# Power Aware Microarchitecture Resource Scaling

Anoop Iyer Diana Marculescu

Department of Electrical and Computer Engineering
Center for Electronic Design Automation
Carnegie Mellon University
Pittsburgh PA 15213 USA

## 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 changes according to the parallelism in the running program. Experiments on some benchmark programs show good savings in total energy consumption; we have observed a decrease of up to 23% in energy/cycle and up to 8% 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.*

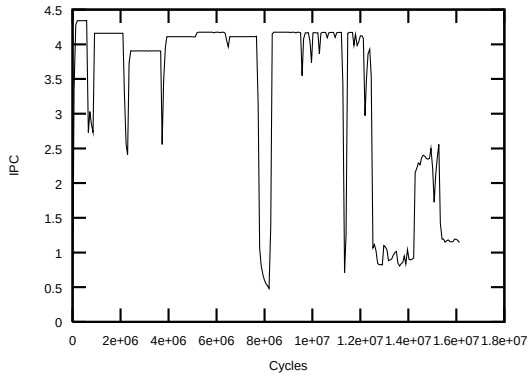


**Figure 1. Execution profile of the epic benchmark**

## 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.

### 1.1. 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]. The various approaches that have been proposed are based on memory hierarchy [2, 3], dynamic power management [4], dynamic supply voltage variation [5, 6], etc.

So far only a few microarchitectural level solutions to the power problem have been proposed; for example [7] proposes a technique that uses confidence estimation to gate the execution of branches that are most likely to be mispredicted, and [8] presents a new paradigm for adapting the execution of application programs for low power using profiling. [9] presents an analysis of different configurations of superscalar processors and derives the optimal "envelope" for energy-delay product; but their approach is not adaptive. The work presented in [10] uses 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.

### 1.2. 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 [8, 11] that there is wide variation in processor resource usage among various applications. In addition, the execution profile of most applications (for example the profile of the *epic* benchmark shown in Figure 1) indicate that there is also wide variation in resource usage from one section of an application's code to another.

The quantity and configuration of the processor's resources will also affect the overall execution profile and the energy consumption. Figure 2 shows the variation of the total energy consumption 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 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. Identifying the right configuration which optimizes the energy consumed per instruction for each region of code is the goal of our work.

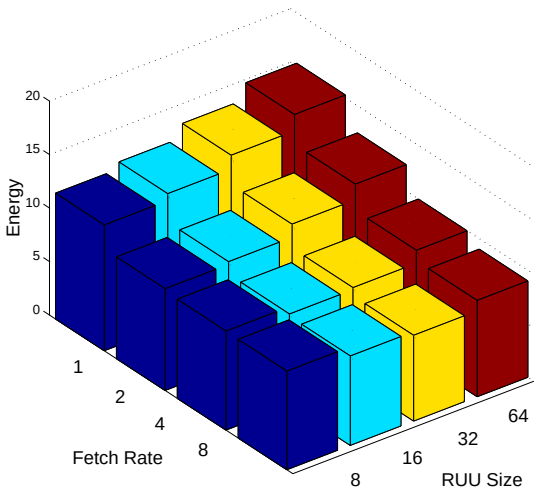


**Figure 2. Energy variation of lisp**

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 and tradeoffs for each region and allocate 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.

### 1.3. Organization of this Paper

The rest of this paper is organized as follows. Section 2 presents the framework needed for hotspot detection. We present in section 3 the methodology for finding the optimal configuration. In section 4 we discuss some practical considerations. Section 5 contains details of our implementation and results on a set of benchmarks. In section 6 we conclude with some final remarks.

## 2. 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 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. These regions or hotspots consist of a number of basic blocks exhibiting strong temporal locality.

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 have implemented a modified version of the scheme proposed by Merten et al [11].

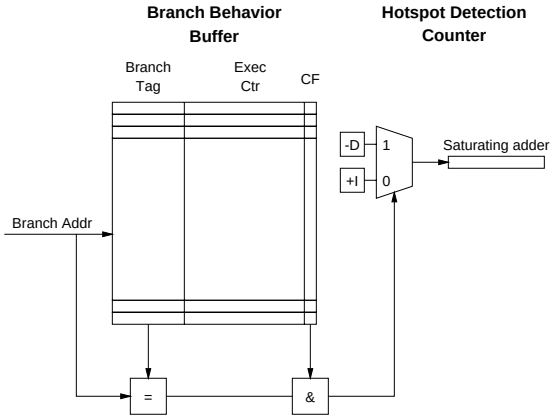


**Figure 3. Hotspot detection hardware**

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 one-bit candidate flag. The execution counter (9 bits wide as suggested in [11]) 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. 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 D, and each time a non-candidate branch is taken, it is incremented by I. When the HDC decrements down to zero, we are in a hotspot. The BBB and HDC are left running even when execution is inside the hotspot. When the code strays away from the hotspot, non-candidate 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 refreshing and flushing of the BBB, the replacement policy for BBB entries, etc. were all implemented as described in [11]. The replacement policy 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. Every 4096 cycles, BBB entries which are non-candidate entries are flushed. Every 64K 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. One possible implementation of the BBB entry is shown in Figure 4.

The changes in our scheme from previous versions of this hotspot detection scheme are twofold. In the BBB, we use the *target* address of the branch instruction (or the starting address of the basic block) to index the table, and not the address of the branch instruction itself. While the scheme proposed in [11] used a separate structure called a monitor table to detect that execution has strayed away from a hotspot, we use the same BBB and HDC structures to achieve this end since our scheme does not need the extra functionality offered by the monitor table. As mentioned above, straying of code away from hotspots is detected by keeping the profiling hardware running at all times and by waiting for the HDC to increment to its maximum value.

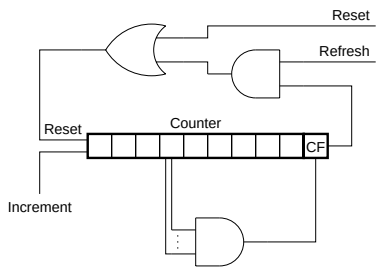


**Figure 4. Implementation of one BBB entry**

| 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**

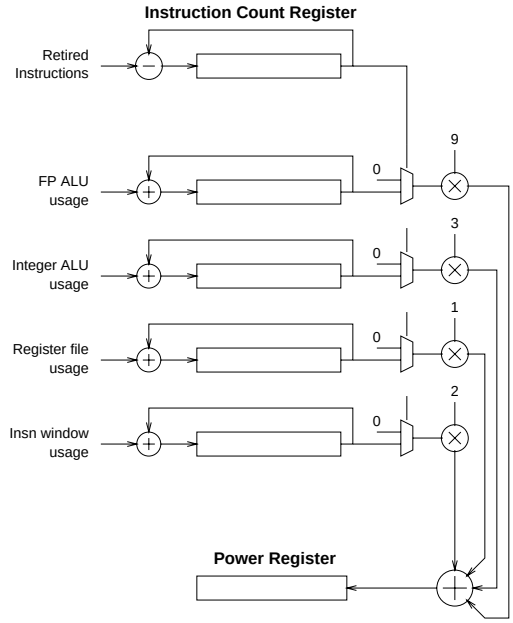


**Figure 5. Power profiling hardware**

# 3. The 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 the parameters under control, which for our experiments were the RUU size and the effective pipeline width. To have a consistent flow of instructions through the pipeline, the decode width, issue width and commit width 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.*

## 3.1. Power 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 [12], 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 per-access energy 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 5. We point out that depending on the implementation, the four 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.

## 3.2. Optimizing the Configuration

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. During each cycle, it 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.

After every 1024 instructions, the FSM reads the power register for an estimate of the power consumed and switches to a new processor configuration. 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 to test power usage. 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 Figure 6.

After the optimum configuration is found, it is stored inside the hotspot table. The table contains one entry for each hotspot which stores the RUU size and fetch rate which have been found to be optimal for that hotspot. The next time the same hotspot is encountered, the optimal values can be taken from the table. This does not lead to much overhead since the size of the table is only 16 entries. In practice, in the programs we have tested, the number of distinct hotspots was less than 16.

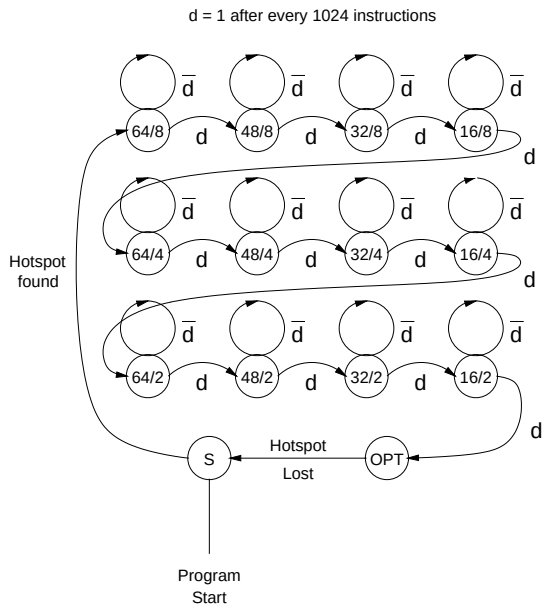


**Figure 6. FSM Example**

# 4. Practical Considerations

## 4.1. 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 fetch rate 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 less than this in most cases.

## 4.2. Performance and Power Overhead of Profiling Hardware

The accesses to the BBB are done after the branch instructions are retired; hence the hotspot detection scheme is not in the critical path of the processor and does not bring about any delay overhead. The profiling hardware is activated only once every branch instruction; hence the power overhead is also quite small. For example in a typical run of *gcc* on Wattch [12] the BBB power is found to be 0.05W out of a total power of 20.9W.

## 4.3. Subbanking in the I-Cache

When the fetch width is scaled down, the required line size of the instruction cache also changes. In this case and (also in general) the instruction fetch stage may not be able to utilize all the words available in one block. Accesses to the instruction cache can be optimized using subbanking methods described in [3]. By using an array of bit flags to indicate whether a particular word in a line should be fetched or not, the array access stage can be programmed to selectively read out words from the cache. This leads to a significant saving in the instruction cache power. This fine-grained scaling of line size is in agreement with the methodology of run-time resource scaling.

## 4.4. 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 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 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 if the cache miss had not happened. 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.

## 4.5. 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 rate and the RUU size [13]. 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 which 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 [14]:

$$D \propto \frac{V_{dd}}{(V_{dd} - V_t)^2} \quad (1)$$

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:

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

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.

The delays of various structures inside a typical superscalar processor have been studied by Palacharla et al [13]. They have shown that to a good approximation, the delay of the rename logic is linear in the issue width of the processor and the delay of the issue logic is quadratic in issue width as well as in RUU size. 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 reduces from 3369 ps to 1995 ps on a 0.8$\mu$m technology. If the supply voltage was 5V to start with, scaling to the lowest configuration now allows the issue logic to run at 3.6V. Assuming that energy dissipation is proportional to $CV_{dd}^2$, the savings in energy dissipated in the issue logic amount to about 48%.

# 5. Implementation and Results

## 5.1. Implementation on SimpleScalar

The above ideas were implemented on the Simplescalar architecture [15]. Simplescalar is a popular industrial-strength simulator 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. The power modeling we used to report power figures was based on Wattch [12], 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 good model for low feature sizes of modern technologies. Wattch also uses the scheme implemented in Cacti [16] for optimizing caches and cache-like structures based on delay analysis.

In keeping 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 2. The schematic of the processor with the profiling hardware included is shown in Figure 7.

| **Processor Core** | |
|---|---|
| RUU size | 64 instructions |
| LSQ size | 32 instructions |
| Fetch queue size | 8 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 |

**Table 2. Baseline configuration used for our experiments**

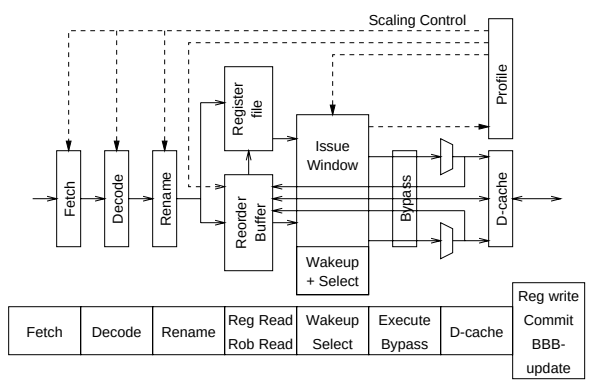


**Figure 7. The processor model with profiling hardware included**

## 5.2. Experimental Results

We performed experiments with programs from the Spec95 CPU benchmark suite as well as the MediaBench suite [17]. The power savings we obtained are summarized in Figures 8 and 9. There are three values indicated for each application: the *constrained* figure represents the power consumption of the processor with the constraint on performance not exceeding specified limits; the *unconstrained* case represents the case in which the FSM optimizes the processor for lowest energy regardless of the performance hit involved; the last case shown, the *fixed* case, is the power consumption for the baseline processor without any resource scaling. The performance for each application under each mode of execution is given in Figure 10.

For the Spec benchmarks, standard workloads 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 320x200 movie clip 68 frames long for testing *mpeg2* decoding. We tested the *pegwit* encryption program using a few paragraphs from the text of this paper. Most applications show significant savings in the average energy per cycle (average power) ranging from 2.6% to 26.3%. The savings in energy per instruction (total energy dissipated) ranges from very low values to 8%. In the case of the *tomcatv* benchmark, the energy is higher with dynamic resource scaling. This could be because the window of instructions profiled after the initial hotspot detection did not match the general execution profile, leading to a sub-optimal configuration being used.

The characteristics of each application have to be taken into account while interpreting these results. For example, most of the execution time of the *mpeg2* decoder is inside a single hotspot, and the optimal configuration derived for this hotspot by our scheme is an RUU size of 16 and a fetch rate of 8 instructions. The parallelism of this application however appears to need a configuration close to the default configuration we tested, since changing the configuration drastically away from the default produces little change in the energy consumption but significant decrease in the IPC and power figures. Thus in *mpeg2* the ALUs and execution units dominate the execution profile; changing the dynamic scheduling in the processor does not save much energy. The same is the case with other benchmarks like *ijpeg* and *epic* which show only marginal savings in energy. However in the case of *mpeg2*, since the entire 68-frame movie clip was decoded in about 72 million cycles, we can safely conclude that run-time resource scaling implemented in a real-world system running at say 500 MHz will not bring about a noticeable performance hit during movie playback.

## 5.3. Dynamic Voltage Scaling

To test the dynamic voltage scaling scheme, we implemented selective voltage scaling for the issue logic stage alone and ran the *gcc* and *go* benchmarks. The saving in energy rose from 7% to 13.5% for *gcc* and from 7.5% to 16.5% for *go* with corresponding reductions in average power. If other structures in the processor were also to incorporate dynamic voltage scaling, the savings would increase.

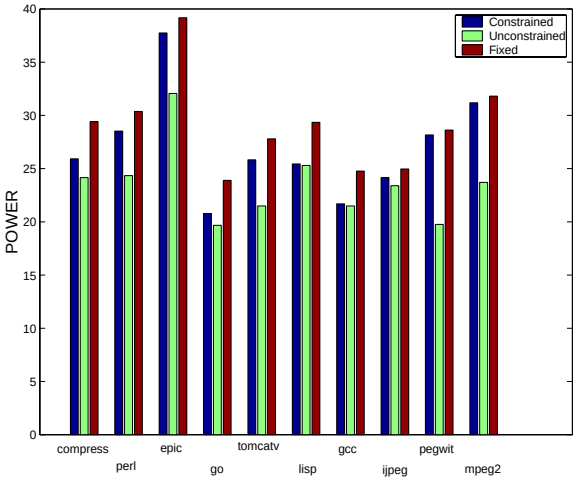


**Figure 8. Variation in power**

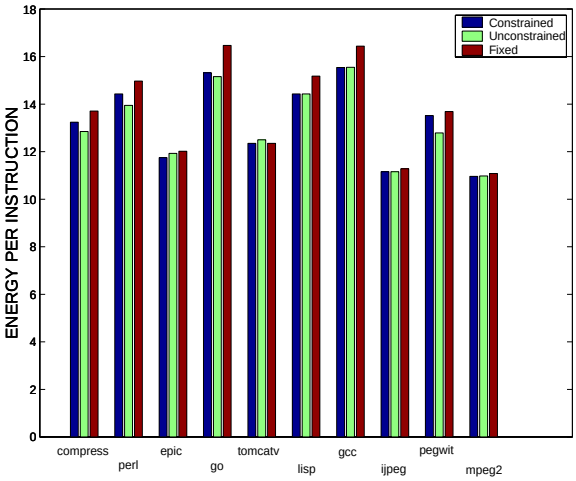


**Figure 9. Variation in energy**

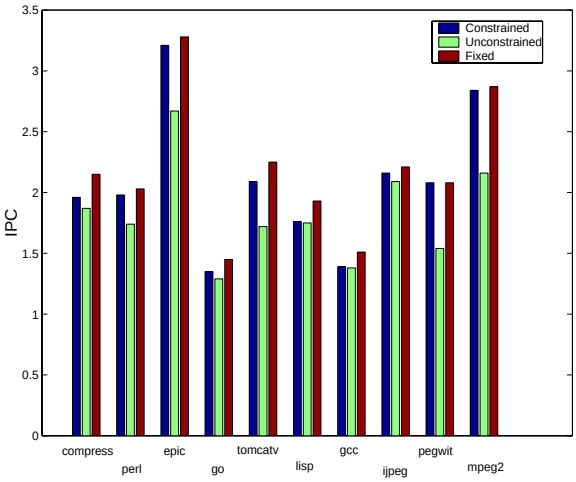


**Figure 10. Variation in performance**

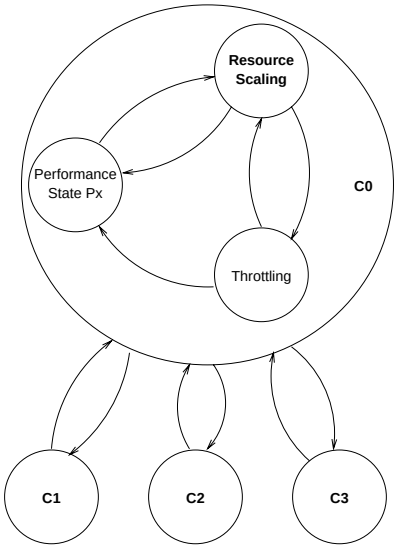


**Figure 11. Resource scaling in the context of ACPI**

## 6. Conclusion

The techniques outlined in this paper for run-time profiling of code and optimization of the processor configuration show good promise for energy savings. Further extensions could be made to our scheme. For instance, in our experiments we have coupled the fetch width, decode width and issue width to the same value; there may be more optimal ways of configuring the processor with different values for these parameters. Other resources which we have not considered in our experiments may also lend themselves to run-time scaling; for example, branch prediction tables, data and instruction caches and TLBs. Selective traversal of possible configurations could be done instead of exhaustive testing of all configurations; in fact this will become a necessity when the parameters under control increase in number. The performance monitoring hardware could be used for more 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. Microarchitecture level scaling could be incorporated as a separate state in ACPI-based power management, as shown in Figure 11. Overall, we believe that microarchitecture level resource scaling and allocation can lead to a significant saving in power 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] N. Bellas, I. Hajj, and C. Polychronopoulos, "Using Dynamic Cache Management Techniques to Reduce Energy in a High-Performance Processor," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 1999.

[3] K. Ghose and M. B. Kamble, "Reducing Power in Superscalar Processor Caches using Subbanking, Multiple Line Buffers and Bit-line Segmentation," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 1999.

[4] Compaq, Intel, Microsoft, Phoenix and Toshiba, *Advanced Configuration and Power Interface Specification*, 2000.

[5] T. Pering and R. Brodersen, "The Simulation and Evaluation of Dynamic Voltage Scaling Algorithms," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 1998.

[6] A. Klaiber, *The Technology Behind Crusoe Processors*. Transmeta Corp, January 2000.

[7] S. Manne, A. Klauser, and D. Grunwald, "Pipeline Gating: Speculation Control for Energy Reduction," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1998.

[8] D. Marculescu, "Profile driven Code Execution for Low Power Dissipation," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 2000.

[9] V. Zyuban and P. Kogge, "Optimization of High-Performance Superscalar Architectures for Energy-Delay Product," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 2000.

[10] S. Ghiasi, J. Casmira, and D. Grunwald, "Using IPC Variation in Workloads with Externally Specified Rates to Reduce Power Consumption," in *Workshop on Complexity Effective Design*, 2000.

[11] 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 *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1999.

[12] 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.

[13] 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.

[14] K. Usami and M. Horowitz, "Clustered Voltage Scaling Technique for Low-Power Design," in *Proceedings of the Workshop on Low Power Design*, 1995.

[15] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, version 2.0," Tech. Rep. 1342, University of Wisconsin-Madison, CS Department, June 1997.

[16] 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.

[17] 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.