Microarchitecture-Level Power Management

Anoop Iyer, *Member, IEEE,* and Diana Marculescu, *Member, IEEE*

***Abstract*—In this paper, we present a strategy for run-time profiling to optimize the configuration of a superscalar microprocessor dynamically so as to save power with minimum-performance penalty. The configuration of the processor is changed according to the parallelism and power profile of the running application. To identify the optimal configuration, additional hardware with minimal overhead is used to detect the parts of the running application which have good potential for energy savings. Experiments on some benchmark programs show good savings in total energy consumption; we have observed a mean decrease of 18% in average power, and 9% in total energy. 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 power-management strategies like advanced configuration and power interface (ACPI) as a replacement for classic thermal management schemes such as static-clock throttling . Our approach is shown to be better than static-throttling methods presently used in power management.**

***Index Terms*—Energy aware computing, power management, resource scaling, voltage scaling.**

I. Introduction

POWER dissipation in 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 and 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 resizing of processor resources to address the issue of power consumption.

A. Prior Work

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]. A large number of these techniques focus on memory and cache power optimization. For example, [2] presents a technique using an additional smaller "L0" cache for storing frequently executed parts of the program, while [3] proposes novel techniques for improving the energy efficiency of caches. The dynamic power management approach to saving power in microprocessors using low-power sleep modes has been implemented in standards like advanced configuration and power interface (ACPI) [4]. Dynamic supply voltage variation techniques also show promise [5] and have been commercially implemented [6]. Microarchitectural level power modeling and simulation tools are presented in [7] and [8]. In [7], an accurate parameterized power simulator (accurate to within 10% when compared with three different high-end microprocessors) is presented, as well as some interesting tradeoffs between energy and performance under varying microarchitecture settings. In [8], the case of datapath dominated architectures is considered, as well as an analysis of the impact of compiler optimizations and memory design on power-consumption values. Software optimization for low-power consumption has also been studied [9].

So far only a few microarchitectural level solutions to the power problem have been proposed; for example, [10] proposes a technique that uses confidence estimation to gate the execution of branches that are most likely to be mispredicted, and [11] presents a new paradigm for adapting the execution of application programs for low power using profiling. [12] presents an analysis of different configurations of superscalar processors and derives the optimal "envelope" for energy-delay product; but their approach is not adaptive. [13] presents a framework for complexity-adaptive processors but does not specify the mechanism for adaptive behavior. In [14], a similar adaptive scheme is presented, but the approach is very coarse-grained, and [15] has a dynamic approach but they look at resizing only the issue queue. The work presented in [16] uses instructions per cycle (IPC) values obtained from profiling to characterize different portions of the code and uses a fixed window of instructions whose execution is monitored in order to reduce the power consumption.

B. Motivation

Most solutions to the power problem are static in nature since they do not allow for adaption to the application. It has been observed in [11] and [17] that there is wide variation in processor resource usage among various applications. In addition, the execution profile of most applications indicate that there is also wide variation in resource usage from one section of an application's code to another. For example, Fig. 1 shows the execution profile of the *epic* benchmark (part of the MediaBench suite) on a typical workload on a eight-way issue processor. We can see several regions of code execution characterized by high IPC values lasting for approximately two million cycles each; toward the end we see regions of code with much lower IPC values.

The quantity and organization of the processor's resources will also affect the overall execution profile and the energy consumption. Fig. 2 shows the variation of the total energy con-

Manuscript received February 26, 2001; revised December 22, 2001. This work was supported in part by NSF Career Award CCR-0084479.

A. Iyer is with AMD, Austin, TX 78741 USA.

D. Marculescu is with the Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213-3890 USA.

Publisher Item Identifier S 1063-8210(02)06535-6.

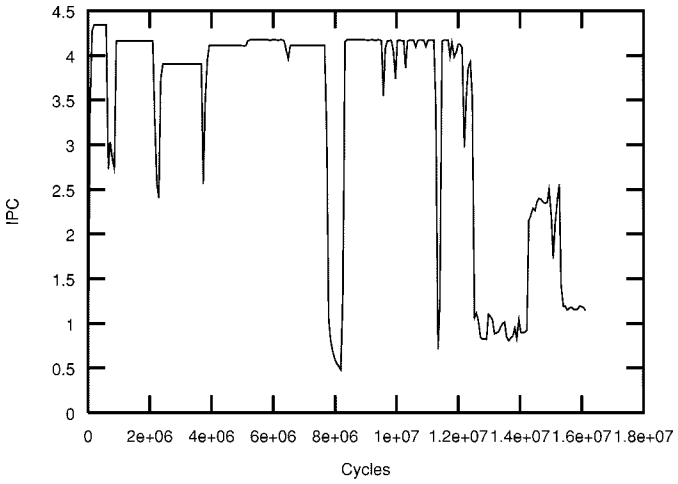

Fig. 1. Execution profile of the epic benchmark.

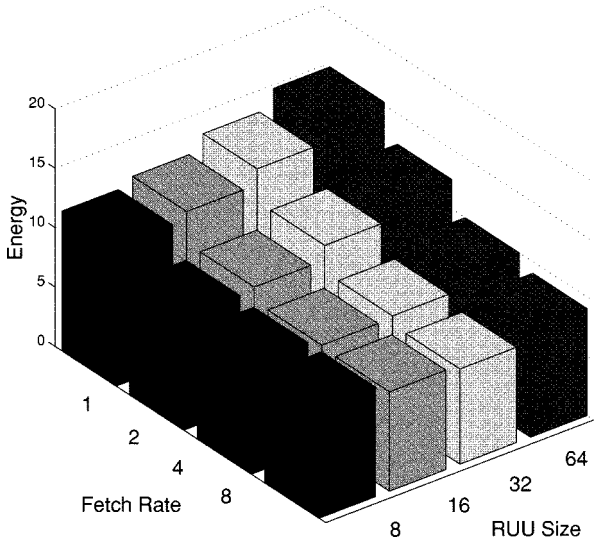

Fig. 2. Energy variation of lisp.

sumption of the *lisp* benchmark with variation in the register update unit (RUU) size and the effective pipeline width. Low-end configurations consume higher energy per instruction due to their inherently high cycles per instruction (CPI); high-end configurations also tend to have high energies in part due to resource usage and in part due to power consumption of unused modules. The ideal operating point is somewhere in between.

Combining the two ideas, we can find the optimal operating point for each region of code in terms of processor resources. The goal of our work is to identify the right configuration for each code region in terms of various processor resources to optimize the overall energy consumption. Our approach allows fine-grained power management at the processor level based on the characteristics of the running application.

We use a hardware profiling scheme to identify tightly coupled regions of code and a hardware-based power estimation method to judge the power requirements for each region of code and scale or resize resources at runtime depending on these estimates. Allocating architectural resources dynamically based upon the needs of the running program, coupled with aggressive clock-gating styles, can lead to significant power savings.

C. Organization of Paper

This paper is organized as follows. Section II presents the framework needed for hotspot detection. We present in Section III, the methodology for finding the optimal configuration. Section IV contains details of our implementation and results on a set of benchmarks. In Section V, we discuss some practical considerations. In Section VI, we conclude with some final remarks.

II. Detecting Hotspots

Let us use the term *basic block* to describe a straight execution path of code ending at any branch or jump instruction. A typical mix of instructions contains one branch every five or six instructions, so the average size of the basic block is also of the order of five or six. In the ideal case, each basic block could be characterized in terms of its parallelism and resource usage and the configuration of the processor could be changed dynamically for each basic block. However, in modern processors, that would require changing the configuration of the processor during almost every cycle, which is not feasible to implement. Hence, we need to look at collections of basic blocks executing together, called *hotspots*. It has been shown that most of the execution time of a program is spent in several small critical regions of code, or in several *hotspots*. These hotspots consist of a number of basic blocks exhibiting strong temporal locality. Merten *et al.* have presented in [17] a scheme for detecting hotspots at run time. We have implemented a reduced version of their scheme which we describe as follows:

- Since a hotspot is a collection of frequently executing basic blocks, identifying hotspots involves keeping a count of all branch instructions committed and finding the most frequent branches. We use a cache-like structure called the *branch behavior buffer* (BBB) to keep track of branches. Each branch has an entry in the BBB, consisting of an execution counter and a 1-bit candidate flag. The execution counter (9-bits wide as suggested in [17]) is incremented each time the branch is taken and once the counter exceeds a fixed value, the branch in question is marked as a candidate branch by setting the candidate flag bit for that branch. Figs. 3 and 4 show the details. The number of entries in the BBB and the width of the execution counter are determined by the average size of a hotspot; 2K hotspot entries and 9-bit execution counters worked well for the applications we tested. For larger applications with bigger average hotspot sizes, these may have to be higher.
- A saturating counter called the *hotspot detection counter* (HDC) keeps track of candidate branches. Initially all bits of the counter are set; each time a candidate branch is taken the counter is decremented by a value D and each time a noncandidate branch is taken, it is incremented by a value I. When the HDC decrements down to zero, we are in a hotspot. The width of the HDC determines how many candidate branches we want to have before we say that we have detected a hotspot; a very-low HDC width may lead to many false alarms which may not be hotspots, whereas

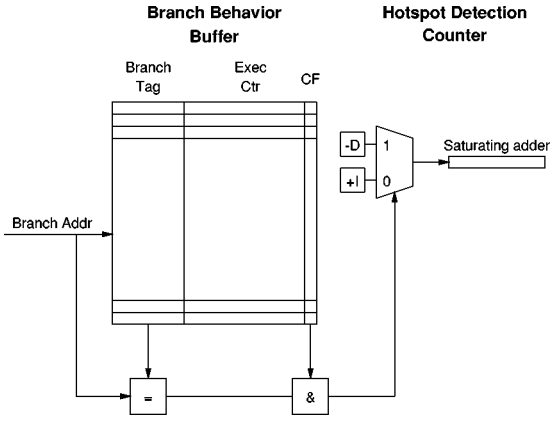

Fig. 3. Hotspot detection hardware.

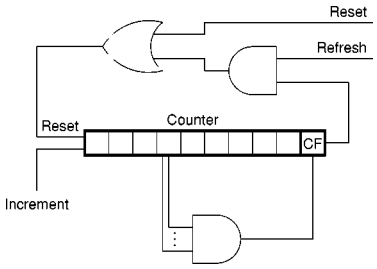

Fig. 4. Implementation of one BBB entry.

a high width may lead to longer times before a hotspot is detected. We found that a 13-bit wide HDC works for most benchmarks. The values of D and I are determined by how fast we want the HDC to count upwards and downwards; their ratio determines the ratio of candidate to noncandidate branches that controls the threshold where we say hotspots are found and lost. For our implementation we chose D as 2 and I as 1.

- The BBB and HDC are left running even when execution is inside the hotspot. When the code strays away from the hotspot, noncandidate branches start to execute more frequently; the HDC then increments to its upper limit eventually and we say that we are out of the hotspot.
- The replacement policy for entries in the BBB is that if there is a conflict, the old entry is retained and the new one discarded. Entries are *not* replaced; this is needed so that the BBB figures reflect the correct execution statistics.
- After every N_F cycles, BBB entries, which are noncandidate entries are flushed. Every N_R cycles, the entire BBB is reset. These two mechanisms ensure that the replacement policy we have adopted does not cause stagnation of entries in the table. Experiments we ran showed that $N_F = 4096$ and $N_R = 65\,536$ were appropriate values.

The amount of time a program spends in hotspots depends on the behavior of the program itself; Fig. 5 illustrates this for the benchmarks that we tested. The average fraction of time spent inside detected hotspots is 92%, with the fraction being higher for MediaBench programs than for the Spec benchmarks. The length of most hotspots is of the order of half a million to tens of millions of cycles, depending upon the application.

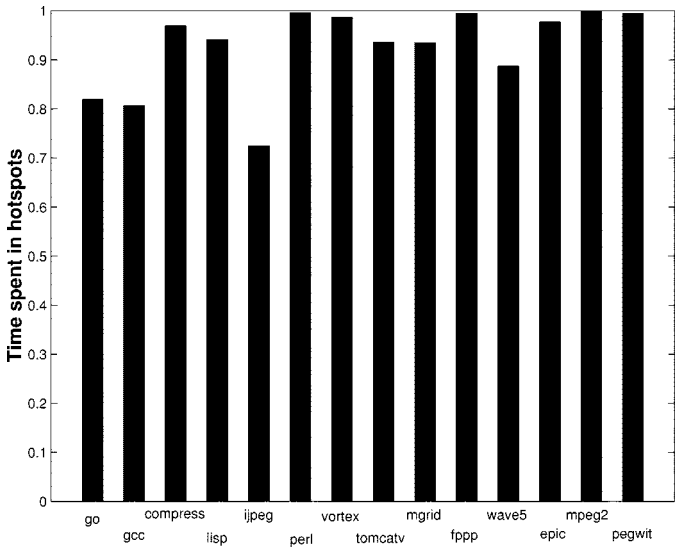

Fig. 5. Fraction of execution time spent in hotspots.

III. Energy-Optimal Configuration

Once a hotspot has been detected, we need to determine an optimum configuration for that hotspot. By the term configuration, we mean a unique combination of several processor parameters under control. We have seen from our experiments and from previously reported results [7] that the size of the RUU and the effective pipeline width are the two factors that most dramatically affect the performance and energy consumption of the processor. Hence, changing the configuration of the processor would mean setting different values for the RUU size and the effective pipeline width. To have a consistent flow of instructions through the pipeline, the fetch, decode, issue and commit widths were all made equal in order to control the effective pipeline width. Since less than half the instructions are memory access instructions, we set the size of the *load-store queue* (LSQ) to be half the size of the RUU. We define the optimum as that configuration which leads to the *least energy dissipated per committed instruction.* We would like to point out that this is equivalent to the power-delay product per committed instruction (the inverse of millions of instructions per second (MIPS) per watt), which is a metric used for characterizing the power-performance tradeoff for a given processor.

A. Energy Profiling in Hardware

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).

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 [7] we collected data on 14 benchmarks to identify the units of the processor which consume the most energy. We identified seven units of processor hardware which when taken together account for over 70% of the energy consumption (excluding clock power) for all applications. The units we identified and their relative

TABLE I
RELATIVE PER-ACCESS ENERGY CONSUMPTION OF THE HOTTEST PARTS OF THE SIMPLESCALAR PROCESSOR MODEL

Unit	Energy
Wakeup logic	1
Reservation stations	3
Register file	2
Data cache	9
Integer ALU	4
FP ALU	9
Resultbus	6

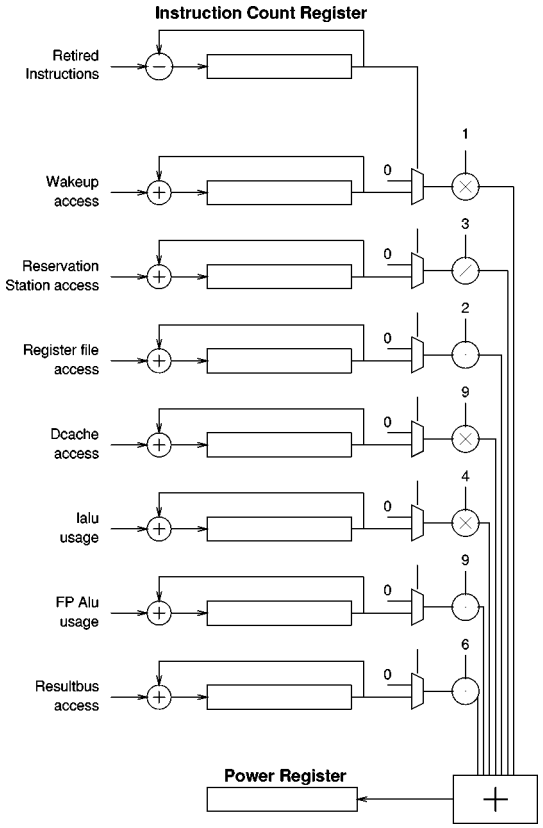

Fig. 6. Power-profiling hardware.

per-access energy dissipations are shown in Table I. 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 Fig. 6. We point out that depending on the implementation, these hottest units may be different from the units shown here or the weights used for estimating power may be different. However, the same scheme can be implemented irrespective of the actual processor.

B. Optimizing the Configuration

When a hotspot is detected, a finite state machine (FSM) walks the processor through all possible configurations for a fixed number of instructions in each. The instruction count register (ICR) is used to keep a count of the number of instructions retired by the processor. It is initialized with the number of instructions to be profiled in each configuration, which we set in our experiments to 1024. The number of instructions for testing each configuration was arrived at empirically; we found that 512 instructions was not accurate enough and 2K instructions was just as accurate as 1K. 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 total energy dissipated. 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. In our experiments, we varied the RUU size and the fetch rate, and ran 1024 instructions in each configuration. Since we were testing configurations with RUU sizes of 16, 32, 48, and 64; and with fetch rates of 4, 6, and 8, we had a total of $4 \times 3 = 12$ configurations, requiring an FSM of only 12 states. A schematic of the FSM is shown in Fig. 7. In general, we have seen that the hotspot detection scheme does not provide many false alarms; but it is indeed possible that the hotspot thus obtained, may be lost before the FSM cycles through all the configurations. In that case, the default configuration of the procesor is restored and execution continues.

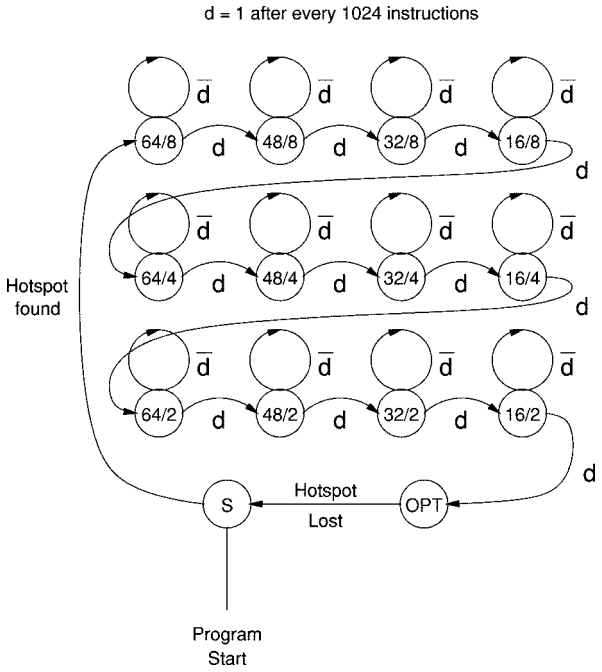

Fig. 7. FSM example.

The approximate energy dissipation statistics obtained by evaluating all possible configurations are then compared and the optimal configuration of the processor for the current hotspot is determined. The processor is then switched to this optimal configuration for the whole duration of the hotspot, which could take several million instructions. As Fig. 5 shows, a majority of the execution time of most applications is spent inside hotspots, and hence, detecting hotspots and optimizing the processor's energy consumption inside hotspots will lead

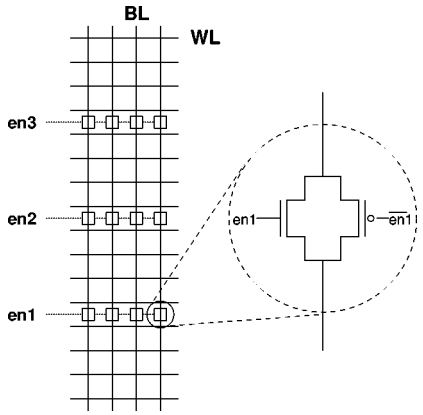

Fig. 8. Segmented bitlines in array structures.

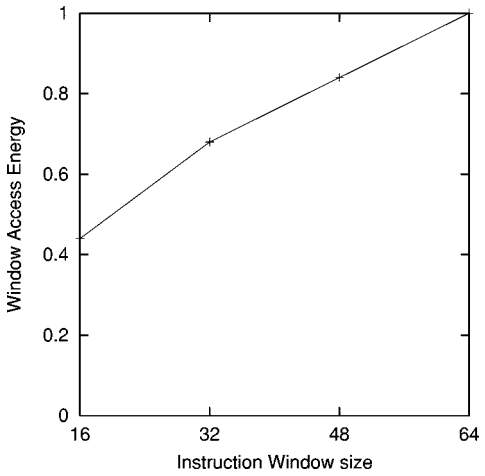

Fig. 9. Window access energy.

to an overall increase in the energy-efficiency of the machine. When execution goes out of the hotspot, the processor is switched back to the default configuration.

C. Energy Impact of Smaller Structures

When we change the active size of any RAM or CAM, we can reduce access capacitances by using segmented bitlines to turn off the unused cells of the array. Fig. 8 illustrates this idea for an array of size 16 with 4-bit long words; for simplicity the decoder array and the sense amplifiers are not shown here. This array is divided into four segments of four words each, and hence, uses three enable-lines, en1–en3. Each enable line needs to drive a transmission gate containing one NMOS and one PMOS transistor; since we need both true and complemented versions of the enable signal, we need a *pair* of physical enable lines for each logical enable line. For example, when the active size of this array is eight, the first enable-line en1 is turned on and en2 and en3 are turned off. This method can also be used in the instruction issue logic of a superscalar processor. Changing the size of the issue window from 64 to 16 can be done if we divide the array into four segments and enable only the first. The variation in the per-access energies of the issue stage (reservation station, wakeup, and select logic) as a function of the instruction window size are shown in Fig. 9.

Instead of using segmented bitlines, we could also organize arrays into several banks and selectively enable banks to implement dynamic sizing.

D. Selective Dynamic Voltage Scaling

Buffered lines in array structures can be used to selectively enable some parts of the structure and disable others. Thus, scaling down the resources of a processor can reduce the critical path delay since the rename and window access stages (which determine the critical path to a large extent) have latencies highly dependent on the instruction issue width W and the RUU size R. The following equations derived in [18] show the relationship:

$$\text{Rename logic delay} = c_0 + c_1 W + c_2 W^2 \quad (1)$$

$$d_{\text{wakeup}} = (c_1 + c_2 W) R + \left(c_3 + c_4 W + c_5 W^2\right) R^2 \quad (2)$$

$$d_{\text{select}} = c_0 + c_1 \log_4 W \quad (3)$$

$$\text{Issue logic delay} = d_{\text{wakeup}} + d_{\text{select}}. \quad (4)$$

We can exploit this to dynamically scale the operating voltage while keeping the clock frequency constant. Delays in some structures scale better than others and some delays do not scale at all. The structures that scale well could be powered by dynamic supply voltages. This would necessitate the use of level-shifters to pass data between different stages which operate at different voltages.

The dependence of path delay on supply voltage is given by the following equation [19]:

$$D \propto \frac{V_{dd}}{(V_{dd} - V_t)^2} \quad (5)$$

If D_0 is the delay of a structure in the default configuration, D is the delay after scaling, and D_l is the delay introduced by the level shifter logic, then the relationship between supply voltage and delays is given by the following equation:

$$\frac{D}{D_0 - D_l} = \frac{(V_{dd(\text{new})} - V_t)^2}{(V_{dd} - V_t)^2} \frac{V_{dd}}{V_{dd(\text{new})}}. \quad (6)$$

In practice, since supply voltages cannot be varied on a continuous scale, the implementation should consist of a few supply voltage rails with logic for switching between them as and when delays reduce to appropriate values.

When the processor goes from its highest configuration we tested (RUU size of 64 and issue width of 8) to the lowest (RUU size of 16 and issue width of 4), the delay in issue logic reduced by about 60% (for instance from 3369 ps to 1995 ps in a 0.8 μm technology). If the supply voltage was 5 V to start with, scaling to the lowest configuration now allows the issue logic to run at 3.6 V. Assuming that energy dissipation is proportional to CV_{dd}^2, the savings in energy dissipated in the issue logic amount to about 48%.

IV. Implementation and Results

A. Implementation on Simplescalar

The above ideas were implemented on the Simplescalar architecture [20]. Simplescalar is a popular industrial-strength simulator that implements a derivative of the MIPS-IV instruction set

TABLE II
BASELINE CONFIGURATION USED FOR OUR EXPERIMENTS

Processor Core	
RUU size	64 instructions
LSQ size	32 instructions
Fetch width	8 instructions/cycle
Decode width	8 instructions/cycle
Issue width	8 instructions/cycle
Commit width	8 instructions/cycle
Functional units	4 integer ALUs
	2 integer multiply/divide units
	2 FP ALUs
	2 FP multiply/divide units
Branch Prediction	
Predictor	Bimodal, 2K table
BTB	2048 entry, 4-way
Return addr stack	8 entry
Mispredict penalty	3 cycles
Memory Hierarchy	
L1 D-cache	64 KB 4-way LRU
	64B blocks, 1 cycle latency
L1 I-cache	64 KB 2-way LRU
	64B blocks, 1 cycle latency
L2 cache	256 KB 4-way LRU
	64B blocks, 6 cycles latency
Memory latency	18 cycles

and has various configuration options including a superscalar out-of-order simulator that we used for our experiments. The power modeling we used to report power figures was based on Wattch [7], 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). Unused units are modeled as consuming 10% of their active power in the idle state; this is a conservative model for low-feature sizes of modern technologies. Wattch also uses the scheme implemented in Cacti [21] for optimizing caches and cache-like structures based on delay analysis.

In accordance with the existing implementation of Simplescalar, the additional structures and options we introduced in the simulator are set through command line options and their power overhead is included in the total power estimates. The baseline configuration of the processor we used for our tests is given in Table II. This is similar to a typical high-performance processor with out-of-order execution. The configuration of the processor with the profiling hardware included is shown in Fig. 10.

B. Maintaining Performance Levels

While resource scaling helps to operate the processor in an energy-optimal mode, scaling down the effective pipeline width during execution does lead to a fall in performance. A performance monitoring counter along with the profiling hardware can restrict this performance hit to acceptable levels. After hotspot detection, while we evaluate the energy usage of each configuration, the performance counter keeps track of the number of

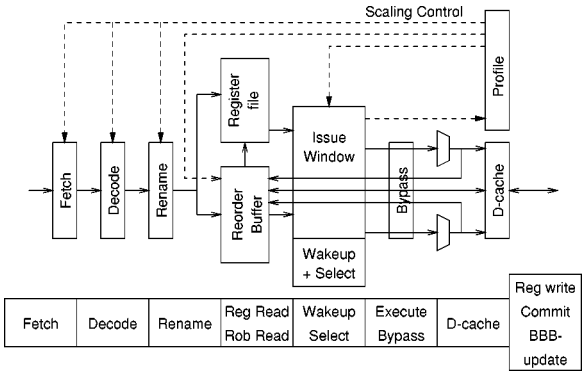

Fig. 10. The processor model with profiling hardware included.

cycles needed for the execution of 1024 instructions in each configuration, thus providing a rough CPI estimate. The acceptable performance hit we defined for our experiments was one-eighth (12.5%). (In particular, this figure was chosen because dividing by 8 can be done by a simple 3-bit shift operation.) If a particular configuration takes more than 12.5% cycles above the baseline configuration, it is rejected. This ensures that for each hotspot detected, the performance hit is not more than 12.5%; hence, the overall performance hit for the application will be less than 12.5%.

Measuring CPI by counting the clock cycles needed for a fixed number of instructions has its caveats. We have found that in the event of an instruction cache miss, the number of cycles counted goes up inordinately and this distorts the CPI figures so that configurations which are feasible in the long run are sometimes left out of consideration. To minimize the chances of this, we discount the cycles spent waiting on a cache miss. This technique gives us a more realistic (though not completely accurate) estimate of the CPI which we could have obtained, had the cache access not resulted in a miss. It should be noted that cache misses do not distort the power estimates since these estimates are determined only by usage of individual units of the processor.

C. Experimental Results

We performed experiments with programs from the Spec95 CPU benchmark suite as well as the MediaBench suite [22]. For the Spec benchmarks, the "reference" workloads supplied by Spec were simulated and the figures given are for complete execution of the application. For the MediaBench applications, we chose and ran appropriate input vectors; for example we used the popular *Lena* image for testing *epic* image compression, and used a short 320×200 movie clip, 68 frames long for testing *mpeg2* decoding. We tested the *pegwit* encryption program using a few paragraphs of random text.

The power and energy savings we obtained are shown in Figs. 11 and 12. In Fig. 11, there are four values indicated for each application. The *Dyn* value represents the power obtained with our dynamic resource scaling scheme assuming a 10% energy overhead for unused units, normalized to the power consumption of the base processor model. The *Ideal* value represents the normalized power consumption assuming no overhead for unused units; while this is becoming increasingly difficult to achieve with smaller technologies, this figure is given only

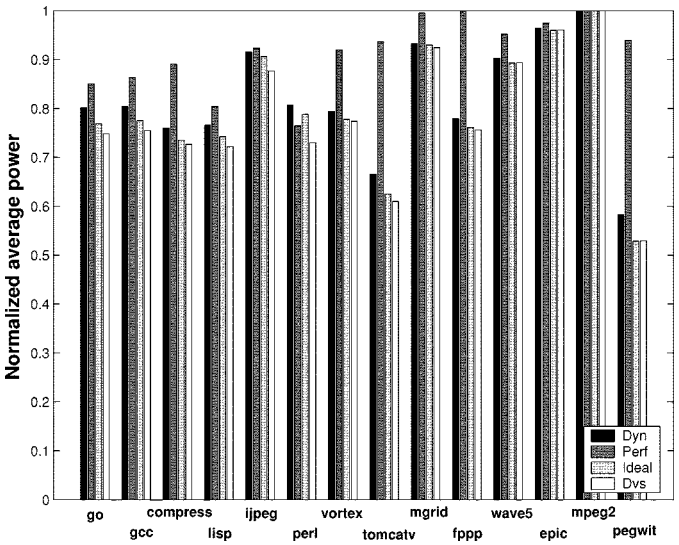

Fig. 11. Variation in power.

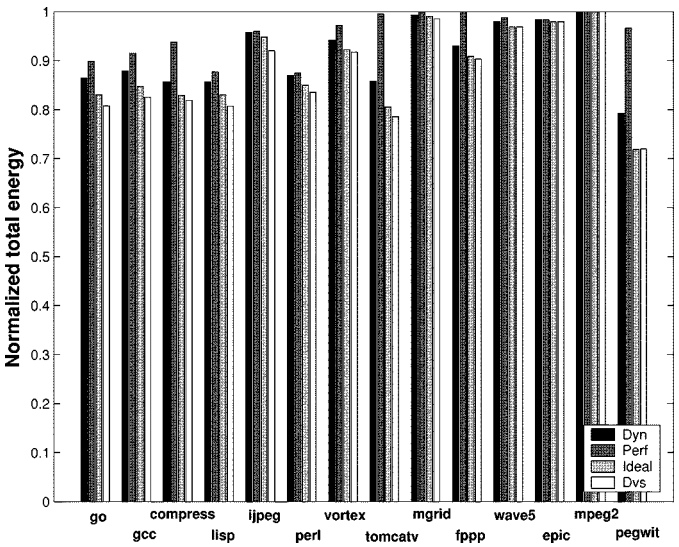

Fig. 12. Variation in energy.

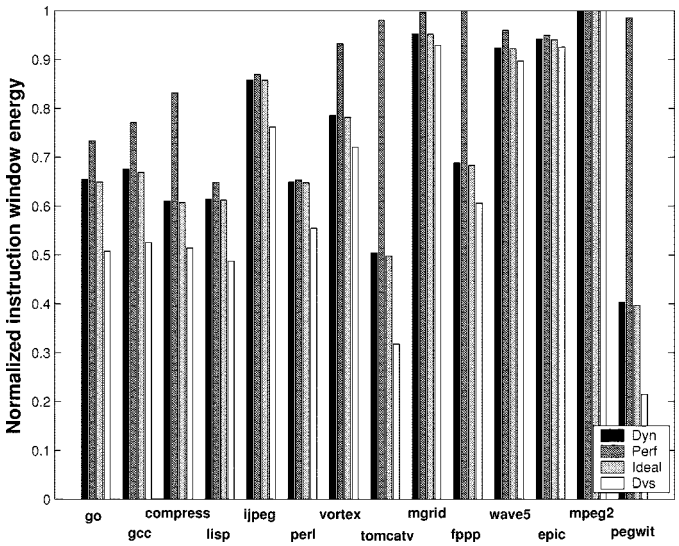

Fig. 13. Variation in instruction window energy.

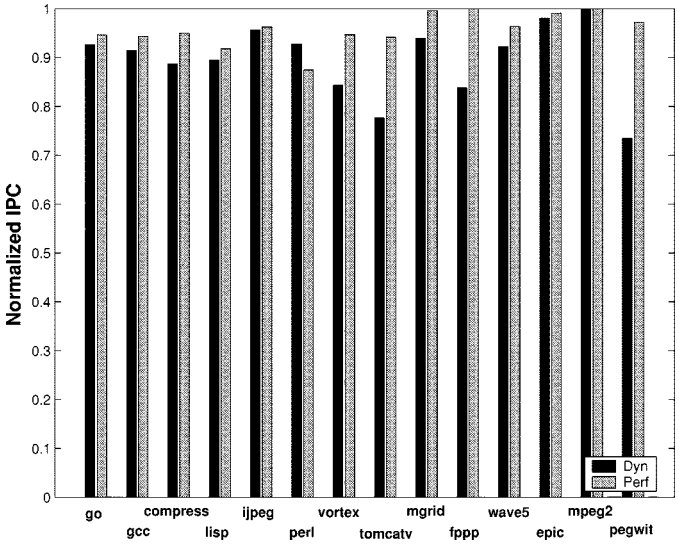

Fig. 14. Variation in performance.

to provide an indication of the potential savings possible with a circuit design aggressively optimized for leakage. The figure marked *Perf* represents the power obtained with the constraint that the performance hit does not exceed 12.5%. The figure marked *Dvs* shows the power obtained with microarchitecture resource scaling combined with dynamic voltage scaling applied to the instruction window alone. The legend for all the other graphs is identical. For the benchmarks we ran, we obtain an average power saving of 18% and an average saving in the total energy of 8% when compared to the base case. With dynamic voltage scaling, the average power saving is 21% and the energy saving is 12%. The instruction issue window energy consumption is shown in Fig. 13. Across the benchmarks, we obtain an average savings of 26% in the instruction window energy; with dynamic voltage scaling, we obtain 36% saving. The performance of the processor for various benchmarks using resource scaling with and without the constraint on performance is shown in Fig. 14. The average performance hit is lower for integer benchmarks than for floating point benchmarks. With the constraint on performance hit, the drop in performance of

TABLE III
SAVINGS OBTAINED USING RUN-TIME RESOURCE SCALING

	Avg %	Max %
Total Energy Savings		
With 10% overhead	8.84	20.77
Ideal	11.24	28.14
Perf constrained	4.54	12.56
With DVS	12.32	28.00
Average Power Savings		
With 10% overhead	18.06	41.76
Ideal	20.08	47.18
Perf constrained	8.49	23.55
With DVS	21.40	47.08
Instruction Window Energy Savings		
With 10% overhead	26.73	59.68
Ideal	27.03	60.32
Perf constrained	12.07	35.19
With DVS	36.00	78.56

all the benchmarks is bounded. Table III summarizes the results and shows comparative percentage savings in total energy and average-power consumption.

The characteristics of each application have to be taken into account while interpreting the results. For example, the *mpeg2* benchmark shows no change in any parameter. This is because the entire execution time of the mpeg decoder is spent inside one hotspot; the optimum configuration determined for this hotspot is the same as the default configuration, so the saving obtained is zero. The *pegwit* benchmark shows a large potential for energy reduction with a corresponding tradeoff in performance; it is in such applications that the performance hit constraint comes in useful. In general the integer applications we ran (the first seven benchmarks in the graphs) showed more energy reductions than the floating point benchmarks (the next four bars in the graphs) and the MediaBench applications (the last three bars) with lower drops in performance.

V. Practical Considerations

A. Performance Overhead of Switching Configurations

Many parts of the processor are implemented as circular queues using *head* and *tail* pointers, eg. instruction issue queue, load-store queue, etc. Each configurable unit has a maximum size (physical capacity) and an active size (fraction of units which are enabled, determined at runtime). The processor is said to switch configurations when the active size of any unit changes.

Whenever a decision is made to change the configuration of the processor (say to reduce the instruction window size from 64 to 32 or to reduce the pipeline width from 6 to 4) a flag is set and the dispatch unit stops pumping instructions into the execution queue. The instructions already in the queue are allowed to run to completion; after they are committed, the active sizes of the reconfigured units are changed. The exact loss of CPU cycles incurred by this pipeline flush done on every reconfiguration depends on the state of the processor at the instant of the switch. Our experiments have shown penalties as low as zero cycles (when the queue is nearly empty) and as high as 30 cycles (for example when the queue is nearly full, when long-latency instructions are already in pipeline, or when we have a cache-miss on a load). However, we do not reconfigure the processor too often; in practice we find that the number of cycles lost is less than 0.5% in the worst case and much less than this in most cases. A summary of the performance overhead induced for each benchmark is given in Fig. 15.

B. Performance and Energy Overhead of Profiling Hardware

The accesses to the BBB are done after branch instructions are retired; hence, the hotspot detection scheme is not on the critical path of the instruction flow and does not bring about any delay overhead. The BBB hardware is activated only once every branch instruction; hence, the power overhead is also quite small. Fig. 16 gives the energy overheads for the profiling hardware for each application we tested. The profiling hardware consumes an average of 0.2% of the total energy across the applications we tested; the maximum was 0.45%.

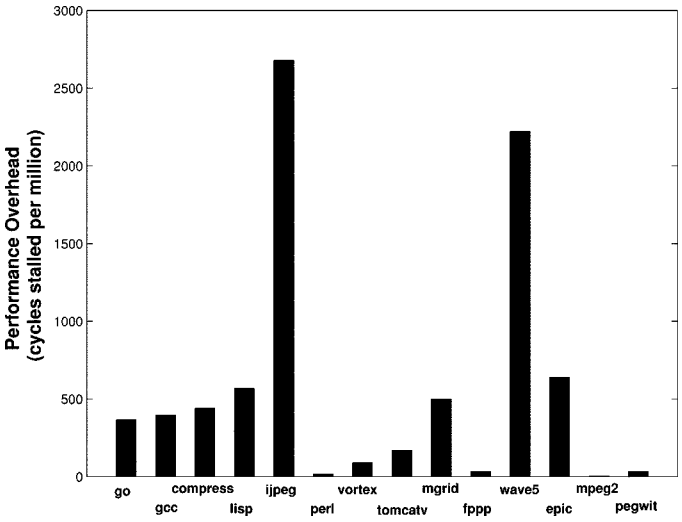

Fig. 15. Performance overheads of profiling hardware.

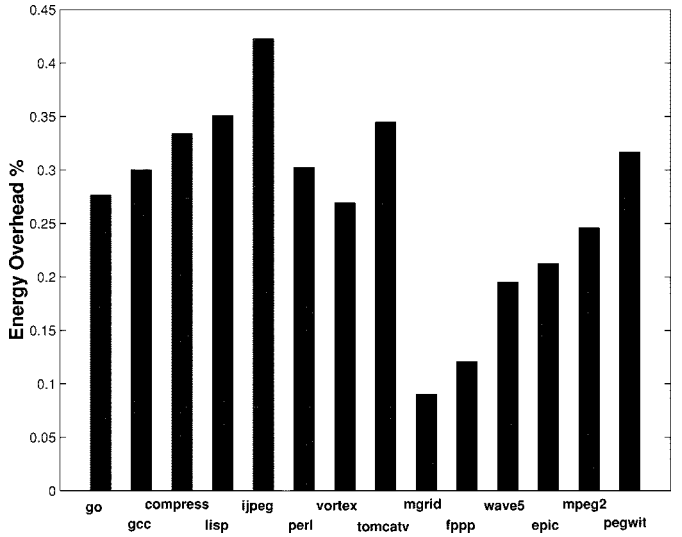

Fig. 16. Percentage energy overheads of the profiling hardware.

C. Comparison With Static-Throttling Methods

Many power-management methods work by reducing the frequency of operation of the chip at run-time. Such static clock throttling methods do not reduce the net energy consumption for a particular task; they only serve to spread out the consumption of the same amount of energy over a longer time period. The reduction in power is then exactly equal to the reduction in clock frequency. Our approach is better, since for a given penalty in performance (which we could restrict to acceptable levels) we obtain a net savings in the total energy consumption. Other approaches have been suggested which throttle the flow of instructions from the I-cache. Fig. 17 shows a graph comparing microarchitecture scaling scheme with static-throttling methods, namely static I-cache throttling [23], with 2 and 4 instructions fetched per cycle and static clock throttling. The graph shows data from all the 14 benchmarks we studied; each point in the graph corresponds to one benchmark. It can be observed that for given values of energy reduction achieved, our method provides significantly lesser delay for all the benchmarks than any of the static throttling methods do.

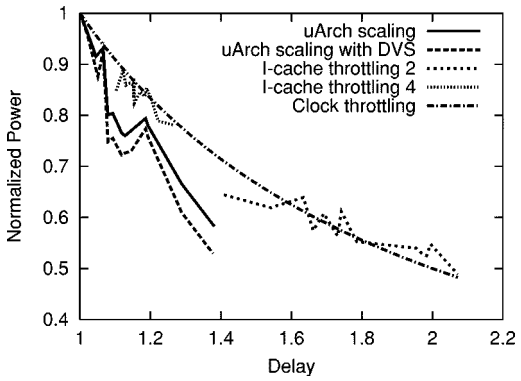

Fig. 17. Delay versus power comparison chart.

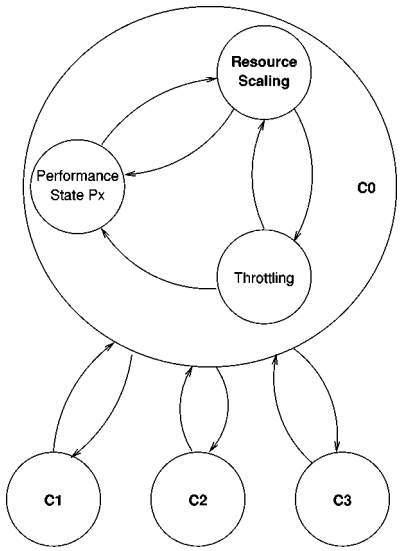

Fig. 18. Resource scaling in the context of ACPI.

D. Power Management Strategies

The dynamic reconfiguration process described above can be used in the context of comprehensive power management strategies like the advanced configuration and power interface (ACPI) model [4]. Briefly, the ACPI standard for processor power management defines one functional state C0 (in which the processor executes instructions) and three sleeping states C1, C2, and C3 (different low-power modes in which the processor does not execute any instructions). Inside the state C0, the standard also specifies various degrees of clock throttling, by which the processor's clock is stopped for different periods of time. While this mechanism directly provides a large reduction in power, it does so at the cost of a correspondingly large hit in performance.

The run-time resource scaling methods presented in this paper can be used as an *additional* stage in the active mode of operation C0, as shown in Fig. 18. This has the advantage of providing power savings with less performance overhead than is associated with clock throttling. as shown in Fig. 17.

VI. Conclusion

Using hardware structure for code profiling enables detection of program hotspots. Most applications have several such hotspots and spend a significant proportion of their execution time within them. Optimizing the processor configuration for each hotspot leads to an optimal overall execution profile, providing good reduction in energy dissipation. This can be exploited in mobile computing systems as well as in conventional systems for thermal management. Our presented approach allows fine-grained power management at the processor level based on the characteristics of the running application.

References

[1] L. Benini and G. de Micheli, "System-level power optimization: Techniques and tools," in *Proc. Int. Symp. Low-Power Electronics Design*, San Diego, CA, Aug. 1999, pp. 288–293.

[2] N. Bellas, I. Hajj, and C. Polychronopoulos, "Using dynamic cache management techniques to reduce energy in a high-performance processor," in *Proc. Int. Symp. Low-Power Electronics Design*, San Diego, CA, Aug. 1999, pp. 64–69.

[3] K. Ghose and M. B. Kamble, "Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation," in *Proc. Int. Symp. Low-Power Electronics and Design*, San Diego, CA, Aug. 1999, pp. 70–75.

[4] "Advanced Configuration and Power Interface Specification," Compaq, Intel, Microsoft, Phoenix and Toshiba, http://www.acpi.info/spec.htm, 2000.

[5] T. Pering and R. Brodersen, "The simulation and evaluation of dynamic voltage scaling algorithms," in *Proc. Int. Symp. Low-Power Electronics Design*, Monterey, CA, Aug. 1998, pp. 76–81.

[6] A. Klaiber, *The Technology Behind Crusoe Processors*. Santa Clara, CA: Transmeta Corp., 2000.

[7] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *Proc. Int. Symp. Comput. Architecture (ISCA)*, Vancouver, Canada, May 2000, pp. 83–94.

[8] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "The design and use of simplepower: A cycle-accurate energy estimation tool," in *Proc. 37th Design Automation Conf.*, May 2000, pp. 95–106.

[9] V. Tiwari, S. Malik, A. Wolfe, and M. Lee, "Instruction level power analysis and optimization of software," *J. VLSI Signal Processing*, vol. 13, no. 1–2, 1996.

[10] S. Manne, A. Klauser, and D. Grunwald, "Pipeline gating: Speculation control for energy reduction," in *Proc. Int. Symp. Computer. Architecture (ISCA)*, Barcelona, Spain, May 1998, pp. 132–141.

[11] D. Marculescu, "Profile driven code execution for low power dissipation," in *Proc. Int. Symp. Low-Power Electronics Design*, Rapallo, Portofino, Spain, Aug. 2000, pp. 253–255.

[12] V. Zyuban and P. Kogge, "Optimization of high-performance superscalar architectures for energy-delay product," in *Proc. Int. Symp. Low-Power Electronics Design*, Rapallo, Portofino, Spain, Aug. 2000, pp. 84–89.

[13] D. H. Albonesi, "Dynamic ipc/clock rate optimization," in *Proc. Int. Symp. Comput. Architecture (ISCA)*, Barcelona, Spain, May 1998, pp. 282–292.

[14] R. I. Bahar and S. Manne, "Power and energy reduction via pipeline balancing," in *Proc. Int. Symp. Comput. Architecture*, Goteborg, Sweden, May 2001, pp. 218–229.

[15] D. Folegnani and A. Gonzalez, "Energy-effective issue logic," in *Proc. Int. Symp. Comput. Architecture (ISCA)*, Goteborg, Sweden, May 2001, pp. 230–239.

[16] S. Ghiasi, J. Casmira, and D. Grunwald, "Using IPC variation in workloads with externally specified rates to reduce power consumption," in Proc. Workshop Complexity Effective Design, Vancouver, Canada, May 2000.

[17] M. C. Merten, A. R. Trick, C. N. George, J. C. Gyllenhaal, and W. W. Hwu, "A hardware-driven profiling scheme for identifying program hotspots to support runtime optimization," in *Proc. Int. Symp. Comput. Architecture (ISCA)*, Atlanta, GA, May 1999, pp. 136–148.

[18] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Quantifying the Complexity of Superscalar Processors," University of Wisconsin-Madison, Dept. Comput. Sci. , Tech. Rep. 1328, 1996.

[19] K. Usami and M. Horowitz, "Clustered voltage scaling technique for low-power design," in *Proc. Workshop Low-Power Design*, Monterey, CA, Aug. 1995, pp. 3–8.

[20] D. Burger and T. M. Austin, "The Simplescalar Tool Set, Version 2.0," Univ. Wisconsin-Madison, Dept. Comput. Sci., Tech. Rep. 1342, 1997.

[21] S. J. E. Wilton and N. P. Jouppi, "An Enhanced Access and Cycle Time Model for On-Chip Caches," Western Res. Lab., DEC, Tech. Rep. 93/5, 1994.

[22] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proc. Int. Symp. Microarchitecture (Micro)*, Research Triangle Park, NC, Dec. 1997, pp. 330–335.

[23] H. Sanchez, B. Kuttanna, T. Olson, M. Alexander, G. Gerosa, R. Philip, and J. Alvarez, "Thermal Management System for High Performance PowerPC Microprocessors," in *Proc. IEEE CompCon*, San Jose, CA, Feb. 1997, pp. 325–330.

Anoop Iyer (S'00–M'02) received the Bachelors degree in electrical engineering from the Indian Institute of Technology, Bombay, and the Masters degree in electrical and computer engineering from Carnegie Mellon University, Pittsburgh, PA, in 2000 and 2002, respectively.

He is currently with AMD, Austin, TX. His research interests are in computer architecture, low-power computing, and VLSI systems.

Diana Marculescu (S'94–M'98) received the M.S. degree in computer science from the Polytechnic Institute of Bucharest, Romania, and the Ph.D. degree in computer engineering from the University of Southern California, Los Angeles, CA, in 1991 and 1998, respectively.

She is currently an Assistant Professor in the Department of Electrical and Computer Engineering at Carnegie Mellon University, Pittsburgh, PA. Her research interests include energy aware computing, VLSI, computer architecture, and CAD for power modeling and estimation.

Dr. Marculescu received the NSF Career Award in 2000. She is a Member of the organizing committee of the ACM International Symposium on Low-Power Electronics and Design and the IEEE/ACM International Workshop on Logic and Synthesis. She also serves on the technical program committee of several conferences, including the IEEE ACM International Conference on Computer-Aided Design, and IEEE Design, Automation, and Test in Europe.