Run-time Scaling of Microarchitecture Resources in a Processor for Energy Savings

Anoop Iyer Diana Marculescu

Electrical and Computer Engineering Department
Center for Electronic Design Automation
Carnegie Mellon University
Pittsburgh PA 15213
Email: {aiyer, dianam}@ece.cmu.edu

Abstract

In this paper we present a strategy for run-time profiling to optimize the configuration of a microprocessor dynamically so as to save power with minimum performance penalty. The configuration of the processor is changed at run-time to attain the optimal energy consumption for the program. Experiments on some benchmark programs show good savings in total energy consumption; we have observed a decrease of up to 25% in energy/cycle and up to 10% in energy per instruction. Our proposed approach can be used for energy-aware computing in either portable applications or in desktop environments where power density is becoming a concern. This approach can also be incorporated in larger power management strategies like ACPI.

1 Introduction

Power dissipation of microprocessors is becoming an important concern for designers because of two factors: (1) the market for mobile and embedded systems is expanding at a rapid rate and in such systems, battery life is important and power is at a premium; (2) complex designs and large on-chip caches present in modern chips require thermal management strategies to prevent the chip from overheating; this is true not only for mobile computing, but for conventional processor design as well. In this paper we present microarchitectural level control and scaling of resources to address the issue of power consumption.

Although low-power design has been an active area of research for the last decade or so, the problem of power modeling and optimization at the microarchitectural level has only recently been addressed. An overview of various approaches to system level power management, power optimization and efficient processor design is given in [1]. These cover memory and cache optimizations, dynamic power management using low-power sleep modes, dynamic supply voltage variation techniques, and compiler and software optimization techniques. Most of these solutions are static in nature; they do not allow for adaptation to the application.

It has been shown [3] that most of the execution time of a program is spent in several small critical regions of code. These small collections of basic blocks exhibiting strong temporal locality are called *hotspots*. Our strategy consists of identifying these hotspots in a program at run-time, characterizing each hotspot in terms of its power usage and arriving at an energy-optimal configuration of resources for the processor. This optimal configuration is determined by evaluating the power usage of the processor for different configurations.

In what follows, we consider a typical superscalar configuration, based on the reservation station model [4]. This structure is used in modern processors like the Pentium Pro and the PowerPC 604. The main difference between this structure and the one used in other processors (like the DEC Alpha 21264 and HP PA-8000) is that the reorder buffer holds speculative values and the register file holds only committed, non-speculative data, whereas for the second case, both speculative and non-speculative data are in the register file. However most of the following discussion is independent of this model and applies to any superscalar processor architecture.

2 Hardware Profiling

Our hardware profiling scheme is an implementation of the hotspot detection scheme proposed by Merten et al [3]. Since a hotspot is a collection of tightly coupled basic blocks, a table of branch addresses and execution counters is used to detect frequently executing code locations. Branches which are executed frequently are marked as candidate branches. If the number of candidate branches being executed in a preset time window exceeds a given threshold, we say that we have detected a hotspot. The number and nature of hotspots detected depend on the application; some applications have fewer than 10 hotspots, while large applications like *gcc* have about 50. More details of the scheme are available in [3].

Unit	Power
Floating point ALU	9
Integer ALU	3
Register File	1
Instruction Window	2

Table 1: Relative power consumption of the four hottest parts of the Simplescalar processor model

3 The Energy-Optimal Configuration

Once a hotspot has been detected, we need to determine an optimum configuration of the processor for that hotspot. Several units of the processor can be scaled to vary the processor's configuration. For every unit which we decide to scale, we have a *maximum* size (corresponding to the total physical capacity of that unit) and an *active* size (corresponding to the enabled capacity). When the active size of any unit changes, the processor is said to switch configurations. For example, the reorder buffer can have a maximum size of 64 entries, and an active size of 32; this means that only 32 entries of the table can be used for reordering. Similarly a processor may have a maximum issue width of four instructions per cycle and an active issue width of two. The caches, branch prediction tables, load-store queue, etc. can be selected as candidates for scaling at run-time, depending on their impact on power consumption in the implemetation.

The estimation of the impact of scaling a resource on energy consumption is not always straightforward. For example having a smaller cache will lead to a lower power consumption in the cache but will lead to a larger miss rate and thus could actually increase the overall energy. From our studies we have found that the size of the reorder buffer and the effective width of the pipeline are the two factors which most strongly control the power consumption of the processor, since these determine the flow of instructions through the pipeline. The effective width of the pipeline is changed by simultaneously changing the widths of the fetch, decode, issue and retire stages.

We define the optimum as that configuration which leads to the least energy dissipated *per committed instruction.* The energy per instruction is the product of the energy per cycle and the cycles per instruction (CPI). This figure is in fact the reciprocal of MIPS per Watt, and is a good metric which reflects both performance and power consumption. To determine the optimum configuration, we need a way to determine approximate energy dissipation statistics in hardware. For this purpose, when a hotspot is detected, two counter registers are set in motion: the *power register* and the *instruction count register* (ICR). These registers are not part of the processor's register file and are not visible to either the programmer or to the renaming hardware; they are just registers used for storing power estimate figures.

The power register is used to maintain power statistics for the four most power-hungry units of the processor. Using the organization and modeling of Wattch [5], in our processor model we have identified these four units to be (1) floating point ALU (2) integer ALU (3) register file (4) instruction window. The relative power dissipation of each unit is shown in Table 1. These figures are not exact but are rounded off for simple integer arithmetic. Multiplying these power figures with the access counts of the respective units provides a rough estimate of the energy consumed in each cycle. These multiplications could be implemented as integer shift and add operations, pipelined if necessary. A schematic view of this process is shown in Figure 1. It should be noted that depending on the implementation, the four hottest units may be different from the units shown here; their relative power consumption may be different too. However the same basic scheme can be implemented on any processor architecture.

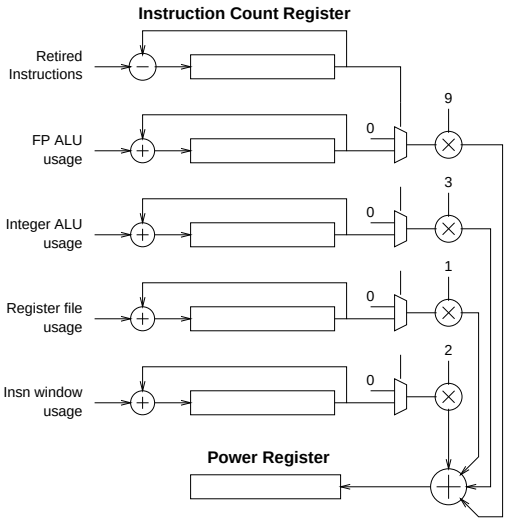

Figure 1: Power profiling hardware

The instruction count register (ICR) is used to keep a count of the number of the number of instructions retired by the processor. When a hotspot is detected, the ICR is initialized with the number of instructions to count (1024 in our experiments) and a finite state machine (FSM) is activated, tracking the processor's configuration. In our experiments we varied the reorder buffer size from 16 to 64 in steps of 16 and the effective pipeline from 2 to 4 to 8; thus our FSM had $4 \times 3 = 12$ states. During each cycle, the ICR is decremented by the number of instructions retired in that cycle. When the ICR reaches zero, the power register is sampled to obtain a figure proportional to the energy dissipated per instruction. If there are n parameters of the processor to vary, exhaustive testing of all configurations would mean testing all points in the n-dimensional lattice for a fixed number of instructions. If the number of parameters increases, evaluation of only selected configurations could be used instead of the exhaustive evaluation being done here. After cycling through all the interesting configurations, the processor is then switched to the optimal one. When the profiling hardware has detected that execution has strayed out of the hotspot, the processor is switched back to its default state, which may be its maximum configuration or an intermediate smaller configuration.

The success of this method depends on the fact that within a hotspot, program behavior remains approximately the same; and that a configuration which was evaluated as optimal at the beginning of the hotspot remains optimal throughout the execution time of that hotspot.

4 Implementation and Results

The above ideas were implemented on the Simplescalar simulator [6]. Simplescalar is a popular industrial-strength tool which implements a derivative of the MIPS-IV instruction set, and has various configuration options including a superscalar out-of-order simulator which we used for our experiments. Our implementation also features an accurate fetch unit model which aligns I-cache accesses to block boundaries. The power modeling we used to report power figures was based on Wattch [5], which is an extension to the Simplescalar simulator. Wattch has various choices for power modeling; the one we chose for our application assumes support for aggressive clock gating styles and parameterized power calculation. This implies that power consumption is scaled according to the number of units (in case of multiple functional units) or ports used (in case of register files and caches). Wattch also uses the scheme implemented in Cacti [7] for optimizing caches and cache-like structures based on delay analysis. We implemented the hardware profiling schemes as extensions into the SimpleScalar simulator and used Cacti routines to optimize the structures involved. The power consumed by the additional hardware was modeled according to Wattch conventions.

We performed experiments with programs from the Spec95 CPU benchmark suite [8] as well as the Mediabench suite [9]. The results we obtained are summarized in Figure 2. The graphs in the figure show the savings obtained by the above run-time resource scaling scheme against the baseline fixed processor configuration with a reorder buffer size of 64 entries and an effective pipeline width of 8 instructions. In the energy and power savings graphs, two figures are shown: the first is the case where we assume a 10% power overhead for units which are shut off; the second assumes ideal clock gating with no overhead. Shrinking feature sizes and increasing leakage currents have made clock gating less advantageous than before; the figures for ideal clock gating are shown here to demonstrate the extra savings that can be achieved with an aggressive circuit design. The graph for IPC shows the performance of the fixed configuration versus the performance using our dynamic resource scaling scheme.

We obtain significant energy savings in most applications, but at the cost of some IPC loss. We have performed some experiments in which we specified an acceptable level of performance loss (in our case one-eighth or 12.5%) and the FSM which evaluates configurations was programmed to reject configurations whose performance was worse than this tolerance level. In all cases the IPC loss remains below 12.5% but the savings in power and energy are slightly less than those shown here.

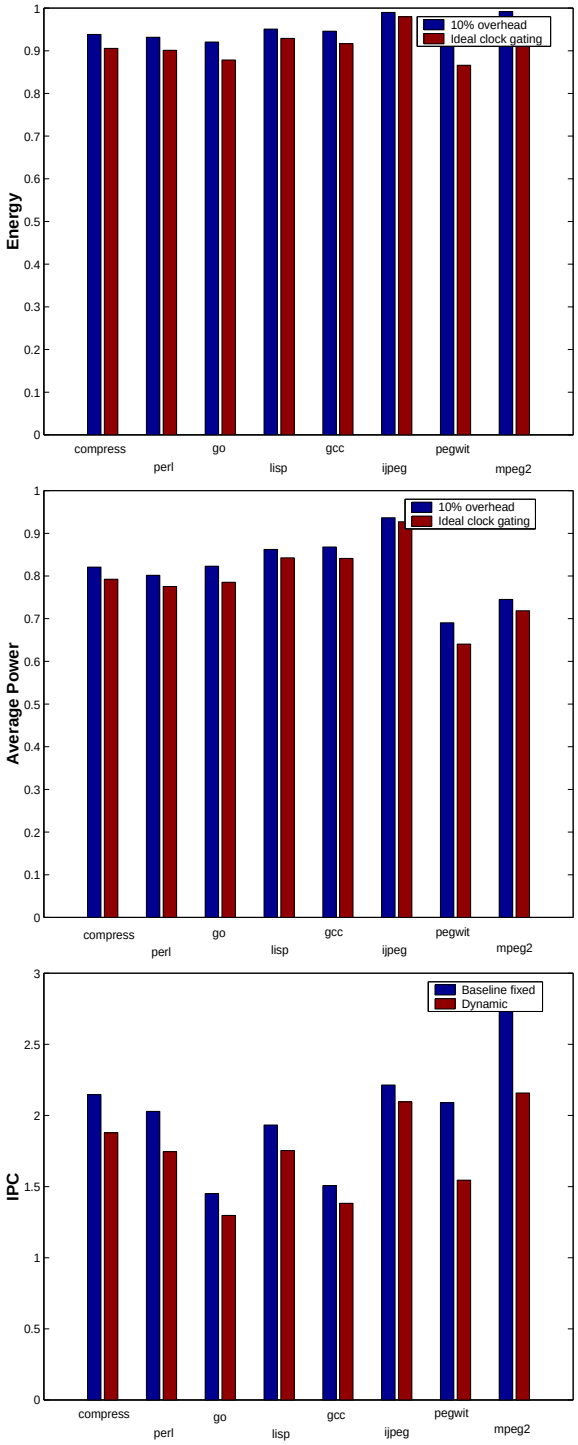

Figure 2: Results obtained from our experiments

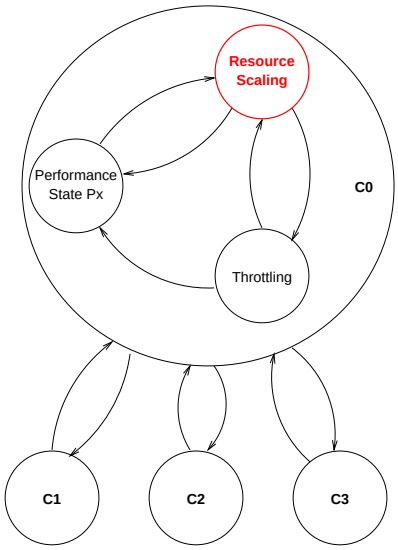

Figure 3: Resource scaling in the context of ACPI

5 Conclusion

The techniques outlined in this paper for optimization of the processor configuration based on run-time profiling show good promise for energy savings. This scheme can be incorporated in the context of larger schemes like ACPI, by incorporating it as an extra stage in the active mode of operation of a CPU as shown in Figure 3. Our method is better than static throttling because static throttling does not decrease the overall energy required to perform a task; it only spreads out the energy usage over a longer period of time, at a direct cost to performance. The scheme we have described above can also contribute to thermal management schemes since we are able to achieve significant reduction in the average power. The power monitoring hardware could be made more sophisticated for advanced power management strategies as well; for example lack of usage of the floating point units by integer applications could be detected and the power to the FPU could be shut off entirely, providing more power savings. Our proposed scheme needs a little adaptation to account for context switching in multitasking systems. Overall we believe that microarchitecture level resource scaling can lead to a significant saving in power consumption while retaining reasonable performance levels.

References

[1] L. Benini and G. de Micheli, "System-level power optimization: Techniques and tools," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 1999.

[2] D. Marculescu, "Profile driven code execution for low power dissipation," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 2000.

[3] M. C. Merten, A. R. Trick, C. N. George, J. C. Gyllenhaal, and W. W. Hwu, "A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1999.

[4] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Quantifying the complexity of superscalar processors," Tech. Rep. 1328, University of Wisconsin-Madison, CS Department, November 1996.

[5] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2000.

[6] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," Tech. Rep. 1342, University of Wisconsin-Madison, CS Department, June 1997.

[7] S. J. E. Wilton and N. P. Jouppi, "An enhanced access and cycle time model for on-chip caches," Tech. Rep. 93/5, Western Research Laboratory, DEC, July 1994.

[8] "Spec cpu95 benchmarks." On the internet: http://www.spec.org/osg/cpu95.

[9] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: a tool for evaluating and synthesizing multimedia and communications systems," in *International Symposium on Microarchitecture (Micro)*, 1997.