Appears in 4th. ACM Workshop on Feedback Directed and Dynamic Optimization (FDDO-4)

Profile-Based Energy Reduction for High-Performance Processors*

Michael Huang, **Jose Renau**, and **Josep Torrellas**
Department of Computer Science
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

ABSTRACT

To reduce the energy consumption of modern processors, designers have proposed many energy-saving techniques. In many cases, these techniques are dynamically activated and deactivated. In systems that employ these techniques, to adapt to changes in application behavior, profiling can help determine how to manage the activation of techniques to improve a certain metric.

In this paper, we propose to use a global approach to such profiling. The idea is to profile the application on a section-by-section basis but to make the decisions in a global manner, competitively comparing the different sections. Under such conditions, the system can be more effective. For example, we can target for reduced energy consumption in an application without slowing it down. The results show that such an approach can reduce the energy consumption of 7 applications by 12% with negligible slowdown.

1 Introduction

In recent years, computer architecture has shifted from a performance vs. cost design to a more challenging performance vs. cost vs. energy design. Energy consumption has become a major concern for the designers of modern processors due to many reasons, including increased importance of mobile computing, high cost associated with heat dissipation systems and to a lesser extent, increasing demand for electricity.

The increasingly high speed of modern processors is an important factor in growing power consumption. Various techniques have been proposed to trade off performance for energy savings [8, 12, 16, 18, 20, 26]. These techniques have been applied in previous frameworks [9, 11, 13, 14, 21, 24, 28] to slowdown the processor mainly for three reasons: the processor is faster than needed and thus wastes energy, the system needs to save battery energy, or the temperature is too high. In all the situations, the basic idea is the same: reduce power consumption at the expense of processor performance. If the system does not have enough slack in execution time, these frameworks will not produce much energy savings without sacrificing performance. In this paper we go one step further, trying to achieve significant energy savings with negligible slowdown. To do this, we perform off-line profiling to obtain a global view of the impact of low power techniques on different code sections. Based on this information, we activate techniques in a smart manner to maximize total energy savings per unit slowdown. Moreover, though we still use techniques that on average slow down the execution, due to the non-uniformity of application's behavior, some of the techniques even speed up code sections. Therefore, it is possible to achieve zero or negligible net slowdown.

In our simulation environment, we show that this global optimization allows us to reduce the energy consumption of a set of diverse applications by an average of 12% and as much as 16% with negligible slowdown.

The rest of the paper is organized as follows: Section 2 motivates the problem considered; Section 3 describes the profiling algorithm and parameters; Section 4 discusses the evaluation environment; Section 5 evaluates the proposed solutions; Section 6 presents the related work, and we conclude in Section 7.

2 Motivation

2.1 Opportunity

It is well known that, as applications execute, they regularly go through changes in high-level parameters such as IPC (Instruction Per Cycle) or power consumption. However, we observe that they also exhibit another important type of variability. The relative impact of an architectural modification is not constant with time; instead, the resulting change in IPC or power consumption varies widely as time proceeds. This observation allows us to activate an architecture-modifying technique only during the most favorable periods, those that save that largest amount of en-

*This work was supported in part by the National Science Foundation under grants CCR-9970488, EIA-0081307, and EIA-0072102; by DARPA under grant F30602-01-C-0078; and by gifts from IBM and Intel.

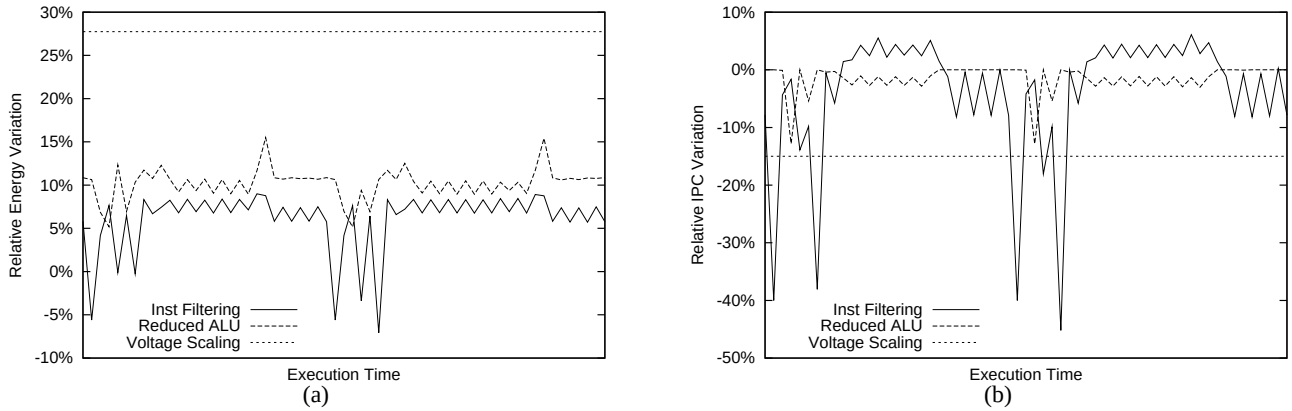

Figure 1: Impact of applying low-power techniques on the energy consumption (a) and IPC (b) as a function of time.

ergy per unit slowdown for example, and deactivate it otherwise.

To visualize this effect, we simulate a processor running one application. The simulation environment and application suite are described in Section 4. In the experiment, we apply two low-power techniques seperately, adding an instruction filter cache [18], or disabling ALUs by clock-gating. Figures 1-(a) and (b) show the relative energy consumption and the relative IPC variation, respectively, as the execution proceeds. In both figures, 0% corresponds to the base system, with none of the techniques activated. Positive numbers reflect energy reduction and better performance.

We can see from the figures that the relative impact of these techniques varies across time. These are, therefore, non-uniform techniques. To emphasize this effect, the figure also shows the impact of voltage scaling[1], a technique whose behavior is qualitatively different. In this case, the relative changes in energy consumption and in IPC remain constant across time.

Our goal in the rest of this paper is to try to exploit this variability of low-power techniques within and across applications. Our system dynamically adapts by activating and deactivating a set of techniques to save energy. Since our focus is on high-performance systems, we strive to reduce energy consumption without incurring much slowdown.

2.2 Decision Mechanism

In order to exploit the above mentioned opportunity, we need to identify the points in the code where it is advisable to adapt the system. The goal is to divide the program execution into different sections, so that each section has relatively uniform reaction to system adaptation. Also, the granularity of such sections has to be relatively coarse so that the transient state and/or any adaptation overhead becomes very small.

Many dynamic systems[3, 4, 7] constantly monitor the program and predict that the behavior in the near future is similar to the current one. This usually involves an observation interval of a fixed duration. In [25], it is shown that programs generally demonstrate periodic behavior, but the period is application-specific. Furthermore, depending on the length of the interval, programs go through different code in neighboring intervals and show different architectural metrics (e.g. IPC, cache miss rate etc.). Indeed, for short intervals, we observe a high variance in IPC among neighboring intervals in the benchmarks studied. Naturally, the optimal duration for the interval depends on the application.

A good unit of behavior repetition is the function. When the same function is executed again, its behavior is unlikely to change much. Arguably, IPC is a good indicator of high level program behavior. In the following test, we analyze the IPC variation of one program during its execution. In the case of fixed interval, we measure the standard deviation of IPC for every interval. In the case of the function, we measure the standard deviation of IPC for all invocations of the same function. Figure 2 shows that, we can predict the behavior of a future function invocation much better than we can predict the future behavior in an interval.

We see that the effectiveness of micro-architecture modifications varies during the program execution, and that we can use functions as a unit to manage techniques. In this paper, we get most out of the low-power techniques without paying a big average performance penalty.

[1]For simplicity, we assume the voltage of the whole system is scaled, not just the processor.

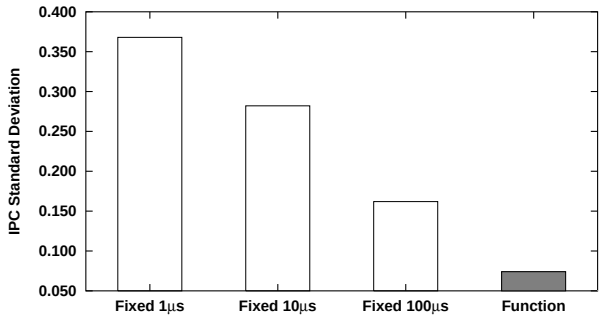

Figure 2: IPC standard deviation for different sampling methods.

3 Profiling Approach

We use profiling to determine when to activate and deactivate low power techniques. In this section, we describe important profiling parameters, including the statistics collected, the sampling interval, and the algorithms used to process the statistics.

3.1 Profiling parameters

3.1.1 Statistics collected

We need to understand the energy and performance impact of applying a technique on a section of code. Therefore, profiling should collect performance and energy statistics. Specifically, we measure cycle count and total amount of energy consumed.

The energy statistics could be obtained by reading from an energy meter, or by approximating from processor activity statistics [17]. As an example of this approximation, cache energy expenditure could be estimated by multiplying the number of accesses by the average energy per access. In this paper, for simplicity, we assume we can read the energy numbers from a meter. In Section 5.2 we also show that in the case energy statistics are not available, using only performance counters is also a viable alternative.

3.1.2 Sampling Interval

An important profiling parameter is the grain size of sampling interval. Since we need instructions to activate techniques at the beginning of each sampling interval, a very fine-grain profiling incurs a lot of overhead. On the other hand, if the grain size is too large, it will shield the phase-changes inside.

To produce a good sampling interval we break down the code into modules. A module is a piece of code that is small enough so that its reaction to low-power techniques is quite uniform, and also large enough so that the overhead of techniques activation is negligible.

We start the instrumentation process with the most frequently executed functions. We instrument at the entrance and exit of the function. Since a small portion of static code often represents most dynamic execution time, we only need to instrument a few functions.

We do not instrument inside functions that have very short execution time per invocation to avoid too much overhead. Instead, we either consider them as part of the caller function or do transformations to reduce overhead of the instrumentation. For instance, we put a wrapper around recursive functions and only instrument in the wrapper function. For a tight loop invoking a short function, we only instrument around the loop body.

Whether or not to instrument functions that have medium execution time per invocation is a trade-off. The more we instrument, the better we could adapt, but this also leads to more overhead. In most of our experiments we choose not to instrument them. In Section 5.1.2 we also evaluate this trade-off quantitatively.

For large functions, overhead of the instrumentation is negligible. However, it may not have a uniform behavior, we can break them down into smaller modules. In Section 5.1.2 we show the result of trying to exploit behavior changes insides functions.

For the profiling runs, the instrumented code will read the statistics counters. We will use these statistics to decide what techniques to apply for each module, and instrument code into the final binary accordingly.

For many applications, the final instrumented modules are functions. For generality, we still call them modules.

3.1.3 Profiling types

In order to find the effect of each technique, we need to profile the application several times, since we do not use statistical profiling. In addition to the original profiling run without low power techniques, we need one run for each technique. This requires $n + 1$ runs for each application, where n is the number of low-power techniques. Here, we assume no interference among techniques. This implies that the performance penalties and energy savings for different techniques are additive when the techniques are applied together. We call this type of profiling, *Independent* profiling.

Profiling only with individual techniques could be inaccurate if two techniques do interfere with each other. So, in another extreme, we could do what we call *Cumulative* profiling, where we run once for each possible combination of the techniques. This exponentially raises the number of profiling runs. In reality, chips will very unlikely include many techniques that target the same area for energy reduction, and techniques that target different components are largely independent of each other. Thus, the number of pro-

file runs can be reduced, by profiling in combination only those techniques that do target the same components. We refer to this kind of profiling as Semi-Independent Profiling (*Semi-Indep*).

3.2 Selection Algorithms

Once we have collected all the profiling information, we can use this information to decide which techniques to apply for every module. The algorithm for selecting techniques is straightforward. We use a metric called *efficiency score* to characterize the impact of every technique on every module. The score is calculated using Equation 1, and is a metric showing the efficiency for trading off performance for energy reduction, hence the name.

$$score = \begin{cases} -1 & \text{if } \Delta E \leq 0 \quad \text{;Wastes energy} \\ +\infty & \text{if } \Delta E > 0 \\ & \text{\& } \Delta D \leq 0 \quad \text{;Improves energy and perf.} \\ \frac{\Delta E}{\Delta D} & \textit{Otherwise} \quad \text{;Trades perf. for energy} \end{cases} \tag{1}$$

In this equation, ΔD and ΔE are the slowdown and the energy reduction respectively. Specifically, let D and E be the total delay and energy consumption of a module in the unaltered application run, and D_i, E_i be that of the module running when technique i is activated. Then $\Delta D = D_i - D$ and $\Delta E = E - E_i$. This way, in the common cases, we will be dealing with positive numbers.

We keep one entry containing ΔD, ΔE and the efficiency score for each pair of module and technique in a table. Since our goal is to maximize energy savings while minimizing slowdown, we sort the table by decreasing efficiency score. Once we have the table sorted, it can be used in different ways. To achieve our goal of saving energy without slowing down, we traverse the table top-down, scheduling all the technique/module pairs at the top of the table while keeping the sum of ΔD less than but as close as possible to zero.

When some slowdown is tolerable, we simply select more entries keeping the the sum of ΔD close to but less than this slowdown. We can also embed this information with the binary, and delay the selection till runtime when the tolerable slowdown is available.

In the case of performance-only profiling, where we do not have energy counters, we use equation 2 to calculate the score and the rest of the algorithm remains the same.

$$score = \begin{cases} +\infty & \text{if } \Delta D \leq 0 \\ \frac{D}{\Delta D} & \textit{Otherwise} \end{cases} \tag{2}$$

By using the reciprocal of relative slowdown as score we favor the cases where the technique has little negative performance impact on the module. This approximation does not give optimal scheduling of technique onto modules, for, by ignoring energy, it is possible to activate one technique with small slowdown, but with little or even negative energy savings. However, energy reduction has a strong correlation with slowdown. For example, for techniques similar to cache filtering, large relative slowdown suggests higher miss rates, which further suggests more energy consumption. Also, processor has some per cycle energy overhead such as clocking, more slowdown directly cuts into energy savings. In Section 5.2 we show that the results of using energy/performance and performance-only profile information are very close.

4 Experimental Setup

4.1 Baseline architecture

In our simulation, the baseline architecture is an out-of-order processor with two levels of caches. The architecture loosely models an IBM Power3 chip scaled to 6-issue. Table 1 lists the parameters used in the simulation. While the latency numbers in the table correspond to an unloaded machine, we model contention in the whole system in great detail. For the processor chip, we assume 0.18 μm technology operating at 1.67V.

4.2 Energy

We simulate the performance of the system using a mint-based execution driven simulator [19] that models an out-of-order processor with its memory subsystem in great detail. We port Wattch [6] to model the energy of such system. Wattch has different clock gating strategies. In all the simulations, for each functional unit, we charge a fixed amount of energy when the unit is idle. This amount is, about 10% of the energy for a typical operation performed in that unit. We enhance Wattch in the following way:

- Wattch uses a modified version of cacti [27] for cache-like structures. We developed our own extended CACTI called XCACTI. For caches of the same size, our new model produces approximately the same latency estimation as CACTI, but it gives considerably lower energy consumption results. The energy consumption is smaller mainly because we replace the Wada sense amplifier model with a more energy efficient latched sense amplifier. We also extended CACTI to model writes and reads differently. A write has a full bitline swing, while bitline swing for reads is around 15%. We calculate different energy numbers for reads, writes, line-fills, write-backs, and cache misses. It can be configured to search for configurations with the lowest delay, energy or energy-delay product. The search

Processor		Caches	Bus & Memory
Freq: 1 GHz	Branch units: 1	L1 size: 32 KB	FSB freq: 333MHz
Issue width: 6	Branch penalty: 8 cycles	L1 OC,RT: 1,3 ns	FSB width: 128 bits
Dynamic issue: yes	Return stack: 32	L1 assoc: 2way-LRU	DRAM: 2-channel Rambus
I-window size: 96	BTB entries: 2K	L1 line: 32 B	DRAM bandwidth: 3.2GB/s
Ld/St units: 2	BTB assoc: 4	L2 size: 512 KB	Mem RT: 108 ns
Int,FP units: 5,4	Predictor: GAp(10,8)	L2 OC,RT: 4,12 ns	
Pending Ld,St: 16,16		L2 assoc: 8way-Pseudo LRU	
		L2 line: 64 B	
		I-Cache: 32KB 2-way	

Table 1: Baseline configuration. OC, RT and FSB stand for occupancy, round trip latency from the processor, and front side bus respectively.

can be optimized based on several timing constraints, the expected ratio of loads and stores, and the cache miss rate.

- We developed a low level model of the DLX processor and derived energy consumption for different types of integer operation based on our spice simulation of the DLX model. We extrapolate the energy number for floating point units based on Wattch and other research [22].

- Wattch assumes a physical register file incorporated into the instruction window. This model eliminates the chance for running out of renaming register at the expense of using a larger, therefore more power hungry device. We follow the common approach of current microprocessors, and use a smaller stand alone register file and halt the decoding stage if we run out of renaming registers. This has nearly negligible performance impact for our benchmarks.

- We add DRAM energy consumption. The energy numbers are based on Intel's white paper [15]. We assume 1.2W for one memory channel operating at full bandwidth, including the overhead in memory controller.

Figure 3 shows the energy consumption breakdown for different components. The breakdown ratios are in line with numbers published by other researchers [6]. As expected, power expenditure is spread out. It is difficult for a single technique to reduce the energy consumption considerably. Thus, the processor should incorporate multiple low-power techniques that target different parts of the system.

4.3 Low-power techniques

In our framework we use several existing low-power techniques. We choose techniques that target major source of energy consumption in the processor. Since we target high performance, these techniques require little or no modification to the critical path of the processor. Notice that we do not employ voltage scaling since the activation overhead is too high to be practical for fine-grain adaption.

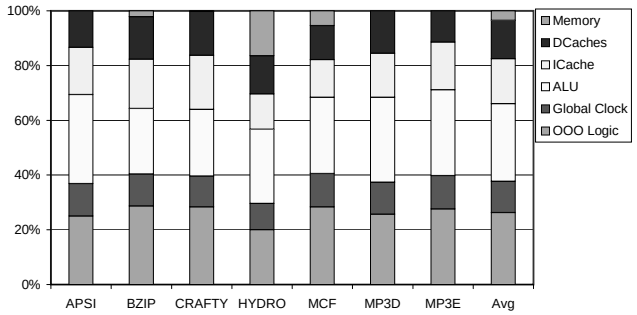

Figure 3: Energy consumption breakdown. *Memory* is the energy consumed by DRAM. *Global clock* is the clock distribution energy consumption. *OOO Logic* is the remaining parts of the processor: instruction window, register file, branch predictor, result bus, register alias table, and load/store queue.

The system is not limited to these techniques. The following sub-sections give a highlight of the techniques.

4.3.1 Instruction Filtering

We use a small filter cache [18] in the instruction memory hierarchy. The filter cache requires much less energy per access not only because it is smaller and direct-mapped, but also because it is virtually tagged, eliminating the need for a TLB check. Traditionally, the filter cache is activated all the time, trading energy for performance on average. We dynamically activate/deactivate it instead. When it is deactivated, instruction fetch goes directly to the instruction cache without checking the filter cache. We do not maintain inclusion for the filter cache and the L1 instruction cache. We only need to invalidate the filter cache when it is disabled, and the code is modified. This only happens when there is a context switch, or when the application self-modifies the

code.

The proposed filter cache is small, fast and energy efficient. It is very effective for code sections with a small footprint, but ineffective in sections with poor instruction locality. Code sections with a big footprint may even spend more energy and slow down. After balancing speed, energy consumption and chip-area, we chose a 1KB filter cache. Read access of the filter cache costs 386 compared to 2022pJ to access the I-Cache.

4.3.2 Stack Filtering

A small specialized stack cache reduces energy consumption without affecting the performance much [12]. Nevertheless, some code parts have worse energy and performance than using a normal L1 alone. In those cases we can shut down the stack cache by redirecting all the stack accesses to the L1 cache. Thus, to maintain coherence, instead of accessing L2 [12] on a stack cache miss, we access L1. Shutting down the stack cache has the overhead of writing back dirty lines, and invalidating the whole stack cache. We use a 1KB cache that can be accessed in 1 cycle as stack filter cache. Compared to 1763pJ for normal L1 accesses, a stack filter cache hit only consumes 209.

4.3.3 Phased Cache

A phased cache [10] is a set-associative cache where an access first activates all the tag arrays. If there is a match, only the correct data bank is subsequently activated, reducing the amount of bitline activity and sense amplification in the data array. Consequently, the phased cache saves energy at the cost of extra delay. Our processor has an L1 data cache with two modes of operation. In the normal mode emphasizing performance, it behaves like an ordinary cache, activating data and tag in parallel. In the low-power mode, it turns into a phased cache, activating the tags before activating the data bank. When changing to phased cache mode, we buffer the signals to the data bank for two cycles, therefore serializing the access. When restoring to normal mode, we block the cache for two cycles to drain the pipeline and stop buffering data bank signals. Reading the L1 cache in the phased cache mode, consumes 974pJ, 45% less than in the normal mode.

4.3.4 Reduced ALU

Wide issue processors tend to have many functional units to reduce structural hazard. This comes at the price of more energy spending. Seldom are all these functional units needed, and even if they are, they are not needed all the time. We divide the functional units in two clusters: master and slave. Each of the cluster consists of two floating point units, two integer units and and one load store unit. The master cluster also has a branch unit. When we reduce the number of available functional units, we clock-gate the entire slave cluster. This saves the clock distribution power, and part of the instruction issue logic power inside the slave cluster. Notice that although this technique can be implemented to reduce leakage as well, we do not address the issue in this paper. Therefore, the energy savings only come from dynamic power reduction. For multi-cycle pipelined functional unit we assume the clock-gating starts after a fixed amount of delay to allow draining of the pipeline. We assume clock distribution power to be around 15% of the average power consumption of the functional units. The actual energy savings per cycle depend on the execution speed and so on. In our simulation, it ranges from 15% increase in the energy consumption in the worst case, to about 9% savings of the average power consumption.

4.4 Applications

Our simulations are based on seven benchmarks representing a mix of multimedia, SPECint, and SPECfp benchmarks. We compile them with the IRIX MIPSPro compiler version 7.3 with -O2 optimization. Applications are simulated from beginning to end, which lasts from hundreds of millions of cycles to a billion cycles. For SPECint and SPECfp benchmarks, we reduce the input dataset. In all the cases, we verify that with the reduced data set the simulated applications produce about the same cache and TLB miss rates as the native execution with the reference data set in an R12K processor. The relative weight of each function does not change much either.

CRAFTY (186.crafty), *BZIP* (256.bzip2), and *MCF* (181.mcf) are from SPECInt 2000. *HYDRO* (104.hydro), and *APSI* (141.apsi) are from SPECfp 95. Profiling is performed with the official *train* input set.

MP3D is an MP3 decoder. We use mpg123 version 0.59r, which is one of the fastest available UNIX GPL MP3 decoders. We reproduce a high quality hifi sample. Profile training is done with a CD quality mp3 file.

MP3E is an MP3 encoder. We use lame3.85, which is fast and widely used in the MP3 community. We encode music with CD quality. We profile using a small voice file.

5 Evaluation

In our simulation, we model the performance and energy overhead for technique activation and deactivation in detail. Specifically, we model the flushing of the stack filter cache, the two-cycle bubble when deactivating the phased cache mode, and the delayed clock gating when disabling the slave cluster. We do not model the overhead for the instructions that actually activate or deactivate the techniques. We assume a single instruction writing a mask to a control register, our results show that, this represents less than 0.1%

of the total simulated instructions.

5.1 Energy/performance profiling

We start our evaluation by profiling in an somewhat idealistic situation. We assume that energy and performance data is available, the same input set is used for profiling and actual execution, and we perform *Cumulative* profiling. In the following sections, we compare the result of relaxing these restrictions. Unless otherwise stated, the target is to reduce energy consumption and *maintain the original performance*.

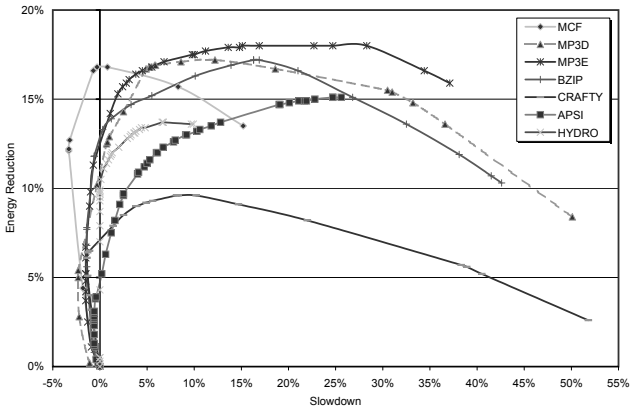

Figure 4: Energy-performance trade-off curve for applications.

Figure 4 helps to understand the trade-off between the energy reduction and the induced slowdown for our profile based approach. For each application, given a desired slowdown (or speedup), we can find the best projected energy saving from the corresponding curve for the application. We should notice two things from the curves. First, the slope differs dramatically from one end of the curve to the other, suggesting the great difference of efficiency: while we can save much energy with little slowdown at the beginning of the curve, we can only save a little energy and pay with much larger performance penalty towards the middle of the curve. In fact, the curve even bend down at the end, where the techniques start to waste energy. This suggest that we arrange carefully where and which techniques to activate. Our second observation from Figure 4 is that when comparing two schemes, looking only at the difference in energy reduction is not enough, special attention should be paid to the delay incurred, since the scheme with lower slowdown leaves more room for trading off more performance to save energy.

Throughout the rest of the section we use one set of bars per application, and an additional set labeled *Avg* for the average of all seven applications. In order to have a fair comparison, in calculating all the average slowdown, we consider any speed up for one application as zero slowdown.

5.1.1 Comparing profiling based approach with other approaches

Figure 5 shows the energy reduction and performance degradation when we apply different techniques. The four bars on the right in each group represent the effect of the four techniques applied individually and statically throughout the application run. The leftmost bar shows the result of the application with compiler inserted directives for technique activation based on profiling information. The bars in Figure 5-(b) for profiling based scheme are very small. The algorithm for obtaining these directives is described in Section 3.2.

As we can see, none of the techniques *consistently* speeds up applications *while* saving energy. Indeed, if such technique were found, it would be incorporated into the baseline architecture. The best technique, one that for unit slowdown saves the most energy, varies from application to application. The average effect of these techniques is to save some energy while slowing down a bit. However, by applying multiple techniques and using profile-based feedback, we can achieve significant energy savings without slowing down the applications.

We now compare Profiling with *Static* and *DEETM** , two different approaches trying to exploit the behavior changes among different applications or within one application.

In the first approach, *static* , we do not activate techniques dynamically. Instead, for each application, we select a set of techniques that, if applied all the time throughout the application execution, save the most energy without incurring noticeable slowdown.

The second approach, *DEETM**, is an improved DEETM [11] framework. In the DEETM framework, different techniques are calibrated off-line and ranked according to their average effectiveness across a set of benchmarks. At run-time, the framework ensures the chip-temperature does not go beyond a set limit for an extended period of time, meanwhile exploits any performance slack to save energy. This behavior is controlled by a thermal and a slack algorithm executed at fixed intervals. Here we use a scheme similar to the slack algorithm. In [11], the order of the techniques is determined by averaging results of several applications. In our improved version, we assume the system knows a priori the effect of each individual technique and uses the best technique order for each application. Many times, a technique that works well on one application does not work well for another. For instance, instruction filter cache tends to work well for MCF, speeding it up and saving energy. But its effect is detrimental for CRAFTY, where it slows down the application significantly and wastes energy. Therefore, the framework will try to activate the instruction filter cache first when running MCF, while not activating it

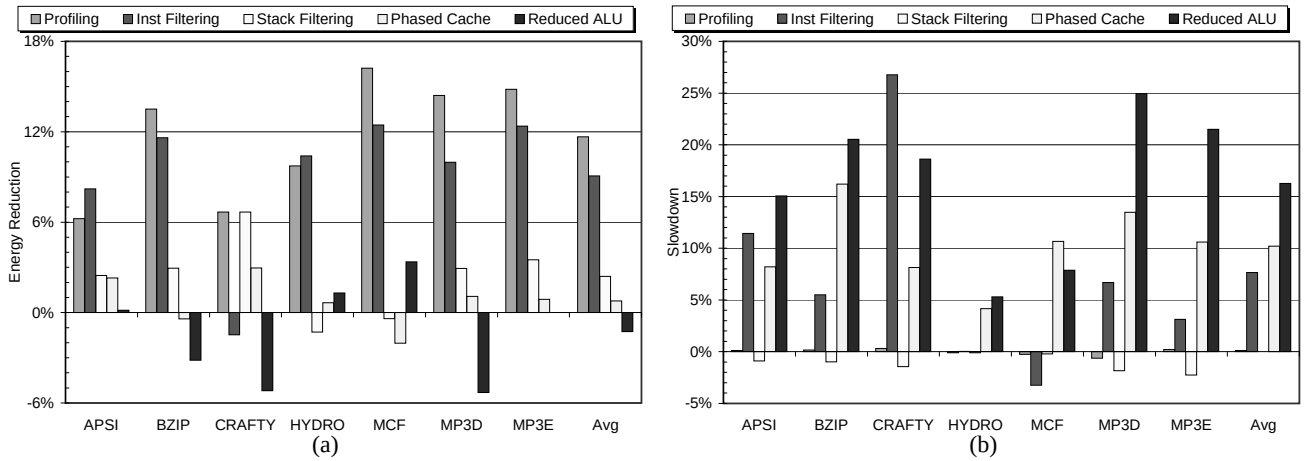

Figure 5: Effect of individual techniques and profiling based approach. (a) shows the energy reduction, and (b) shows the slowdown.

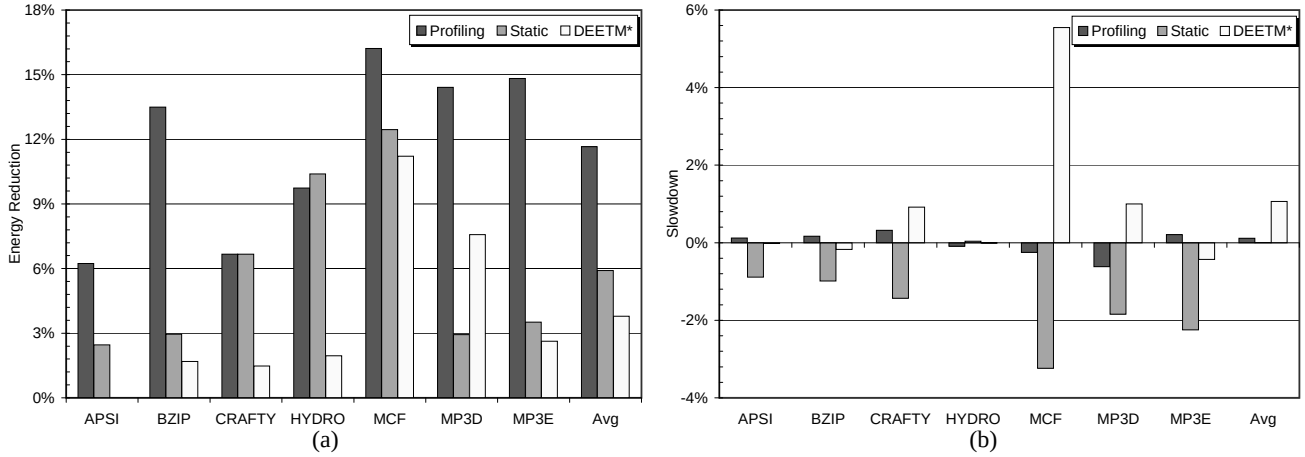

Figure 6: Comparing profiling based approach with static optimization and DEETM*.

at all for CRAFTY. The DEETM framework requires a positive slack to save energy. In this simulation, the slack used is a small 0.5%.

In Figure 6, the leftmost bar of each group is the same as in Figure 5. The bar in the middle of the group shows the *static* approach, and finally, the rightmost bar gives the result of *DEETM**.

A closer inspection of Figure 5 and 6 shows that the applications fall into three groups:

- Little phase-change within application:
  Naturally, if different parts of the application reacts similarly to low-power techniques, or those parts that react very differently do not have enough weight of execution time, then we can not exploit intra-application phase change. Therefore, doing profiling at module level will not be much different from doing profiling at the granularity of the whole application (the static approach).

  CRAFTY and HYDRO belong to this group though each of them favors a different technique. For these two applications, the static approach gives results very similar to that of the profiling based approach. Recall that when comparing the energy reduction in part (a) of the figures, we also have to take into account the slowdown shown in part (b), since it is usually possible to slow down even more for more energy savings(Figure 4).

- Moderate phase-change within application:
  Some applications have more noticeable behavior changes, thus the profiling based approach works moderately better, saving more energy after factoring in the slowdown, than the possible static combination. APSI and MCF fall into this category.

- Significant phase-change within application:
  This final category is the best for the module-level pro-

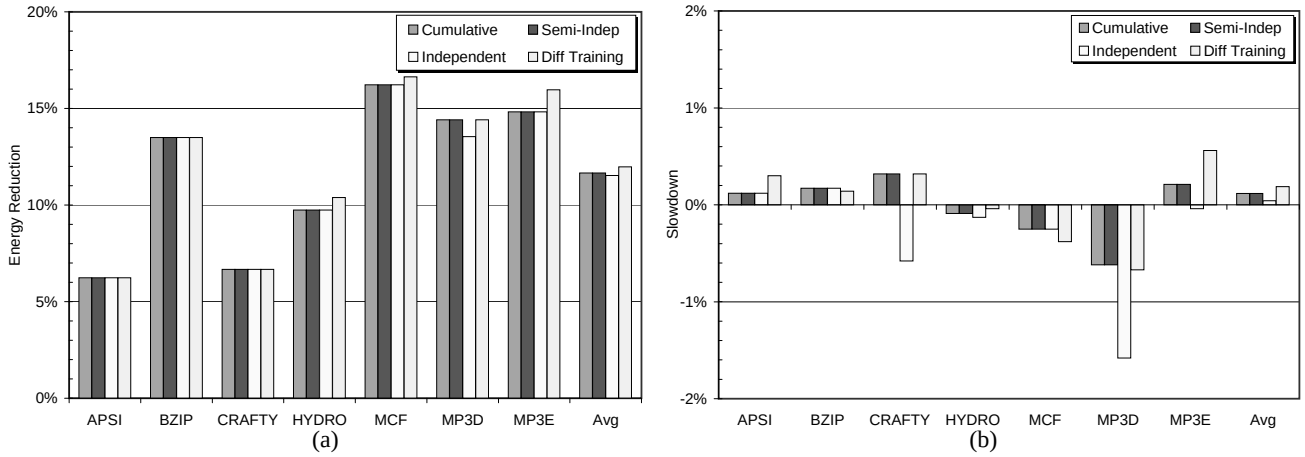

Figure 7: Parameters for profiling.

filing. These applications consist of sections of code that react very differently to techniques, so it pays off to profile them and apply each technique only at the suitable modules. BZIP and the two media benchmarks MP3D and MP3E belong to this category, and indeed the profiling based approach significantly outperforms any individual or static combination of techniques.

Finally, the improved DEETM framework nearly always produces a slower and less energy efficient solution than the profile-based approach. The reason is three-fold: first of all, instead of triggered when a different module is executed, the hardware re-tests the system at regular time intervals. This leads to random testing and is thus difficult to capture phase-changes well. Secondly, assuming a fixed order of techniques throughout the application leads to less efficient trade-off in some sections of the code. Finally, when testing slowdown of one technique, the hardware tests IPC at two time points before and after applying the technique, without knowing which part of the code is being executed at these points, this leads to misprediction of the slowdown which may lead to over-reacting (slowing down too much, using relatively inefficient techniques) or under-reacting (not slowing down enough, wasting the chance to save energy) in others.

To be fair, we need to note that DEETM is not designed with near-zero slack in mind. In a system with large slack, these drawbacks are less significant.

5.1.2 Profiling parameters

Section 5.1.1 analyzes *Cumulative* profiling. In Figure 7, we also show the result of *Independent* and *Semi-Indep* profiling. In our system, both stack filtering cache and phased cache try to reduce the energy consumed in L1 data cache. To quantify the overlap, *Semi-Indep* profiling performs an additional profile run with these two techniques activated simultaneously.

As we can see, by testing all possible combinations, *Cumulative* knows the interaction of techniques, and this generally leads to less misprediction then estimating based on statistics from each individual techniques, but this approach is really not necessary since it produces results that are very close to the other two styles which require much less profiling runs.

We can also see that in our case, just by performing one additional profile run, *Semi-Indep*'s results are nearly identical with those of *Cumulative* and slightly better than those of *Independent*. In the case of MP3D, *Semi-Indep*'s energy savings are noticeably better than those of *Independent*. Therefore, we recommend *Semi-Indep* type of profiling.

Another factor that is unrealistic in Section 5.1.1 is that we are using the same input file for the profiling runs. Now, we relax this constraint by running the profiling runs with a different training input file. In Figure 7, we label this experiment as *Diff Training*. In this experiment we only use *Semi-Indep* profiling. The figures show that the difference in the input files affects the results very little. This may seem strange, but in fact, it is quite logical because we use *efficiency score* to rank module/technique pair, though the absolute number of energy and performance might change significantly with different input set, this score is more stable, because it represents a characteristic of impact the technique has on the module. Also, we select a set of module/technique pairs to activate, the change in *efficiency score* may change some relative orders but as long as it doesn't change the content of the set much, the result will not change much either.

Finally, until now, the sampling interval selected is at function level without instrumenting medium sized func-

tions or breaking large functions into smaller modules. In Figure 8, we compare the effect of different sampling intervals in a limited fashion.

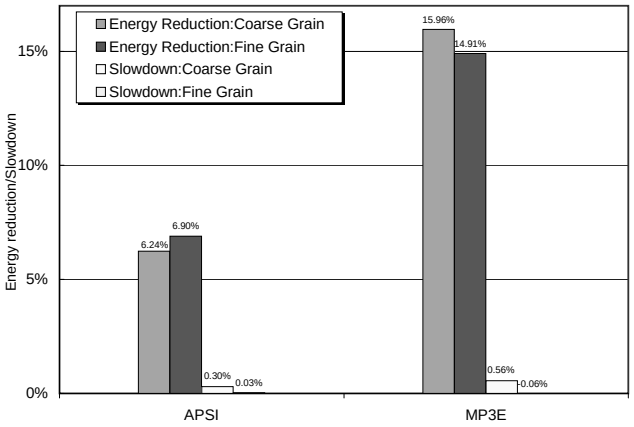

Figure 8: Effect of different grain size for profiling.

We carry out two experiments using two different applications. In the first experiment, we reduce the sampling granularity by instrumenting more functions in APSI. These are many medium-sized functions. Before instrumenting them, their behavior is blended with the caller's. By being more fine-grain, we are able to capture more behavior changes and save more energy. Specifically, we improve energy savings slightly. However, the extra instrumentation introduce much more overhead. In particular, the average switching frequency increases from once every 22,234 cycles to once every 606 cycles. There is, therefore, a trade-off of the benefit and the price for being more fine-grain.

In a second experiment, we reach finer granularity by breaking some functions into smaller modules in application MP3E. As shown in Figure 8, taking into account the larger slowdown for the coarse-grain scheme, the two produce very similar results.

These two experiments suggest that in many cases, function is a pretty natural entity that shows uniform reaction to low-power techniques. Obviously, programming style for a particular application also determines whether this is true or not. Considering the effort in determining where to instrument the code, and runtime overhead for this instrumentation, we believe, focusing on functions is more cost-effective.

5.2 Performance-only profiling

If a chip is not equipped with energy statistics registers, we are forced to use only performance statistics as feedback. As described in Section 3.2 we use only slowdown to calculate efficiency score for ranking the techniques. The results are shown in Figure 9.

As discussed in Section 3.2, by ignoring energy, it is possible to activate a technique with a small slowdown that has little or even negative energy savings resulting in non-optimal energy reduction per unit delay. In reality, our data shows that this does happen, but very infrequently, and the quantitative difference of the two approaches is not significant.

In general, the information about energy consumption helps to better shield the noise of using different input files and normal fluctuations. With the energy statistics, we know the behavior better, and can produce better scheduling, but the results based on performance-only profiling are sufficiently satisfying for practical purposes.

6 Related work

The main related work are the dynamically adaptable systems. Among the many works, Albonesi introduced an interesting concept of CAP [1], Complexity Adaptive Processors. A CAP is a processor that adapts the resources to the application.

While [1] is only performance oriented, the concept can be naturally extended to incorporate energy issues. Other CAP works [2, 4] propose different heuristics to decide when to activate or deactivate a single technique. These heuristics are closely related to the specific technique discussed, and therefore can not be applied in a general manner in a system with multiple unrelated techniques. An interesting difference between our findings is that in [4], after comparing sampling at subroutine level and at fixed periodic intervals, the latter is selected because it is simpler and better. In contrast, we sample at subroutine level for two reasons. First, we want to capture behavior change. This is an inherent property of the code thus the system should consider the code. Second, using subroutines, a global view of the program is obtained. This facilitates us to slow down one part of the code more than another and obtain more energy reduction than if we choose to slow down the application uniformly.

In [21], applications are tested with different configurations to determine the best one for each basic block and this selection is encoded in an extended instruction set to adapt the system at runtime. The extension is required to avoid the otherwise significant overhead of such fine-grain adaptation. Our work shows that by adapting at subroutine level, a much coarser grain size, we eliminate the need for an instruction set redesign. Also different is that we maximize energy savings per unit slowdown, a metric that has to be optimized globally, while [21] optimizes locally. This is because energy is used as the metric to optimize. Being an additive metric, it can be optimized locally for each basic block.

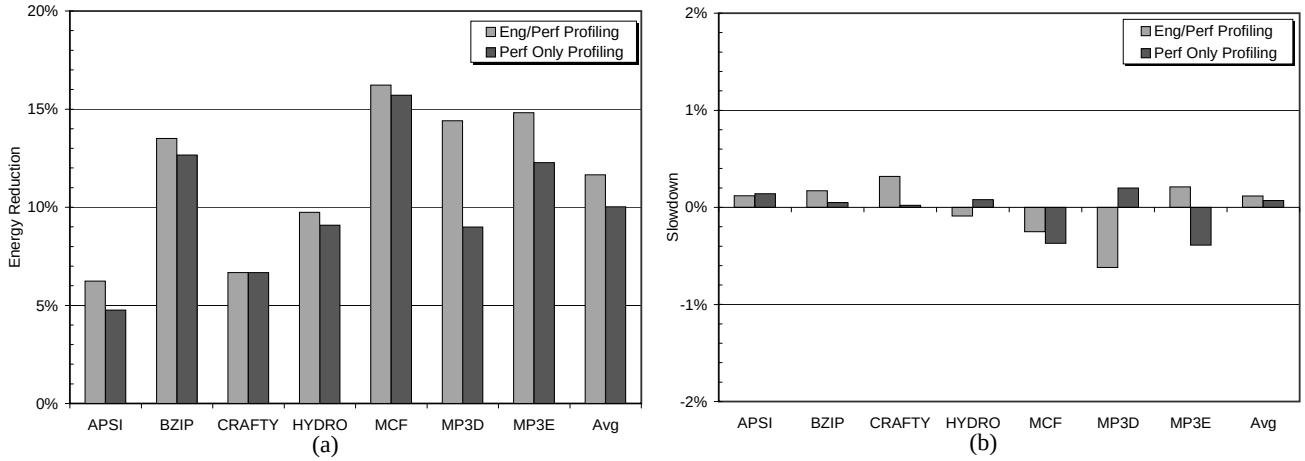

Figure 9: Energy/Performance profiling vs. Performance only profiling.

Many schemes have been proposed to trade off, sometimes extra, performance for energy/power reduction for curbing temperature surge, extending battery life, and so on [5, 9, 20, 23, 24]. To this end, the following two works are the most related.

In [13], a framework is designed to adapt system configuration and frequency to lower the energy consumption while still meeting the deadline of multimedia applications. That work focuses on multimedia applications and adapts at a coarser granularity. Our system targets broader range of applications, and does not rely on any knowledge of the application's functionality. Our approach uses finer granularity to exploits the non-uniformity. Additionally, [13] tries to eliminate performance slack, while we target iso-performance.

The DEETM [11] framework also tries to maximize energy savings for a given tolerable slowdown. This hardware based approach does not depend on profile, thus without knowing the characteristics about applications, and the changes inside them, DEETM optimizes for the common case and produces less optimal energy savings for very small slowdown. By using profile, we adapt the system better and thus energy savings become more significant in our system.

7 Conclusions

Knowing that applications go through different phases, in this paper we show that this phase-change also manifests itself in terms of relative impact of system configuration change. In particular, different sections of the code exhibit very different reaction, in terms of relative slowdown and energy reduction, after applying one of the common micro-architectural low-power techniques. We show that this non-uniformity of reaction can be easily exploited using profiling to improve the efficiency of trading off performance for energy reduction.

Our study shows that function level sampling, with some simple transformations to further reduce the activation overhead, is a very cost-effective approach to exploit phase-changes. We find that with low profiling cost, semi-independent profiling, produces results almost identical to those of cumulative profiling, a more costly approach in a system that employs multiple low-power techniques. Also, as we focus only in the relative change of performance and energy after applying a certain technique, the influence of using different input file is very small. This renders profiling very practical. Finally, even without energy counters, we can still exploit the non-uniformity effectively.

Using profiling, enabled by a set of four different low-power techniques, we manage to reduce on average 12% of the energy consumption for a set of seven diverse applications with negligible slowdown. The result is significantly better than an improved DEETM system that also tries to maximize energy reduction for a given slowdown or a static solution that employs the most efficient set of techniques per application.

REFERENCES

[1] D. Albonesi. Dynamic IPC/Clock Rate Optimization. In *International Symposium on Computer Architecture*, pages 282–292, July 1998.

[2] D. Albonesi. Selective Cache Ways: On-Demand Cache Resource Allocation. In *IEEE Journal on Instruction Level Parallelism*, volume 2, 2000.

[3] R. Bahar and S. Manne. Power and Energy Reduction Via Pipeline Balancing. In *International Symposium on Computer Architecture*, pages 218–229, 2001.

[4] R. Balasubramonian and D. Albonesi. Memory Hierarchy Reconfiguration for Energy and Performance in General-

Purpose Processor Architectures. In *International Symposium on Microarchitecture*, pages 245–257, December 1999.

[5] D. Brooks and M. Martonosi. Dynamic Thermal Management for High-Performance Microprocessors. In *International Symposium on High-Performance Computer Architecture*, 2001.

[6] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *International Symposium on Computer Architecture*, pages 83–94, June 2000.

[7] D. Folegnani and A. Gonzales. Energy-Effective Issue Logic. In *International Symposium on Computer Architecture*, pages 230–239, 2001.

[8] K. Ghose and M. Kamble. Reducing Power in Superscalar Processor Caches Using Subbanking, Multiple Line Buffers and Bit-Line Segmentation. In *International Symposium on Low Power Electronics and Design*, pages 70–75, August 1999.

[9] T. Halfhill. Transmeta Breaks x86 Low-Power Barrier. *Microprocessor Report*, 14(2):1,9–18, February 2000.

[10] A. Hasegawa et al. SH3: High Code Density, Low Power. *IEEE Micro*, pages 11–19, Dec 1995.

[11] M. Huang, J. Renau, S.-M. Yoo, and J. Torrellas. A Framework for Dynamic Energy Efficiency and Temperature Management. In *International Symposium on Microarchitecture*, pages 202–213, December 2000.

[12] M. Huang, J. Renau, S.-M. Yoo, and J. Torrellas. L1 Data Cache Decomposition for Energy Efficiency. In *International Symposium on Low Power Electronics and Design*, pages 10–15, August 2001.

[13] C. Hughes, J. Srinivasan, and S. V.Adve. Saving Energy with Architectural and Frequency Adaptations for Multimedia Applications. In *International Symposium on Microarchitecture*, 2001.

[14] Intel. *Pentium III Processor Mobile Module: Mobile Module Connector 2 (MMC-2) Featuring Intel SpeedStep Technology*, 2000.

[15] Intel Corporation. *Mobile Power Guidelines 2000, Rev 1.0*, 1998.

[16] K. Itoh. Low Power Memory Design. In *Low Power Design Methodologies*, pages 201–251. Kluwer Academic Publisher, 1996.

[17] R. Joseph and M. Martonosi. Run-Time Power Estimation in High Performance Microprocessors. In *International Symposium on Low Power Electronics and Design*, pages 135–140, 2001.

[18] J. Kin, M. Gupta, and W. Mangione-Smith. The Filter Cache: An Energy Efficient Memory Structure. *International Symposium on Microarchitecture*, pages 184–193, December 1997.

[19] V. Krishnan and J. Torrellas. An Execution-Driven Framework for Fast and Accurate Simulation of Superscalar Processors. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 286–293, October 1998.

[20] S. Manne, A. Klauser, and D. Grunwald. Pipeline Gating: Speculation Control for Energy Reduction. In *International Symposium on Computer Architecture*, pages 132–141, July 1998.

[21] D. Marculescu. Power-Driven Code Execution for Low Power Dissipation. In *International Symposium on Low Power Electronics and Design*, August 2000.

[22] A. Nannarelli. *Low Power Division and Square Root*. PhD thesis, University of California, Irvine, Department of Electrical and Computer Engineering, June 1999.

[23] E. Rohou and M. Smith. Dynamically Managing Processor Temperature and Power. In *2nd Workshop on Feedback-Directed Optimization*, November 1999.

[24] H. Sanchez et al. Thermal Management System for High Performance PowerPC Microprocessor. In *IEEE Computer Society International Conference*, pages 325–330, February 1997.

[25] T. Sherwood, E. Perelman, and B. Calder. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. In *International Conference on Parallel Architecture and Compilation Techniques*, September 2001.

[26] C.-L. Su and A. Despain. Cache Design Trade-offs for Power and Performance Optimization: A Case Study. In *International Symposium on Low Power Electronics and Design*, pages 63–68, April 1995.

[27] S. Wilton and N. Jouppi. CACTI: An Enhanced Cache Access and Cycle Time Model. *IEEE Journal on Solid-State Circuits*, 31(5):677–688, May 1996.

[28] B. Xu and D. Albonesi. Runtime Reconfiguration Techniques for Efficient General-Purpose Computation. In *IEEE Design & Test of Computers*, pages 42–52, January 2000.