bits* ISA register file. An overview of Blackfin's RF has been provided in Section 4.1. We developed for our experiments an Instruction-Set Simulator (ISS) based on the Trap-ADL [7]. Both Blackfin ISS and the ISA have been extended to support the enhanced micro-architecture with the *RFC* register, and *Config.* instruction as outlined in Section 4.2 and Section 4.3. For implementing out AFReP approach, the appropriate length of *RFC* is 34bits, as the zero-hardware-loop registers can be power-gated together. To configure power-gating circuit, two 32bit *Config.* instructions are required for loading a bitmap value into *RFC*.

As AFReP is based on the binary-level analysis, we did not need to modify the compiler (gcc). Benchmarks are selected form MiBench [7] and DSPstone [23] suits as real workloads for control and signal processing applications, respectively. The benchmarks are compiled with gcc (-O3) to produce the input binary for AFReP.

For estimating the register static power, we considered two states: operational and power-gated. The results reported in [21], show that static power in a power-gated register is reduced to 5% of a operational register. The remaining 5% is due to residual static power that cannot be eliminated, in part by the power-gating transistor. The power-gating transistors can be ignored for static power consumption during operations as its contribution over an operational register is negligible. However, the additional *RFC* register slightly increases the overall static power, which we account for in our calculations.

Fig.10 shows the RF static power reduction with our AFReP calculated with input from ISS simulation of the benchmarks. The static power overhead of *RFC* register has been counted in the power saving calculations. Our AFReP approach reduces the RF static power by 60% and 52% for Control and DSP benchmarks on average. Based on register utilization our minimal observed reduction is 46% (e.g. AES) while, AFReP saves up to 73% in other benchmarks (e.g. Quicksort).

In addition, we compared our results with the most closely related previous approach [5], discussed in Section 4.2. In a nutshell, [5] applies drowsy state to unused registers of most executed segment of program (we reference [5] as kernel-level). In the drowsy state, the power gated registers are in state-retention mode where the register can hold its content with cost of additional power overhead. [9]

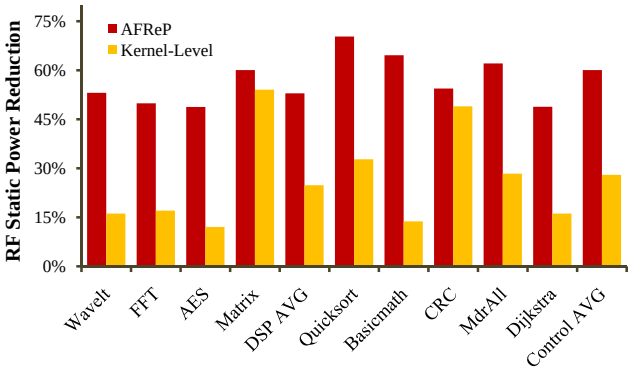

Figure 10: RF static power reduction comparison.

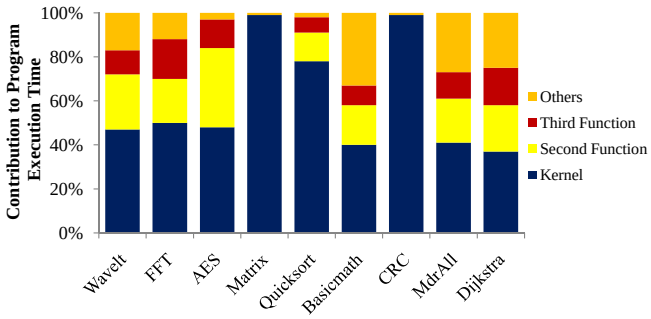

Figure 11: Contribution of functions to entire execution.

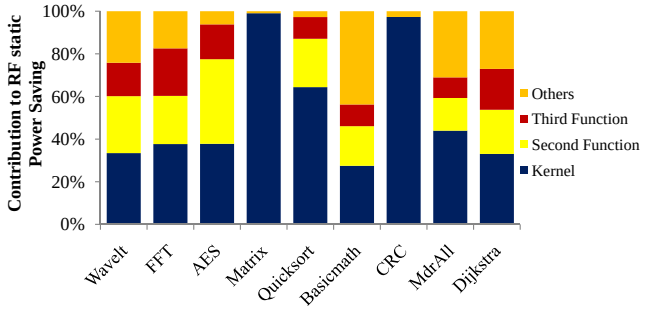

Figure 12: Contribution of functions to overall RF static power saving.

shows that the static power in drowsy register is reduced to 20% of a operational register. For realizing the kernel-level idea of [5], we manually profiled the application to determine function contributing the most to execution time (kernel). Then, we manually added the RFC config instructions to prologue and epilogue of kernel. Fig.10 shows that [5] only reduces RF static power by 31% and 27% on average for control and DSP applications.

It is interesting to note that for the *matrix* and *crc* benchmarks both approaches yield close amount of power savings. At the same time, other benchmarks significantly benefit more from AFReP. To analyze the cause of different effectiveness, we captured each function's contribution to total execution time in Fig. 11. Additionally, we estimated the contribution of each function to overall RF static power saving in Fig. 12. In both figures, we show the contribution of first three functions that have most share in execution time. In addition we show the sum of all remaining functions (Others). We observe that in the benchmarks like *crc* and *matrix* one function has the main contribution in execution. Hence, both approaches operate on the same granularity for these examples and achieve similar results. In contrast, in the most of benchmarks, such as wavelet, AES, FFT and BasicMath, more than one functions significantly contributes to program execution. Since our AFReP approach auto-

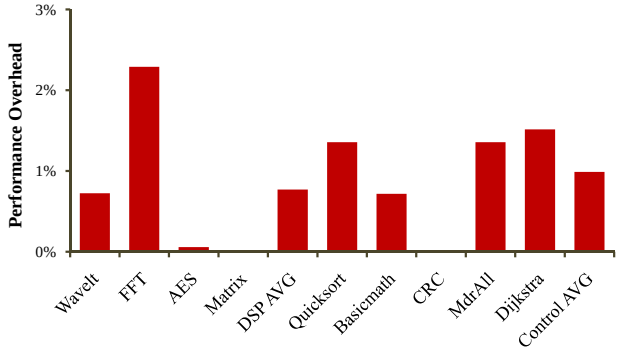

Figure 13: AFReP performance overhead.

matically instruments all functions, it leads to much more power saving than [5] – about 30% on average. It is interesting to note that in the 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 [5] approach at a further disadvantage. For example, in Wavelet, only fourteen registers can be power-gated in the fist function while average of passive registers for others functions is 25 registers. Therefore, the contribution of first function (kernel) to the total execution of Wavelet is 47%. At the same time kernel only contributes with 33% to the total power saving.

Fig.13 shows the performance overhead of our approach due to additional instruction for *RFC* initialization and stack operations. The performance overhead is calculated by running the original application binary and instrumented binary in a AFReP-enhanced Blackfin ISS. The results vary significantly with applications, based on their function granularity. Frequently called small functions like in FFT and recursively called such as in Quicksort, yield a measurable performance overhead. In contrast, CRC with few function calls shows practically no overhead. In most benchmarks, the overhead is negligible in comparison to overall execution time. The performance overhead is less than 1% on average both for control and DSP applications.

6. CONCLUSIONS

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 per-function granularity. It 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 60% and 52% on average for control and DSP applications. Performance overhead is low with 1%. In addition to reducing the overall processor power, a power-gating approach such as ours also mitigates high power density complication RFs. With future processors using further larger register files and being build with smaller feature sizes, the impact of AFReP will even further increase.

7. REFERENCES

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

[2] Tms320c64x/c64x+ dsp cpu and instruction set reference guide. *Texas Instruments*, july 2010.

[3] Snapdragon s4 processors: System on chip solutions for a new mobile age. *Qualcomm, Inc.*, pages 327–339, March 2011.

[4] D. Atienza, P. Raghavan, J. L. Ayala, G. De Micheli, F. Catthoor, D. Verkest, and M. López-Vallejo. Joint hardware-software leakage minimization approach for the register file of vliw embedded architectures. *Integr. VLSI J.*, 41(1):38–48, Jan. 2008.

[5] J. L. Ayala, A. Veidenbaum, and M. López-Vallejo. Power-aware compilation for register file energy reduction. *Int. J. Parallel Program.*, 31:451–467, December 2003.

[6] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: simple techniques for reducing leakage power. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pages 148 –157, 2002.

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

[8] X. Guan and Y. Fei. Register file partitioning and recompilation for register file power reduction. *ACM Trans. Des. Autom. Electron. Syst.*, 15:24:1–24:30, June 2010.

[9] X. Guan and Y. Fei. Register file partitioning and recompilation for register file power reduction. *ACM Trans. Des. Autom. Electron. Syst.*, 15:24:1–24:30, June 2010.

[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*, pages 3 – 14, dec. 2001.

[11] T. M. Jones, M. F. P. O'Boyle, J. Abella, A. González, and O. Ergin. Exploring the limits of early register release: Exploiting compiler analysis. *ACM Trans. Archit. Code Optim.*, 6:12:1–12:30, October 2009.

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

[13] H.-O. Kim, B. H. Lee, J.-T. Kim, J. Y. Choi, K.-M. Choi, and Y. Shin. Supply switching with ground collapse for low-leakage register files in 65-nm cmos. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 18(3):505 –509, march 2010.

[14] M. Mueller, A. Wortmann, S. Simon, M. Kugel, and T. Schoenauer. The impact of clock gating schemes on the power dissipation of synthesizable register files. In *Circuits and Systems, 2004. ISCAS '04. Proceedings of the 2004 International Symposium on*, volume 2, pages II – 609–12 Vol.2, may 2004.

[15] J. Oliver, R. Rao, P. Sultana, J. Crandall, E. Czernikowski, L. W. Jones IV, D. Franklin, V. Akella, and F. T. Chong. Synchroscalar: A multiple clock domain, power-aware, tile-based embedded processor. In *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, pages 150–, Washington, DC, USA, 2004. IEEE Computer Society.

[16] E. Pakbaznia and M. Pedram. Design of a tri-modal multi-threshold cmos switch with application to data retentive power gating. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, PP(99):1 –5, 2011.

[17] S. Park, A. Shrivastava, N. Dutt, A. Nicolau, Y. Paek, and E. Earlie. Bypass aware instruction scheduling for register file power reduction. *SIGPLAN Not.*, 41:173–181, June 2006.

[18] S. Roy, N. Ranganathan, and S. Katkoori. State-retentive power gating of register files in multicore processors featuring multithreaded in-order cores. *Computers, IEEE Transactions on*, 60(11):1547 –1560, nov. 2011.

[19] W. Shieh and C. H. Saving register-file static power by monitoring instruction sequence in rob. *Elsevier Journal of Systems Architecture*, 31:327–339, June 2010.

[20] V. Sridharan and D. Kaeli. The effect of input data on program vulnerability. *EEE Workshop on Silicon Errors in Logic - System Effects*, October 2009.

[21] S. Yang, S. Khursheed, B. M. Al-Hashimi, D. Flynn, and S. Idgunji. Reliable state retention-based embedded processors through monitoring and recovery. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(12):1773 –1785, dec. 2011.

[22] X. Zhou, C. Yu, and P. Petrov. Compiler-driven register re-assignment for register file power-density and temperature reduction. In *45th ACM/IEEE Design Automation Conference*, pages 750 –753, june 2008.

[23] 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*, pages 715–Ű720, 1994.