International Journal of Parallel Programming, Vol. 31, No. 6, December 2003 (© 2003)

Power-Aware Compilation for Register File Energy Reduction

José L. Ayala,[1] Alexander Veidenbaum,[2] and Marisa López-Vallejo[1]

Most power reduction techniques have focused on gating the clock to unused functional units to minimize static power consumption, while system level optimizations have been used to deal with dynamic power consumption. Once these techniques are applied, register file power consumption becomes a dominant factor in the processor. This paper proposes a power-aware reconfiguration mechanism in the register file driven by a compiler. Optimal usage of the register file in terms of size is achieved and unused registers are put into a low-power state. Total energy consumption in the register file is reduced by 65% with no appreciable performance penalty for MiBench benchmarks on an embedded processor. The effect of reconfiguration granularity on energy savings is also analyzed, and the compiler approach to optimize energy results is presented.

KEY WORDS: Register file management; compiler support; energy aware.

1. INTRODUCTION

Low power is an important concern in the design of mobile and embedded systems. Battery technology improvements are not satisfying the fast growing requirements in system design and performance has to be sacrificed to extend the battery life.

[1] Departamento de Ingeniería Electrónica, Universidad Politécnica de Madrid, E.T.S.I. Telecomunicación, Ciudad Universitaria s/n, 28040 Madrid, Spain. E-mail: {jayala,marisa}@die.upm.es

[2] Center for Embedded Computer Systems, University of California, 444 Computer Science Building, Irvine, California 92697-3425. E-mail: alexv@ics.uci.edu

0885-7458/03/1200-0451/0 © 2003 Plenum Publishing Corporation

There are many reasons why power consumption is becoming a limiting factor in high performance processors. New technology processes currently allow higher integration density and larger chips, which directly translates into higher power consumption and heat radiation. Traditional cooling mechanisms are not able to manage this heating increase and aggressive and expensive cooling alternatives are beginning to be used. Also, packaging materials are not designed to work under these operating conditions.[1]

There is a strong relationship between power consumption and fault probability. High temperature in the chip causes glitches, races and increases frequency of errors. At the same time, superscalar processors require parallel functional units and significant resources in terms of number of transistors. Clearly high performance and power consumption are closely related. According to Ref. 2, the superscalar architectures that are capable of issuing multiple instructions increase their power consumption at a higher rate than the increase in performance when compared to conventional single issue RISC architectures.

Reduction in feature size and increase in the number of devices integrated on a similarly sized die have made the leakage current one of the most significant sources of power consumption. Most of the increase is due to very low threshold voltage devices having significantly increased subthreshold currents that charge and discharge static loads. Many architectural and system level techniques have been designed to reduce energy dissipation by gating the clock tree to unused functional units, but not all the power hungry CPU units have been efficiently managed.

As will be shown in the next section, the register file consumes a sizable fraction of the total power in embedded processors and becomes a dominant source of energy dissipation when other power saving mechanisms have been used. Many recent compiler optimization techniques increase the register pressure, and there is a current trend towards implementing larger register files. Both of these facts increase power consumption in this unit, and new techniques to manage this situation are needed. By putting those registers not used by a code section into a low-power state ("drowsy" state) and gating the clock to these unused units, both static and dynamic energy consumption is significantly reduced. In this paper, we propose a mechanism for power-aware register file reconfiguration based on compiler support and code profiling. With a minor change in the ISA (*Instruction Set Architecture*), the compiler can dynamically select the most suitable register file configuration for application requirements. This approach presents no execution penalty and low design time overhead.

The remainder of the paper is organized as follows: Section 2 discusses the sources of power dissipation, Section 3 describes related work in this

area, Section 4 presents our approach to the register file reconfiguration. Finally, Section 5 outlines the compilation environment, Section 6 shows the experimental methodology and results, and several conclusions are drawn in Section 7.

2. SOURCES OF POWER DISSIPATION

In order to achieve significant savings in power aware architectures, it is necessary to select those devices and modules that have the greatest impact on power consumption. There are such devices with high switching capacitance or devices that, for any reason, are permanently working. Those are the *hot points* for energy reduction policies. Memories are a primary example of the former, while clock is an example of the latter.

Clock power consumption reaches 32% of the total power in the Alpha 21364. For previous generation Alpha microprocessors this amount was nearly 40%. The reasons behind this fact are the long clock tree and the undesired switching activity of the devices driven by this network. Memory devices represent a large area in current microprocessors. These devices are not only the on-chip main memory and caches, but also the register file, the instruction and load-store queues, the re-order buffer and the branch prediction tables. Fifteen percent of power consumption in Alpha 21364 comes from the 128 kB of on-chip cache,(3) while in Alpha 21464 the power consumption of the memory reaches a 26%. Actually, in the Alpha 21464, the register file design was several times larger than the 64 KB primary caches(4) and was split to reduce cycle time impact. Both large size and high number of ports result in slow access and high energy dissipation.

Duplicate resources in current microprocessors are a non-negligible source of power consumption. Integer and floating execution units add up to 20% of total power consumption in Alpha 21364 but rarely all of them are working.[3] In Alpha 21464, the execution units add up to 22% of total power consumption. These over-sized resources allow high performance and superscalar architectures, but they also represent a high penalty on power consumption.

The register file consumes a sizable fraction of the total power in embedded processors and becomes a dominant source of energy dissipation when other power saving mechanisms have been applied. Register file power consumption in embedded systems depends very much on system configuration, mainly on the number of integrated registers, the cache size and the existence of a branch predictor table (i.e., it depends on the relative

[3] Depending on program resource requests.

Table I. Register File Size

Processor	Integer RF	Floating RF
ARM	32	8
Power PC	32	32
XScale	32	32
Transmeta Crusoe	64	64

size of other memory devices). In the Motorola's M.CORE architecture, the register file energy consumption could achieve 16% of the total processor power and 42% of the data path power.[5] This amount is even higher when the associated clock tree is taken into account.

Table I shows the register file size for several modern embedded processors clearly showing significant increase in size. This motivates the need for efficient techniques to reduce register file energy consumption in embedded systems. This paper focuses on the register file in embedded processor as many techniques have already been proposed for other units. The register file would dominate energy consumption if other techniques are applied but nothing is done in the register file.

3. RELATED WORK

There has been a lot of interesting work on power oriented resource management in microprocessors. Iyer *et al.*[6] propose a hardware approach to adjust the *Register Update Unit* size and effective pipeline width to the program requirements, saving a considerable amount of energy. This work follows the research of Merten *et al.*[7] who proposed a hardware scheme to detect *hot spots*[4] in running programs. Maro *et al.*[8] followed an approach similar to ours to capitalize on this phenomenon of underutilization of resources. However, though they deal with reduction in power consumption, they focus on monitoring IPC variation and they do not evaluate alternative power reduction ways or model the power consumption in the new logic. Compiler optimization mechanisms have been proposed to reduce power consumption in microprocessors,[9] but these approaches only work on code optimizations and do not count on hardware support. Other solutions include code versioning and selection by the compiler using heuristics and profile data.[10] Finally, there are several software approaches based on code profiling and code annotation,[11] operating system management,[12] and circuit level optimizations in the register file.[13]

[4] A collection of frequently executing basic blocks.

Other techniques target more general memory and register file hierarchy, where caches as well as various types of memories (SRAMs, DRAMs) are available. In these architectures, data transfer and placement are tightly controlled to minimize power consumption per memory access.[14] These approaches do not always obtain high energy savings due to the energy-performance trade off. Also, power conscious implementations of the renaming logic in superscalar processors have been proposed[15] as well as the register file itself.[16] Both approaches can lead to performance penalty when their assumptions are not met. Finally, hardware modifications to the computer architecture based on power-aware reconfiguration policies have also been studied[17] for in-order processors.

The work presented in this paper differs by proposing an energy management of the register file driven by the compiler. The design of the new register file storage cell and related management policies reduce power without performance penalty in the execution time.

4. REGISTER FILE RECONFIGURATION

4.1. Background

Large register files present several advantages: decreased power consumption in the memory hierarchy (cache and main memory) by eliminating accesses, improved performance by decreasing path length and memory traffic by removing load and store operations. Previous work showed that the existing compiler technology could not make effective use of a large number of registers,[18] and the industry followed this statement by implementing most processors with 32 or fewer registers. However, current research in this area[19] and the efforts on optimizing spill code indicate a need for more registers. Sophisticated compiler optimizations can increase the number of variables and the register pressure. Furthermore, global variable register allocation can increase the number of registers that are required.

Many of the power reduction policies work on gating the clock to functional units and architectural modules, but the register file is always clocked. These power management policies usually disable other power hungry devices and, as a result, the register file becomes dominant in power consumption.

While current microprocessors are designed to allow the parallel execution of independent instructions by allocating source operands in different architectural registers and providing multiple functional units, the lack of parallelism in source code, the locality of data and the number of

available resources make the architectural register file underutilized most of the time. Many registers are thus consuming power while not being used.

The main objective of our work is the power reduction in embedded systems, where power consumption is a major constraint. In this way, we will center on embedded processors, which usually use *in-order* architectures because of power, area and cost constraints (e.g., ARM and MIPS processors).

4.2. Proposed Approach

When a section of code does not use all the registers in the register file, it should be possible to use the accessed registers and turn off the unused ones without significant performance penalty. Turning off these unused registers will lead to high energy savings due to the static power consumption decrease and due to the possibility of gating the clock to these devices. When code using the turned off registers is reached they have to be turned back on.

However, something has to be done in the unused registers to prevent the information from being lost. In this way registers that are not used can be turned into a low-power mode which keeps the register contents but reduces static power consumption to a minimum. A number of previous works try to reduce leakage power consumption in cache memories by gating V_{dd}[20] or by reducing voltage source with DVS (Dynamic Voltage Scaling) techniques. The main difference between these approaches and the objective that has to be satisfied in the register file is that the contents of cache line are lost when it is turned into the low-power mode, and needs reloading from main memory or L2 cache (but this cannot happen in the register file). Recent research on this topic has lead to the design of cell memories that can be turned into a low-power mode ("drowsy" as opposed to "sleep") where power consumption is almost negligible but the contents are preserved.[21] Extending these ideas to the register file architecture, leads to a drowsy register file cell like the one presented in Fig. 1. The area overhead of this approach is similar to the results shown by Flauter *et al.* (less than 3%), but the time overhead would be smaller due to simpler access to the register file than to the cache (less than 1% on average).

The proposed solution to deal with the "scalable" register file works as follows. Instead of working with the whole register file, only the currently used registers are addressable and maintained on. The rest of the devices are put into a drowsy state where the static power consumption is reduced to a minimum and the clock distribution network is gated as well. By default, every register in the register file is kept in the drowsy state (the `drowsy` signal is driven with a "1"). Whenever the registers need to be

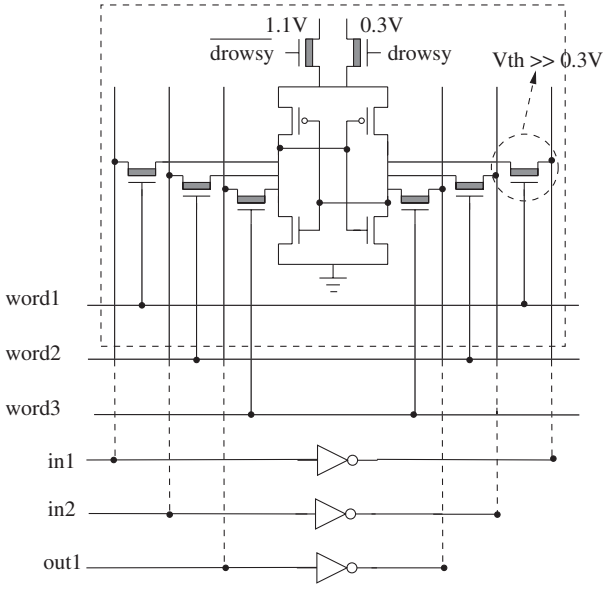

Fig. 1. A drowsy register file cell.

turned on, the required `drowsy` signals are driven with a "0" by decoding the operand of the reconfiguration instruction (a mask) and selecting the proper lines (see Fig. 2).

The product $V_{dd} \times I_{dd}$ is reduced for all the registers kept in the drowsy state due to the decrease in the voltage power supply, and thus the total static power consumption is also reduced.

It is assumed that the ISA is augmented with one instruction to turn a set of registers on and off, therefore, one clock cycle is lost whenever the register file is reconfigured. This penalty is obviously negligible if the register file is reconfigured a reduced number of times (usually, there are 2 or 4 reconfigurations in a run, what means 4 or 8 clock cycles overhead out of thousand cycles). The compiler is responsible for using this instruction to manage the register file. It selects a section of code where N registers are used and inserts an instruction to turn all the other registers off.

This register file management technique requires a code exploration and code generation phase that marks the code sections where the register file reconfiguration should be performed. This phase has to detect those frequently executed code sections with reduced number of required registers and mark the beginning and end of such portion of code. The run-time management and the compiler-driven approach are described next.

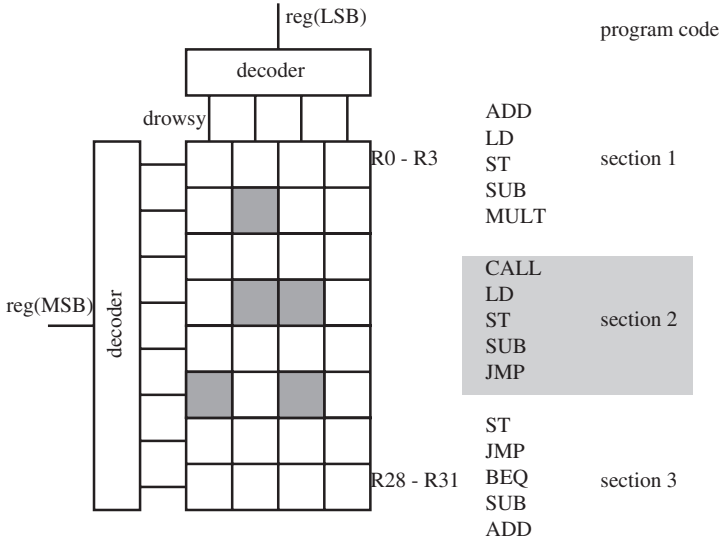

Fig. 2. Activation of the register drowsy state.

5. COMPILATION

The register file management policy that has been outlined above requires a single ISA modification. It needs a signaling mechanism to mark the beginning and the end of a code section. This signaling is performed by incorporating a new instruction that determines when the registers have to be turned on and off. After register allocation, the compiler knows exactly which registers are used by all instructions, and can specify the register requirements per function (or a loop inside the function if a lower granularity is used) in a new instruction that turns the unused registers into the drowsy state. After this portion of code, the same instruction turns on the unused registers. ARM Thumb[22] uses a similar approach for changing the execution context to a second ISA.

The main objective of the compiler approach is to select the optimum configuration for the register file satisfying one important constraint: not to increase the code size by inserting new instructions. A similar problem arises in the area of code-size reduction techniques. One approach used for reducing code size employs a dual instruction set, one of them with shorter operation code and limited number of accessible registers. Obviously, the processor architecture has to support this dual instruction set.[23] A mode

instruction is used to switch between the two sets. A similar instruction can be used to change register file configuration.

Several RISC processors for embedded systems implement the mode functionality: the processor ARM7TDMI from ARM Inc., ST100 Core from ST Microelectronics or Tangent-A5 from ARC. One example of this is the 16 bit Thumb instruction set in ARM processor core. By using the Thumb instruction set it is possible to obtain significant reductions in code size in comparison with the ARM code. As a result of the reduction in code size, the instruction cache energy expended in Thumb mode is also significantly lower. However, many times this reduction in code size is obtained at the expense of a significant increase in the number of instructions executed by the program.[24] Moreover, these architectures do not employ any power saving mechanism with the unused registers in the register file.

As has been previously discussed, our approach requires a compilation support responsible of selecting the most suitable configuration for the smaller register file and marking when it has to be used after the code analysis. Profiling information can be used to improve the selection of the most time and power consumming function. This section outlines the main ideas of this compiler environment.

The compiler approach consists of the following phases:

1. Phase 1 in which the compiler marks the sections of code that can be optimized in register usage, such as entire functions, loops or code blocks which are supposed to be frequently executed. This compilation phase provides several options to the user at the command line: perform the reconfiguration at the most frequently executed function, perform the reconfiguration at any code function, or perform the reconfiguration at any code loop.
2. Phase 2 in which the profitability of reconfiguring the register file is analyzed. The number of unused registers inside selected sections is counted and if this number is above a given threshold (also defined by the user and usually based on a prior profiling phase) the reconfiguration is performed. If the number of unused registers does not reach this user-defined minimum, the compiler looks for any loop inside the function to perform the reconfiguration at this lower granularity level. The definition of the threshold leads the search to the most interesting functions where to perform the reconfiguration. The most time consumming functions could not be a good target for reconfiguration if they require many registers. In that case, the user defined threshold moves the selection to other smaller functions where the register usage is lower and more energy can be saved.

3. The last phase inserts the marking instructions at the beginning and the end of the selected code sections. This compilation phase writes assembler instructions into the code, which will be translated into object code by the next compilation phase.

Figure 3 represents the simplified flow graph for this compiler.

This power aware compilation has been implemented in the GNU compiler, *gcc*. The main compilation phases available in gcc that have been modified were the basic-block analysis and the assembler generation. There is little overhead compilation time if the profitability analysis is successful on first pass. However, if the compiler has to look for any inner loop where to perform the reconfiguration, some of the compilation phases have to be run again, increasing the compilation time. Therefore, a user can allow or disable such functionality in the compiler.

This increase in the compilation time is not a serious drawback if we consider that embedded systems run applications for a long time (or even, "forever"). Consequently, it is affordable to sacrifice compilation time for saving power, specially if there is no penalty on the execution time, as it is the case.

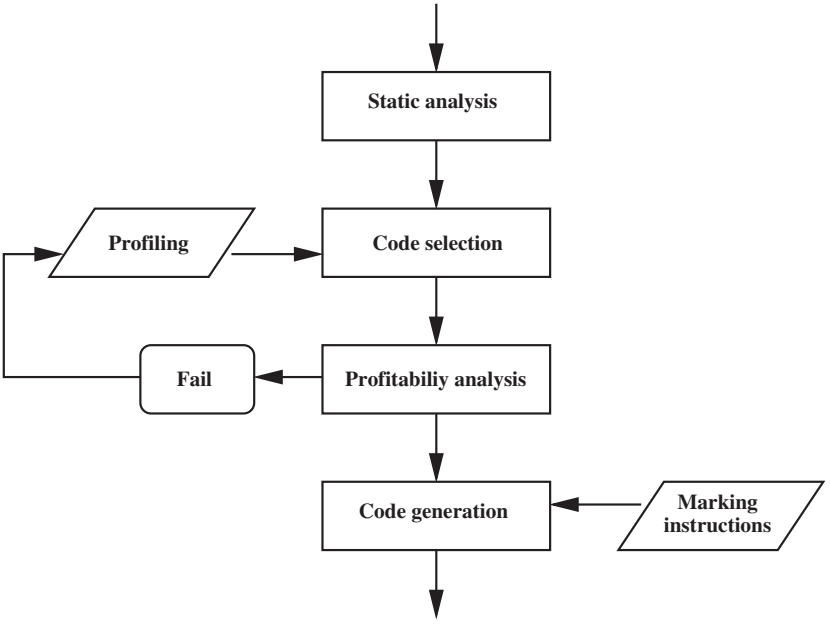

Fig. 3. Compilation flow.

There is a tradeoff in the granularity and the obtained energy savings, and the compiler has to analyze the options in order to obtain optimal results. Higher granularity is usually translated into higher energy savings because the portion of code for which the register file size is optimized is also larger. This reduces the overhead of turning registers on and off. Therefore, a function-level granularity should lead to the best results in power consumption. However, this is not always true because the whole function may require more registers than those specified by the user. In such case, reducing the granularity can help finding time-consuming loops inside the function that can be mapped onto a smaller number of registers. To achieve this, a profiling phase that characterizes the execution frequency in the code can be used. This is the approach that has been followed in the experiments described next.

6. EXPERIMENTAL METHODOLOGY

The approach presented in this paper was modeled via simulation. The simulator used in this work is derived from the SimpleScalar tool suite.[25] SimpleScalar is an execution-driven simulator that implements a derivative of the MIPS-IV instruction set. The version used in our experiments is 3.0c. We have included in the simulator the following enhancements:

- Register file management architecture
- Register file reconfiguration by code annotation

The power models used to validate the ideas come from Wattch,[26] an extension of the SimpleScalar simulator capable of estimating power consumption at the architectural level. Wattch has also been modified to take into account our reconfiguration policies and the overhead from the new reconfiguration logic: CACTI[27] models for the register file, parameterized to work with the run-time reconfiguration.

The baseline configuration of the processor used for our tests is given in Table II.

Different real benchmarks from the MiBench suite,[28] compiled on an x86 machine for the target processor previously described, have been simulated for the proposed architecture and compiled with the also proposed compiler.

Granularity is an important issue in these experiments. As a first approach and in order to keep consistency between experiments, a single-function granularity level is selected. However, some cases require a more careful management. In some cases the whole function cannot be mapped

Table II. Baseline Processor Configuration

Baseline architecture	
Clock	600 MHz
Data width	32 bits
Scheduling	in-order
Decode width	2 insts/cycle
Issue width	2 insts/cycle
Commit width	2 insts/cycle
Functional units	1 integer ALU
	1 integer multiply/divide unit
	1 FP ALU
	1 FP multiply/divide unit
Register file	64 registers

into the number of registers allowed by the user. If the most time consuming function that is considered for the reconfiguration contains any time consuming loop, reducing the granularity to the loop-level and performing the register file reconfiguration in this portion of code allows a smaller number of registers and saves an extra amount of energy.

Finally, if more than one function is selected to perform the register file reconfiguration, higher energy savings are expected with minimal performance overhead. This case has also been considered in the following experiments by enabling the reconfiguration for the two most time consumming functions.

6.1. Results

For the first set of experiments, the compiler is not allowed to descend below the function level granularity in case the number of registers is above the user-defined threshold.[5] Only the most time consuming function is compiled with register management.

Figure 4 shows a reduction in the total energy consumption in the register file achieved with this approach. As can be observed in the graph, total energy consumption in the register file has been reduced by 43%, on average. We can observe benchmarks with really high energy savings (from 70 to 95%). These correspond to those with a function which dominates the execution time and has a low register usage (under 15 registers). There

[5] Set, in this case, to 15 registers.

are also some benchmarks which achieve lower energy savings (from 10 to 23%). These correspond to those without a single function which dominates the execution time. In this case, higher power savings can be expected if the reconfiguration policy is not restricted to only the most time consuming function. Profiling results show that 90% of the execution time is expended in 2 or 3 functions.

Two of the benchmarks in Fig. 4, *Susan* and *FFT*, cannot be executed with only 15 registers as this setting requires. Therefore, the compiler is not able in this case to insert the reconfiguration instructions, and no energy savings are obtained. They are good candidates for a lower granularity compilation approach.

For the second set of experiments, the previous analysis is repeated but allowing the compiler to descend into the function granularity in case the number of registers is above the user-defined threshold. Obviously, only two benchmarks are expected to show any change on the energy savings achieved. Figure 5 shows the new energy savings achieved after the new compilation flow. As can be observed, *Susan* and *FFT* benchmarks fit now the user-defined threshold and the register file reconfiguration has been performed for the most time consuming loop inside the most time consuming function. With this second approach, the register file energy consumption for the whole set of benchmarks can be reduced on nearly 50% on average, at expense of longer compilation time when descending into granularity is required.

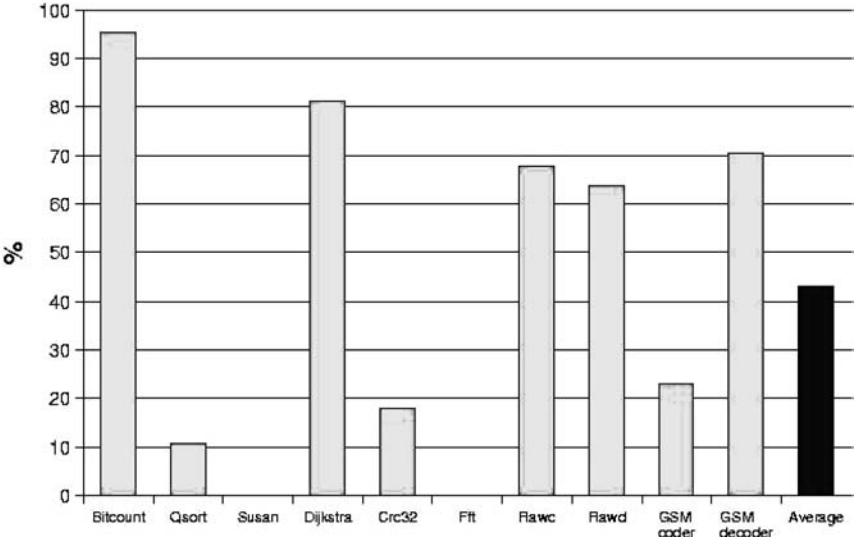

Fig. 4. Reduction in register file energy consumption with function-level granularity.

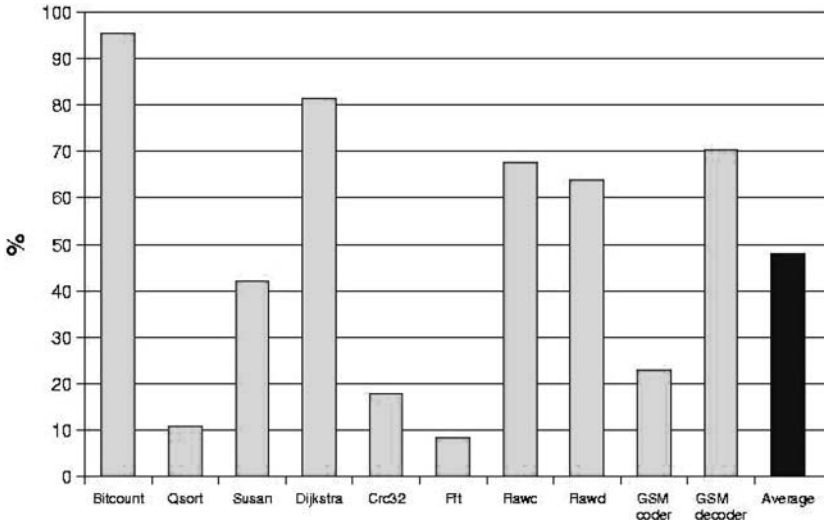

Fig. 5. Reduction in the register file energy consumption with lower granularity.

Finally, the last set of experiments has been run selecting the two most time consumming functions to perform the reconfiguration, and enabling as well the ability to descend into the granularity level. Figure 6 shows the energy savings in the register file after this new simulation process.

As can be seen, several benchmarks increased the energy saving achieved after performing the reconfiguration for the two most time consumming functions (*susan*, *fft*, etc). Some of them have significantly increased the energy savings (*crc32*, for example), due to the reduced register usage but intensive execution of the second function selected. On the other hand, some benchmarks have not modified the energy saving achieved (*dijkstra*, *bitcount*) because there is only one time-consumming function in the source code. Overall, energy savings grow up to 65% on average when selecting two target functions to perform the reconfiguration. The drawback of this approach is the increased compilation time for enabling the ability to descend into the granularity level and perform the extended compilation flow for two functions, but as was explained before, this does not affect the performance of the final application, and can be perfectly afforded.

7. CONCLUSIONS

After clock gating and power-aware system level policies have been applied for power minimization in embedded systems, energy consumption

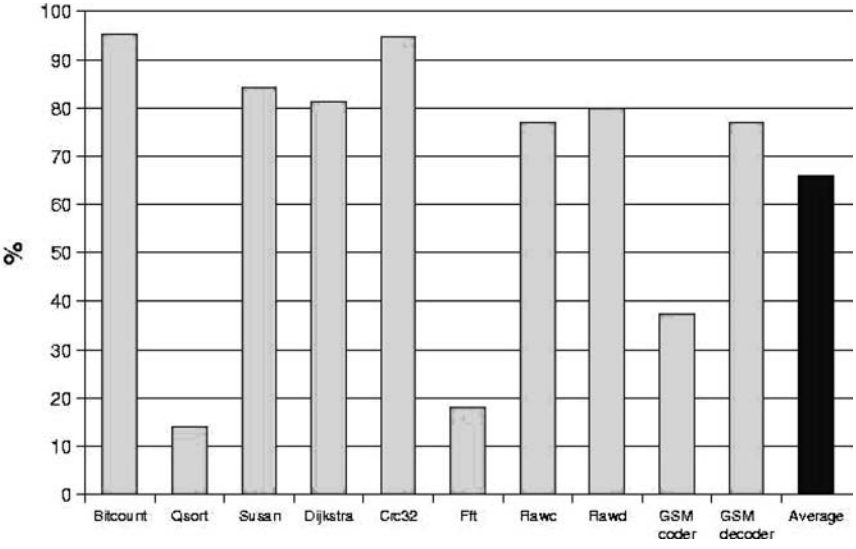

Fig. 6. Reduction in the register file energy consumption when the descend into granularity is allowed and two target functions are selected.

in the register file becomes dominant. Current sophisticated compiler optimizations also require larger register files and increase the register pressure. In this paper, a power-aware reconfiguration of the register file driven by a compiler is presented. The lack of parallelism in the source code, the locality of data and the locality in the accesses to the register file leave most of the registers unused at any given time, while a small number of them is frequently accessed. The mechanism described in the paper marks sections of code where this behavior can be exploited. Turning the unused registers into a low-power state in these code sections leads to significant total energy savings, without performance penalty.

Overall, experimental results collected from selected MiBench benchmarks show a 46% total energy reduction in the register file when the most time-consuming function is selected to perform the reconfiguration of the register file. In many cases, these results can be improved if a lower granularity in the code section is selected by the compiler. Also, negligible increase in the execution time is achieved.

The main contributions of the paper can be summarized as follows. It has been shown how much energy can be saved in embedded processors with a "gated" register file. A power-aware reconfiguration policy to fit the register file size to the program requirements has been presented.

Finally, a compiler support environment to perform these tasks has been outlined and the in uence of the granularity in the optimizations has been analyzed.

REFERENCES

1. S.-W. Lee and J.-L. Gaudiot, *Power Considerations in the Design of High Performance Multi-Threaded Architectures*, Technical report, Department of Electrical Engineering, University of Southern California (2000).
2. R. Gonzalez and M. Horowitz, Energy Dissipation in General Purpose Microprocessors, *IEEE Journal of Solid-State Circuits*, **9**(21):1277–1284 (September 1996).
3. M. K. Gowan, L. L. Biro, and D. B. Jackson, Power Considerations in the Design of the Alpha 21264 Microprocessor, *Design Automation Conference* (1998).
4. R. P. Preston *et al.*, Design of an 8-Wide Superscalar RISC Microprocessor with Simultaneous Multithreading, *International Solid-State Circuits Conference* (2002).
5. D. R. Gonzales, Micro-RISC Architecture for the Wireless Market, *International Symposium on Microarchitecture* (1999).
6. A. Iyer and D. Marculescu, Power Aware Microarchitecture Resource Scaling, *Design and Test in Europe* (2001).
7. M. C. Merten, A. R. Trick, C. N. George, J. C. Gyllenhaal, and W. W. Hwu, Hardware-Driven Profiling Scheme for Identifying Program Hot Spots to Support Runtime Optimization, *International Symposium on Computer Architecture* (1999).
8. R. Maro, Y. Bai, and R. I. Bahar, Dynamically Reconfiguring Processor Resources to Reduce Power Consumption in High-Performance Processors, *Workshop on Power Aware Computing Systems* (2000).
9. M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye, Influence of Compiler Optimizations on System Power, *Design Automation Conference* (2000).
10. A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum, and A. Nicolau, Architectural and Compiler Strategies for Dynamic Power Management in the COPPER Project, *International Workshop on Innovative Architecture* (2001).
11. A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum, and A. Nicolau, Profile-Based Dynamic Voltage Scheduling Using Program Checkpoints in the COPPER Framework, *Design and Test in Europe* (2002).
12. G. Savransky, R. Ronen, and A. Gonzalez, Lazy Retirement: A Power Aware Register Management Mechanism, *Workshop on Complexity Efficient Design* (2002).
13. J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero, Software and Hardware Techniques to Optimize Register File Utilization in Lovliw Architectures, *International Workshop on Advanced Compiler Technology* (2001).
14. K. Inoue, *High-Performance Low-Power Cache Memory Architectures*, Ph.D. thesis, Kyushu University (2001).
15. D. Ponomarev, G. Kucuk, and K. Ghose, Energy-Efficient Design of the Reorder Buffer, *International Workshop on Power and Timing Modeling, Optimization and Simulation* (2002).
16. N. S. Kim and T. Mudge, Reducing Register Ports Using Delayed Write-Back Queues and Operand Pre-Fetch, *International Conference on Supercomputing* (2003).
17. J. L. Ayala, M. López-Vallejo, A. Veidenbaum, and C. A. López, Energy Aware Register File Implementation Through Instruction Predecode, *International Conference on Application-Specific Systems, Architectures and Processors* (2003).

18. S. A. Mahlke, W. Y. Chen, P. P. Chang, and W. Hwu, Scalar Program Performance on Multiple-Instruction Issue Processors with a Limited Number of Registers, *Hawaii International Conference on System Sciences*, pp. 34–44 (1992).
19. M. Postiff, D. Greene, and T. Mudge, *The Need for Large Register File in Integer Codes*, Technical Report CSE-TR-434-00, Electrical Engineering and Computer Science Department, The University of Michigan, USA (2000).
20. M. Powell, S. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar, Gated-v_{dd}: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories, *International Symposium on Low Power Electronics and Design* (2000).
21. K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and M. T, Drowsy Caches: Simple Techniques for Reducing Leakage Power, *International Symposium on Computer Architecture* (2002).
22. ARM, *ARM7TDMI*, Technical Manual (2001).
23. A. Halambi, A. Shrivastava, P. Biswas, N. Dutt, and A. Nicolau, An Efficient Compiler Technique for Code Size Reduction Using Reduced Bit-Width ISAs, *Design and Test in Europe* (2002).
24. A. Krishnaswamy and R. Gupta, Profile Guided Selection of ARM and Thumb Instructions, *ACM SIGPLAN* (2002).
25. T. Austin, E. Larson, and Dan Ernst, Simplescalar: An Infrastructure for Computer System Modeling, *Computer*, **35**(2):59–67 (February 2002).
26. D. Brooks, V. Tiwari, and M. Martonosi, Wattch: A Framework for Architectural-Level Power Analysis and Optimizations, *International Symposium on Computer Architecture* (2000).
27. S. J. E. Wilton and N. P. Jouppi, *An Enhanced Access and Cycle Time Model for On-Chip Caches*, Technical Report 93/5, Computer Science Department, University of Wisconsin-Madison, USA (1994).
28. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, MiBench: A Free, Commercially Representative Embedded Benchmark Suite, *Annual Workshop on Workload Characterization* (2001).