In Proceedings of the 31st Annual International Symposium on Computer Architecture, June 2004

A Formal Approach to Frequent Energy Adaptations for Multimedia Applications*

Christopher J. Hughes
Intel Corporation
Architecture Research Lab
christopher.j.hughes@intel.com

Sarita V. Adve
University of Illinois at Urbana-Champaign
Department of Computer Science
sadve@cs.uiuc.edu

Abstract

Much research has recently been done on adapting architectural resources of general-purpose processors to save energy at the cost of increased execution time. This work examines adaptation control algorithms for such processors running real-time multimedia applications. The best previous algorithms are mostly heuristics-based and ad hoc, requiring an impractically large amount of application- and resource-specific tuning.

We take a more formal approach that does not require the large tuning effort of previous approaches, and yet obtains average energy savings comparable to the best previous approach. We pose control algorithm design as a constrained optimization problem: what configuration should be used at each point in the program to minimize energy for a targeted performance given that each configuration has a different energy-performance tradeoff at each point? We solve this with the method of Lagrange multipliers, which assumes knowledge of the energy-performance tradeoffs. We develop a technique to estimate these tradeoffs using properties of multimedia applications. Our technique is likely extendible to other application domains.

We compare our algorithm to the best previous algorithm for real-time multimedia applications, which is heuristics-based. We demonstrate the practical difficulty of the tuning process for the previous algorithm. Compared to a painstakingly hand-tuned version of that algorithm, our new algorithm provides similar energy savings through a more formal approach that does not need such heroic tuning, making it practical to implement.

*This work is supported in part by a gift from Intel Corp., and the National Science Foundation under Grant No. CCR-0096126, EIA-0103645, CCR-0209198, CCR-0205638, EIA-0224453, and CCR-0313286. Sarita V. Adve was also supported by an Alfred P. Sloan Research Fellowship and Christopher J. Hughes was supported by an Intel Ph.D. Fellowship.

1 Introduction

Energy consumption is a key design consideration for general-purpose processors, especially for mobile systems. This paper focuses on reducing energy consumption of general-purpose processors when running multimedia applications for mobile systems.

One way to reduce energy consumption is to leverage variability in some aspect of the execution (e.g., resource usage) and adapt. Many such techniques have been proposed (e.g., [6, 9, 26]), and some adaptation techniques have been implemented in real systems (e.g., Pentium M).

Ideally, we would like to adapt frequently to fully exploit any variability. However, existing techniques for adapting frequently are ad hoc and require an impractically large amount of tuning effort. We propose a new, more formal method for controlling frequent adaptations. Relative to previous techniques, the new technique shows comparable energy savings, but requires very little tuning and is practical to implement.

This work is part of the broader Illinois GRACE project exploring adaptations at multiple granularities and across different system layers [29, 33]. Although here we focus on multimedia applications, we believe our techniques can be extended to other domains as well.

1.1 Previous Adaptation Algorithms

Current processor adaptation techniques target two types of variability. First, the processor sometimes runs faster than needed, incurring higher power dissipation than necessary; thus, we can slow it down to save energy. We refer to this as exploiting *temporal slack*. For real-time multimedia applications, temporal slack is inherent. These applications need to process discrete units of data, generally called frames, within a deadline; they need only meet the deadline, not beat it. The difference between the required and the actual execution time is temporal slack, which varies from frame to frame. For other applications, the allowed slowdown could be a user-specified target.

The second type of variability exploited through adaptation is in the usage of microarchitectural resources. Sometimes reducing the amount of a resource for a while does not affect performance. We refer to this as *resource slack*. Resource slack varies with the type of computation being performed, and typically changes rapidly.

A key to effective adaptation is the control algorithm, which must determine when to adapt (adaptation time scale) and what to adapt (adaptation scope). Our previous work has referred to the former as the temporal granularity and the latter as the spatial granularity of adaptation [30].

There is usually a conflict between the temporal and spatial granularity of adaptation [29]. Ideally, we would like to adapt frequently (i.e., temporally local granularity) and we would like to consider all of the resources together when adapting (i.e., spatially global granularity). Previous algorithms typically do one or the other.

Many previously proposed algorithms exploit temporally local variability (e.g., [1, 2, 3, 5, 6, 9, 25, 26, 28]). However, they typically have the following four limitations. 1) It can be hard to predict the impact of adaptations at this temporal granularity; thus, to avoid accidental slowdowns, they do not exploit temporal slack. 2) To simplify predicting adaptation behavior, they operate at a spatially local granularity. 3) Due to their unpredictability, they require a large tuning effort to achieve best energy savings with a "small" slowdown. This effort grows exponentially with the number of adaptations. 4) Due to their frequent adaptation, they cannot control high overhead adaptations. Since these algorithms adapt in a fine-grained manner, or locally, in both a temporal and spatial sense, we refer to them as *LL* (for local-local).

We believe that the tuning effort required makes LL algorithms impractical for real systems. For example, in the systems we evaluate here, which adapt only three microarchitectural resources, there are 2^{57} possible design points in the tuning process, and simple search strategies are unlikely to find a reasonable one.

Some previous work on multimedia applications has looked at more coarse-grained temporal variability. For multimedia applications, we can predict the performance and energy of all architecture (resource) configurations at the frame granularity [18, 20]. Therefore, frame-level, or temporally global, algorithms can exploit temporal slack, can adapt in a spatially global manner, require little tuning, and can control high overhead adaptations; thus, they do not suffer from any of the above limitations of LL algorithms. However, these algorithms must use a single configuration for an entire frame. Since these algorithms adapt globally in both a temporal and spatial sense, we refer to them as *GG* (for global-global).

Previously, we combined the above two classes of algorithms to get the benefits of both [30]. Our Global+Local algorithm, which we refer to here as *GG+LL*, uses performance and energy prediction to pick a spatially global configuration for each frame to exploit temporal slack, and lets LL algorithms react during a frame to exploit remaining resource slack. However, GG+LL still requires an extensive tuning effort for the LL part.

1.2 Our Contribution

The limitations of the GG+LL algorithm motivate asking: 1) Can we get the benefits of GG+LL without so much tuning? 2) Is there a way to exploit both temporal and resource slack in an integrated manner? If so, how much benefit does this give?

We propose a new algorithm that addresses both of the above questions. Our algorithm decouples the temporal granularity of adaptation from the type of slack that is exploited. The temporal granularity for adapting a resource is now determined only by the adaptation overhead, the ability to predict performance and energy impact of adaptation, and the variability exhibited. Furthermore, if there are multiple adaptations with largely different overheads, our algorithm allows operating at multiple time scales, but exploits temporal slack at all time scales. For example, we resize architectural resources every 1024 instructions and the voltage and frequency at the granularity of a frame, and both adaptations consider temporal slack.

Our approach for controlling small overhead adaptations at a specific time scale (i.e., a certain interval size) is as follows. For a given interval, each hardware configuration uses up a certain amount of temporal slack and consumes a certain amount of energy. We want to pick a single configuration for each interval such that the total temporal slack used is no more than that available for the frame, and the total frame energy is minimized. This approach allows for temporal slack to be traded off, or "spread," between intervals in an optimal manner. In contrast, for GG+LL, the GG part chooses a single configuration for a whole frame (so it cannot spread temporal slack optimally), and the LL part of GG+LL is not intended to exploit temporal slack. Choosing the best set of configurations can be viewed as solving a constrained optimization problem. We apply a previously known optimization method to choose the best mapping of interval to configuration.

However, solving this problem requires perfect information about the temporal slack used and the energy consumption for each configuration, for each interval, for each frame. We show how to predict this information, independent of input, from profiled information for a single frame.

To incorporate large overhead adaptations (at the frame granularity), we solve the constrained optimization problem for different temporal slacks. We then split the temporal slack between temporally local and global adaptation, choosing the split that provides the most energy savings.

Since our algorithm can operate at temporally local and spatially global granularities, we refer to it as *LG*.

Compared to GG+LL, our formal approach results in only four independent parameters which took little effort to tune, one of which is common to GG+LL. GG+LL needs six more parameters for the system evaluated here (with three adaptive microarchitectural resources), and this number will grow as more resources are made adaptive. More importantly, the resource-specific parameters are difficult to tune even individually, and are not independent, necessitating a joint tuning process for best results. To our knowledge, nobody has proposed a technique to automatically search the GG+LL parameter space; therefore, we develop our own. These systematic methods of tuning fail to find acceptable design points (far too many deadlines are missed). Therefore, we rely on hand-tuning, which is extremely time consuming. Furthermore, GG+LL needs to be re-tuned if new resources are made adaptive, or if new applications are added to the test set, unlike for our LG algorithm.

The other possible advantage of LG over GG+LL is its ability to spread temporal slack more intelligently throughout each frame. However, we find the energy benefits of our new algorithm are marginal relative to GG+LL. Exploring this further, we find that the LL part of GG+LL unintentionally exploits temporal slack in addition to the intentional exploitation of resource slack. Further, it exploits temporal slack in different ways across different intervals, and the large tuning effort results in a serendipitous close-to-optimal spreading of temporal slack across the frame. Thus, GG+LL already sees the majority of the benefits from this technique, albeit it extracts these benefits in a very ad hoc manner (reflected in the great effort to tune GG+LL). Our LG algorithm spreads temporal slack in a more formal manner, and so is much simpler to implement in practice, while providing as much energy savings as GG+LL.

2 Design Space and Previous Algorithms

This section describes previously proposed GG, LL, and GG+LL adaptation control algorithms. We consider two classes of adaptations. The first, dynamic voltage and frequency scaling (DVS), is a high overhead adaptation. With DVS, the voltage of the processor is lowered to save energy, necessitating a frequency drop and increasing execution time. The second class of adaptations, architecture adaptation, is usually low overhead. Typically, part of the architectural resource being adapted is deactivated or reactivated. These adaptations may or may not impact execution time, depending on the amount of resource slack. Below, we use the term *architecture configuration* to refer to the configuration of all architecture resources and we use *hardware configuration* to refer to the combination of the architecture configuration and the voltage and frequency.

2.1 Previous GG Algorithm

Previously, we proposed (to our knowledge) the only GG algorithm for multimedia applications that integrates DVS and architecture adaptation [20]. At the beginning of a frame, it predicts the hardware configuration that will minimize energy for that frame without missing the deadline. The frame is run with this configuration.

The algorithm consists of two phases: a profiling phase at the start of the application and an adaptation phase. The profiling phase profiles one frame of each type[1] for each architecture configuration A, at some base voltage and frequency. For each A, the algorithm collects the instructions per cycle (IPC_A) and average power (P_A) for each frame.

We previously showed that for several multimedia applications and systems, for a given frame type, average IPC and power for a configuration are roughly constant for all frames [18, 20]. Thus, IPC_A and P_A values from the profiling phase can be used to predict IPC_A and P_A of all other frames of that type. Previous work also showed that IPC is almost independent of frequency for these applications [18], so IPC_A can be used to predict the IPC for architecture A at all frequencies. Similarly, P_A and the voltage for each frequency can be used to predict the power for architecture A at all frequencies ($Power \propto V^2 f$).

For each hardware configuration, H, with architecture A_H, frequency f_H, and voltage v_H, the algorithm computes the most instructions H can execute within the deadline as $I_{max_H} = deadline \times IPC_{A_H} \times f_H$. It also computes the energy consumed per instruction (EPI) for each H as $EPI_H \propto \frac{P_{A_H} \times {V_H}^2}{IPC_{A_H}}$. It then constructs a table with an entry for each H containing its I_{max} and EPI, sorted in order of increasing EPI.

After profiling is complete, the algorithm enters the adaptation phase. Before executing a frame, it predicts the number of instructions the frame will execute, using a simple history-based predictor (takes the maximum of the last five frames of the same type and adds some leeway). It then searches the table (starting at lowest EPI) for the first entry with $I_{max} \geq predicted\, instructions$. It predicts this to be the lowest energy configuration that will meet the deadline, and so chooses it.

2.2 Previous LL Algorithms

A number of researchers have studied algorithms to control individual adaptive architecture resources, although

[1]Two applications in our suite have multiple frame types (i.e., I, P, and B frames for MPEG-2 encoder and decoder). For these, the algorithm profi les and adapts for each frame type separately.

most of this work has been done for general applications (e.g., SPEC). Adaptations considered include changing the instruction window size (or issue queue and reorder buffer sizes) [5, 6, 9, 28, 30], changing the number of functional units and/or issue width [2, 26, 30], and others [1, 3, 10, 15, 25]. In this study, we focus on changing the instruction window size and the number of active functional units (and issue width). We do not consider controlling DVS with an LL algorithm because DVS is a high overhead adaptation – it takes a relatively long time to change the frequency (about 10μs [13]).

Previously, we proposed the best current LL algorithms for adapting the instruction window size and the number of active functional units evaluated for multimedia applications [30]. These algorithms operate at fixed time intervals (every 256 cycles in [30]). During an interval, each algorithm independently collects and computes statistics (e.g., the number of issue hazards) which serve as proxies for performance loss from adaptation. At the end of each interval it compares the statistics to a set of *thresholds*. The outcome of these comparisons determines the configuration for the next interval – resources are reactivated if the estimated performance loss is high, or deactivated if low.

2.2.1 Tuning

Although the LL algorithms ideally do not use temporal slack, in practice they are permitted to reduce performance by a "small" amount, since most of the time this leads to much larger energy savings. The thresholds of the LL algorithms (each algorithm we consider has two thresholds) must be carefully tuned to achieve the right tradeoff between energy and performance. It is difficult to tune the thresholds for even one resource since they are based on heuristics and the design space is large – the algorithms for each of the three resources we consider have a design space of 2^{19} points each.

Each set of thresholds gives a different tradeoff between performance and energy (which may vary across applications). Our work considers a fixed performance target in terms of a maximum fraction of missed deadlines (5% in our experiments). Therefore, the process of tuning the thresholds involves running experiments with various combinations of thresholds and picking one which meets the performance target with lowest energy. Given the large design space, it is infeasible to test all points; therefore, we need to carefully search the space. Such a search is complicated by a number of factors. First, the relationship between energy and any one threshold is hard to predict because energy depends both on power and execution time. Sometimes, deactivating part of a resource increases energy consumption. Second, the experiments in the search will be on only a small set of applications. For a given set of thresholds, the heuristics may behave differently for other applications than for the test set. Third, the instruction window and functional unit algorithms each have two thresholds that are dependent on one another; thus, finding the optimal design point likely requires tuning two thresholds simultaneously. Similarly, for a system with multiple adaptive resources, the adaptive resources may interact. For example, if the instruction window algorithm reduces the size of the window, the ALU utilization may drop. Therefore, the thresholds for all such resources may need to be tuned simultaneously. For the system modeled here, there are 2^{57} total design points. In general, if all thresholds are tuned together, the tuning effort grows exponentially with the number of resources.

Another disadvantage of this type of LL algorithm is the effort required to make a new resource adaptive. In addition to having to develop new heuristics for it, much of the previous tuning work may be useless, making it time consuming to extend an existing design.

One possible way to reduce the amount of tuning necessary for LL algorithms is to design more formal ones based on control theory. To our knowledge, no such algorithm has been proposed for the resources considered here. Such an algorithm would need to handle the complicated response of energy savings to adaptation while maintaining performance. Further, the task of obtaining a feedback signal is difficult since energy savings and slowdown (relative to the base system) are not known during execution. Nevertheless, this is a promising direction and we leave its exploration to future work.

2.3 Previous GG+LL Algorithm

Sasanka et al. also propose a combined GG+LL algorithm [30]. The LL algorithms always run while the GG algorithm runs, including during both the profiling and adaptation phases. GG sets the maximum amount of each resource; LL cannot increase beyond these limits. Running LL during the profiling phase allows GG to account for the energy and performance impact of the local adaptations. This combined approach exploits both temporal and resource slack, but separates them into different temporal granularities. It also requires tuning due to the use of LL algorithms. In addition, the LL algorithms may need to be re-tuned once integrated with GG.

3 LG Algorithm

We now present our new LG algorithm for exploiting both temporal and resource slack in an integrated manner. Our LG algorithm also relies on more formal methods, resulting in a drastically reduced tuning effort compared to GG+LL, and making LG much more practical.

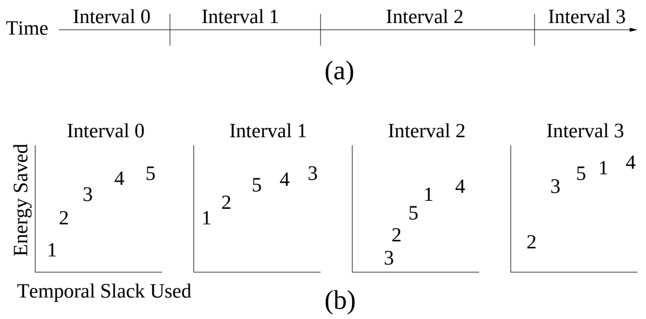


Figure 1. (a) Example frame composed of four intervals. (b) Energy-performance tradeoffs for the intervals. The numbers correspond to different architecture configurations.

We decouple the use of low and high overhead adaptations, (close-to-optimally) splitting the available temporal slack between them. Section 3.1 first presents the method for controlling low overhead adaptations (which are invoked frequently within a frame), focusing on architecture adaptation. Section 3.2 explains how we control high overhead adaptations (which are invoked once per frame), focusing specifically on DVS. It also describes how we split the temporal slack between low and high overhead adaptations. Section 3.3 gives the overheads of the LG algorithm, and Section 3.4 describes the design parameters for it. Finally, Section 3.5 discusses how our LG algorithm might apply to other application domains.

3.1 Controlling Low Overhead Architecture Adaptation

The Optimization Problem
We use the term *interval* to refer to the granularity at which low overhead adaptations are invoked. In this work, we divide each frame into multiple intervals, each with the same number of instructions (1024 in our system). Each interval can be run with one of several architecture configurations, each with a different energy/performance tradeoff. Figure 1 shows an example set of intervals, with the *energy saved* vs. *temporal slack used* data for each architecture configuration (with respect to the base configuration). Our goal is to determine the best configuration (C_i) for each interval i such that together these configurations result in the most energy saved while using no more than the target slack for architecture adaptation for the frame. In other words, we want to find the best way to "spread" the temporal slack across the frame.

Our problem can be stated as a constrained optimization problem:

$$maximize\ \frac{1}{N}\sum_{i=1}^{N} E_i(C_i),\ subject\ to\ \frac{1}{N}\sum_{i=1}^{N} S_i(C_i) \leq S_{target}$$

where N is the number of intervals in the frame, $E_i(C_i)$ and $S_i(C_i)$ are respectively the average energy saved per instruction (EPI-saved) and the average temporal slack used per instruction (SPI-used), by using C_i vs. the base configuration for interval i. S_{target} is the available target slack per instruction for the frame (Section 3.2 describes how this is determined).

The Solution
There are many methods for solving constrained optimization problems. Our approach is inspired by work on an equivalent problem in the wireless communications domain, where the method of Lagrange multipliers is applied [22]. Details on our solution are in [17]. The key idea is that the best configurations for the different intervals must have the same tradeoff between (ratio of) EPI-saved to SPI-used. This assumes that the configurations, if plotted to create an EPI-saved vs. SPI-used curve (as shown in Figure 1(b)), must form a convex curve. If this is not the case for an interval, we disregard some configurations to force this condition, possibly leading to a suboptimal solution.

Obtaining Energy-Slack Tradeoffs
Our solution to the constrained optimization problem assumes that we have perfect EPI-saved and SPI-used information for all configurations, for all intervals, for all frames. This is infeasible since it requires profiling all frames once per configuration before adapting. Instead, we profile to collect this perfect information for a single frame of a single input at application installation time, and find the optimal configurations for that frame. At run-time, no further profiling is done. Instead, we map the solution found at installation time to all other frames using some insights about real-time multimedia applications. Our technique is likely extendible to other application domains.

We define some property of each interval as its mapping key – for an interval with key value k, we use the same configuration as an interval in the profiled frame with key value k. We chose the program counter (PC) value of the first instruction in an interval as the key. While this works reasonably well for us, for reasons given below, one avenue of future work is to leverage recent research on program phase detection and prediction to develop a more sophisticated key [7, 31].

Using PC as the key means that for a given PC, the algorithm will choose the same configuration for all intervals that start with that PC. For this configuration to be close to the best for all of those intervals, two properties should hold: 1) intervals with the same starting PC value should have similar tradeoffs between slack and energy for each configuration, and 2) the fraction of intervals with each starting PC value should be the same across all frames. Both properties are generally exhibited by multimedia applications. Their repetitive nature leads to the first property holding. The second property holds due to the following. Previously, it was reported that the per frame IPC (and EPI) for real-time multimedia applications is almost constant across different frames [18]. The intuition for this is that while different frames may do a different amount of work, the nature of the work is the same. This implies that the fraction of intervals with each starting PC value is almost the same across different frames.

Nevertheless, PC is not a perfect key, and sometimes multiple intervals from the profiled frame have the same PC, but different optimal configurations, necessitating a compromise between the configurations for those intervals (this would be true for other choices for a key as well). For each PC value, for each resource, the algorithm computes the mean of the optimal configuration (e.g., instruction window size) for the intervals of the profiled frame with that PC. It then chooses the supported configuration closest to that.

The algorithm builds a table that maps PC to a chosen configuration. While running the application, at each interval, the algorithm uses its starting PC value to select the configuration to use for it. To reduce aliasing effects in the finite table, rather than using one entry per instruction it uses one entry per block of consecutive instructions (256 in our experiments).

The above table needs to be built for each S_{target} as explained below.

3.2 Controlling High Overhead Adaptation and Integration with Low Overhead Adaptation

As mentioned earlier, LG can integrate the control of high overhead adaptations performed at the frame granularity with low overhead adaptations performed during a frame. We use DVS at the frame granularity as an example of a high overhead adaptation. To control DVS alone, or when controlling DVS in conjunction with architecture adaptation, LG proceeds much like GG. Before the start of each frame, LG examines all configurations to determine which will meet the deadline with minimum energy. However, instead of considering the low overhead adaptations individually, LG treats all of those adaptations together as a single "resource" whose "configurations" consist of different amounts of temporal slack (i.e., different values of S_{target}).

At application installation time, LG runs the (low overhead) architecture adaptation optimization algorithm for a set of S_{target} values (we choose multiples of 0.01 between 0 and 1 cycle per instruction). For each S_{target} value, it estimates the resulting per frame IPC and EPI and records this information to use at run-time. LG estimates EPI by determining which architecture configuration it would use for each interval of the profiled frame, and summing the corresponding EPI-saved across all intervals. IPC is estimated in an analogous way. For each combination of voltage/frequency and S_{target}, LG computes I_{max} and EPI us-

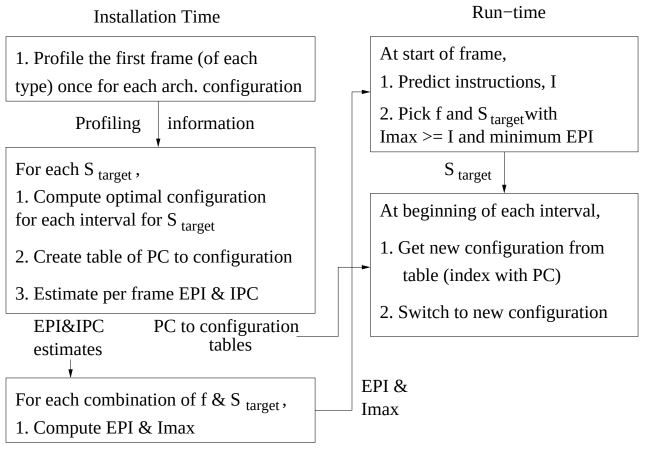

Installation Time
Run−time
1. Profile the first frame (of each type) once for each arch. configuration
Profiling information
For each S target ,
1. Compute optimal configuration for each interval for S target
2. Create table of PC to configuration
3. Estimate per frame EPI & IPC
EPI&IPC estimates
PC to configuration tables
For each combination of f & S target ,
1. Compute EPI & Imax
EPI & Imax
At start of frame,
1. Predict instructions, I
2. Pick f and S target with Imax >= I and minimum EPI
S target
At beginning of each interval,
1. Get new configuration from table (index with PC)
2. Switch to new configuration

Figure 2. The new LG algorithm.

ing the above estimates and taking into account the impact of DVS. At run-time, before each frame, LG uses the same instruction count predictor as GG (Section 2.1) to choose the combination of voltage/frequency and S_{target} that can make the deadline with least energy.

Figure 2 summarizes the complete algorithm.

A system with other high overhead adaptations may require a profiling phase at application installation to account for their interaction with low overhead adaptations.

For systems with many possible frequencies, the number of combinations of voltage/frequency and S_{target} may be too large to store and search through the I_{max} and EPI values. Previously, we developed a variation of GG for such systems [20]. It chooses a single architecture configuration for all frames by computing a frequency-independent measure of EPI for each. It is straightforward to extend LG to incorporate this idea. We omit a description here for lack of space.

3.3 Required Support and Overhead

Our LG algorithm requires some special support for both the installation time and run-time portions. At installation time, LG profiles one frame (the same one) per architecture configuration considered. Applications can be restarted for each architecture to avoid modifying them. This is done only once ever for each application and the number of frames profiled is likely to be small compared to the number of frames processed during a single run of the application. Nevertheless, we have considered ways to reduce the profiling effort. One simple way is to consider only a subset of the possible architecture configurations, at the cost of some energy savings. This can be done strategically to minimize the lost savings. We have developed a technique along these lines [17], but omit a description here for lack of space. We also run the optimization algorithm at installation time. Its complexity is relatively small, $O(N \log M)$, where N is the number of intervals in the profiled frame and M is the number of architecture configurations considered. The installation time portion of LG needs to be implemented in software. Besides the ability to adapt the hardware, this requires no special hardware support.

At run-time, once per frame, the LG algorithm predicts the instruction count and chooses the S_{target} and voltage/frequency, for which the required support is small. LG also chooses a configuration once per interval. It does this with a lookup into a PC-indexed table. We found that for these applications, a small table for each S_{target} is sufficient (128 entries). Thus, this lookup is very fast, and since it is performed only once per interval, we expect the energy overhead to be negligible (as long as the interval size is sufficiently large).

3.4 Design Parameters

As opposed to the LL algorithms, LG does not use heuristics-based thresholds to help control adaptation, and thus avoids the enormous tuning effort required for those algorithms. However, LG still has the following design parameters: the PC to configuration table size, the block size used when indexing the table, the interval length, and the per frame instruction count predictor. We use the same instruction count predictor as for GG. It performs well for all our applications, but could be made adaptive, if necessary (e.g., be more conservative if we detect too many missed deadlines). Finding good values for the other parameters is simple for the following reasons.

These parameters are all independent of one another and the table and block size are independent of the adaptive resources in the system. All are also relatively insensitive to the application (for those considered here). Finally, assuming we restrict their values to powers of two, there are few choices for each. The above factors combine to make the design space for the parameters very small, especially compared to that for the LL algorithms.

The table size (128 entries), block size (256), and interval length (1024) were chosen independently through evaluation of less than 10 design points using only a subset of our applications. The results are relatively insensitive to the table and block size. We choose to exploit as much variability as possible; thus, we choose as small an interval size as possible to maximize energy savings. However, we are constrained by the fact that if the interval size is too small then adjacent intervals interact and we miss too many deadlines.

3.5 Using LG for Other Application Domains

While LG targets multimedia applications, we believe it could be extended for use with other application domains as well. The key to LG working well is its ability to map the profiled frame to all other frames. While non-multimedia applications may not have frames, recent work has shown that many applications do exhibit phased and predictable behavior [7, 31]. It seems likely that a combination of a phase detection/prediction algorithm and LG would apply to these applications. The former would predict phase changes and the type of phase that was about to start and feed this information into LG, which would use knowledge about the phase type to spread the slack across the phase. We leave exploration of this idea to future work.

Base Processor Parameters	
Processor speed	1GHz
Fetch/retire rate	8 per cycle
Functional units	6 Int, 4 FP, 2 Add. gen.
Integer FU latencies	1/7/12 add/multiply/divide (pipelined)
FP FU latencies	4 default, 12 div. (all but div. pipelined)
Instruction window (reorder buffer) size	128 entries
Register file size	192 integer and 192 FP
Memory queue size	32 entries
Branch prediction	2KB bimodal agree, 32 entry RAS
Base Memory Hierarchy Parameters	
L1 (Data)	64KB, 2-way associative, 64B line, 2 ports, 12 MSHRs
L1 (Instr)	32KB, 2-way associative
L2 (Unified)	1MB, 4-way associative, 64B line, 1 port, 12 MSHRs
Main Memory	16B/cycle, 4-way interleaved
Base Contentionless Memory Latencies	
L1 (Data) hit time (on-chip)	2 cycles
L2 hit time (off-chip)	20 cycles
Main memory (off-chip)	102 cycles

Table 1. Base (default) system parameters.

App.	Frames	Base IPC	Tight Deadline	% Slack on Base	
				Tight	Loose
GSMdec	1000	3.7	15μs	9.8	54.9
GSMenc	1000	4.6	45μs	8.9	54.4
G728dec	250	2.3	51μs	10.2	55.1
G728enc	250	2.1	65μs	9.2	54.6
H263dec	150	3.4	450μs	15.6	57.8
H263enc	150	2.3	18.78ms	25.1	62.5
MPGdec	250	3.6	2.11ms	31.2	65.6
MPGenc	250	3.0	55.3ms	38.7	69.4
MP3dec	1000	3.0	495μs	22.3	61.1

Table 2. Workload, deadlines, and temporal slack (% of deadline) on the base processor. Base IPC is the mean per frame IPC on the base processor. The loose deadlines are twice the tight ones.

4 Experimental Methodology

4.1 Systems Modeled

We use the execution driven RSIM simulator [19] for performance evaluation and the Wattch tool [4] integrated with RSIM for energy measurement.

Table 1 summarizes the base, non-adaptive, processor studied. In this work, the architecture resources we adapt are the instruction window and the integer and floating point ALUs. We model a centralized instruction window with a unified reorder buffer and issue queue, composed of eight entry segments (the physical register file is a separate structure). For instruction window adaptation, at least two segments must always be active. For functional unit adaptation, we assume that the issue width is equal to the sum of all active functional units and hence changes with the number of active functional units. Consequently, when a functional unit is deactivated, the corresponding instruction selection logic is also deactivated. Similarly, the corresponding parts of the result bus, the wake-up ports of the instruction window, and ports of the register file (including decoding logic) are also deactivated. Experiments with DVS assume a frequency range from 100MHz to 1GHz with 1MHz steps and corresponding voltage levels derived from information available for Intel's XScale processor [21] as further discussed in [20].

We assume clock gating for all processor resources. If a resource is not accessed in a given cycle, Wattch charges 10% of its maximum power to approximate the energy consumed by logic within the resource that cannot be gated (or cannot always be gated when unused). To represent the state-of-the-art, we also gate the wake-up logic for empty and ready entries in the instruction window as proposed in [9]. We assume that resources that are deactivated by the adaptive algorithms do not consume any power. In our model, due to clock gating, deactivating an unused resource saves only 10% of the maximum power of the resource.

We evaluate all of the above adaptation control algorithms. Since GG+LL was previously found to be better than GG or LL alone [30], we evaluate GG and LL for reference only, using the algorithm in [20] for GG and "Local" in [30] for LL. For GG+LL, we use the "Global+Local" algorithm from [30]. Tuning of the LL algorithms for LL and GG+LL is discussed in Section 5.1. For LG, we use the algorithm described in Section 3.

For GG and GG+LL, we profile all possible combinations of the following architecture configurations (54 total): instruction window size $\in \{128,96,64,48,32,16\}$, number of ALUs $\in \{6,4,2\}$, and number of FPUs $\in \{4,2,1\}$. GG+LL, due to its incorporation of LL, can choose any supported architecture configuration during a frame (i.e., it is not restricted to the 54 profiled ones). As discussed in Section 3.3, for LG, we profile configurations according to the pruning algorithm given in [17]. We also evaluated LG's sensitivity to profiling. Although not shown here for lack of space, the results (available in [17]) indicate that LG is relatively insensitive to both the amount of profiling and the profiling input. For example, when profiling with the same 54 configurations as GG, the energy savings are reduced by 1%-2% on average (maximum 4%).

For adaptation done once per frame, we ignore time and energy overheads for both architecture adaptation and DVS since they are very small compared to the time and energy for a frame. For temporally local adaptation, we model a delay of 5 cycles to activate any deactivated resource. The results are not very sensitive to this parameter. We also model the energy impact of the extra bits required in each instruction window entry for LL instruction window size adaptation (four bits, as in [30]). Other energy overheads for controlling temporally local adaptation are likely to be small, and so are ignored as explained in detail in [30].

4.2 Workload and Experiments

Table 2 summarizes the applications and inputs we use. These were also used in [18, 20, 30] and are described in more detail in [18] (for some applications, we use fewer frames, and for G728 codecs we use only one frame type – we combine one frame of each type from [18] for each frame here). We do not use multimedia instructions because most of our applications see little benefit from those available in the SPARC v9 ISA our simulator uses and we lack a power model for multimedia enhanced functional units.

In our experiments, we assume a soft real-time system where each application has frames arrive with a fixed period. We assume an operating system scheduler allocates a fixed amount of time to process each frame (the deadline).[2] This deadline depends on the system load; thus, for our experiments we use two sets of deadlines. The first set is the tightest deadlines for which the base processor still makes the deadline for all frames. We refer to this as the tight set. For the second, loose set, we double each of the tight deadlines. Table 2 gives the tight deadlines and mean temporal slack on the base processor for both sets of deadlines. The table shows that some applications have a lot of temporal slack even with the tight deadlines. This is due to per frame execution time variability, and also, for MPG codecs, to multiple frame types, with some requiring less execution time. If the execution time exceeds the deadline, the deadline is missed. We allow up to 5% of deadlines to be missed.

We evaluate the algorithms both with and without DVS support (all experiments have architecture adaptation support). With DVS, we use only the loose deadlines since there is generally little room for voltage and frequency adaptation with the tight deadlines. Also, for experiments with DVS, we compare against a base system with no architecture adaptation, but with DVS controlled by GG.

[2]There are several interactions between the scheduler and adaptive hardware that we do not consider [33]. For example, temporal slack from a frame may be given to other applications or frames through buffering. This interaction is beyond the scope of this study, and part of our future work.

5 Results

5.1 Tuning of LL and GG+LL

To evaluate LL and GG+LL, we need to choose values for all (six) of the thresholds as described in Section 2.2.1. No previous work describes a systematic way of tuning such thresholds.

We first attempt to automate the tuning process. As discussed in Section 2.2.1, there are a number of difficulties in designing a tuning algorithm. Therefore, we proceed in two steps. The first step is to narrow the range of values for each of the thresholds, culling values that keep almost all of a resource activated or deactivated. Given this pruned design space, the second step is to evaluate a number of random points within it, and take the point that gives the least energy consumption while missing no more than 5% missed deadlines for all applications (for the tight deadlines). We elaborate on this process next.

In the first step, we use a binary search to find the maximum (and minimum) threshold value that keeps some of the resource activated (or deactivated) on average. This process reduces the design space from about 2^{57} points (1.4×10^{17}) to about 2^{45} points (5.6×10^{13}), still an enormous number.

We now choose random points within the pruned design space. We do this in two ways. (1) Randomly choose the thresholds for one resource at a time, then combine them. The best thresholds are the ones that give least average energy across applications and less than 5% missed deadlines for all applications, or, if no set of thresholds makes enough deadlines, the ones that give the fewest missed deadlines on average. (2) Randomly choose all thresholds together, and take the best set.

We evaluate 128 random points for each resource, and another 128 for the resources together, on a subset of our applications (GSMdec, G728dec, G728enc, and H263dec). The second method described above fails to find a single set of thresholds that gives less than 5% missed deadlines (or even comes close to this criterion) for any of the four applications; therefore, we do not consider it further. The first method finds a set of thresholds that meets the maximum deadline miss requirement for three of the four applications with which we tune. For the fourth application, H263dec, we get a large fraction of misses (23%).

We also manually tuned the thresholds. Our goal was to find a set of thresholds that gives good energy savings (the most we can find) while meeting our deadline miss criterion. This process required weeks of effort. We tuned the LL algorithms for LL alone for all nine applications, and when we use them in GG+LL, they cause more than 5% deadline misses for one application, H263dec (7.4%). Due to the time intensive nature of manual tuning, we did not manually re-tune the algorithms for GG+LL.

App	Tight w/o DVS			Loose w/ DVS		
	GG+LL		LG	GG+LL		LG
	A	M		A	M	
GSMdec	0.3	0.3	1.2	6.2	1.8	1.9
GSMenc	99.5	4.3	0.6	4.2	0.4	0.4
G728dec	4.2	3.6	4.7	3.6	4.2	4.2
G728enc	3.1	4.2	1.6	0.5	0.0	0.0
H263dec	23.4	7.4	4.3	2.1	2.1	2.1
H263enc	2.2	2.2	4.4	3.3	3.3	3.3
MPGdec	6.0	2.6	5.0	0.0	0.0	0.0
MPGenc	0.8	0.0	0.0	6.8	0.8	0.0
MP3dec	3.1	2.8	4.5	0.5	0.7	0.7

Table 3. Missed deadlines (%) for automatically (A) and manually (M) tuned GG+LL and for LG. For loose deadlines and no DVS, $<$ 1% deadlines are missed for all applications and algorithms.

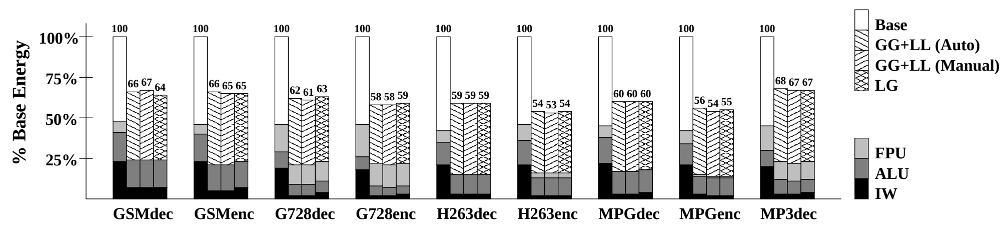
5.2 Missed Deadlines

Table 3 gives the fraction of deadlines missed for GG+LL (with both Auto and Manual tuning) and for LG. The most interesting case is for tight deadlines, since that is where deadline misses are most sensitive to the architecture adaptation control algorithm. Only LG stays within the 5% deadlines missed goal for all applications. GG+LL with Auto tuning misses more than 5% of the deadlines for three applications (GSMenc, H263dec, and MPGdec), missing a lot more than the 5% limit for two of them. Of particular interest is the near 100% missed deadline rate for GSMenc – the performance of GSMenc is more sensitive to the instruction window size than the applications used for tuning, and too much is deactivated for it. This highlights the unpredictability of heuristics-based LL algorithms; even when such an algorithm is tuned to give few deadline misses for a set of test applications, it may behave quite differently for other applications. GG+LL with Manual tuning only exceeds the 5% limit for one application, as discussed earlier.

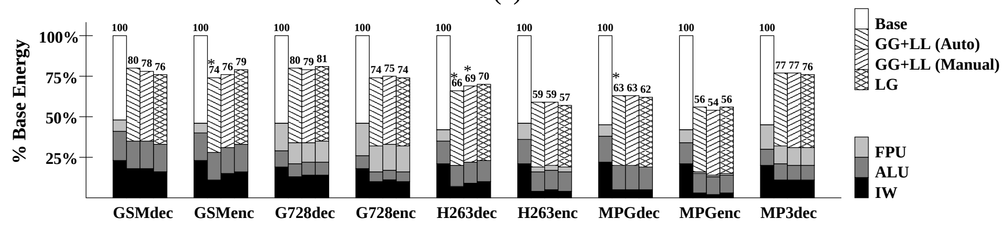
5.3 Energy Savings

Figure 3 shows the energy consumption of Base (the non-adaptive processor), GG+LL with Auto tuning, GG+LL with Manual tuning, and LG, all normalized to Base for each application. Table 4 gives the energy savings for LG relative to the base processor and other algorithms, including GG and LL alone (detailed results not shown since a previous study already showed GG+LL does better than both [30]).

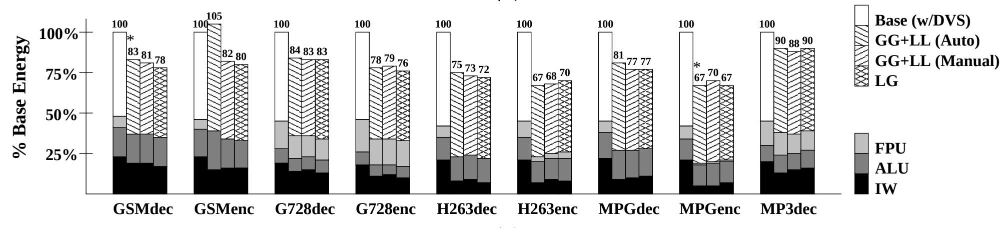
For all combinations of deadlines and DVS, GG+LL with Auto tuning, GG+LL with Manual tuning, and LG give similar energy savings for all applications, with one notable exception explained below (GSMenc). Recall, however, that GG+LL with Auto tuning misses too many deadlines for some applications (as indicated in Table 3) so these



(a)



(b)



(c)

Figure 3. Energy consumption, normalized to Base. (a) With loose deadlines and no DVS. (b) With tight deadlines and no DVS. (c) With loose deadlines and DVS. The shaded parts of each bar indicate the energy consumed by the instruction window, ALUs, and FPUs. For some of the experiments, GG+LL misses too many deadlines – these points are indicated with a star above the bar.

LG savings relative to energy of	without DVS				with DVS	
	Loose		**Tight**		**Loose**	
	Avg	Max	Avg	Max	Avg	Max
Base	39	46	30	44	23	33
GG	4	10	10	17	9	21
LL	18	23	6	16	2	6
GG+LL (Auto)	0	3	-1	5	4	24
GG+LL (Manual)	-1	4	0	3	1	5

Table 4. Average energy savings (%) for LG.

energy savings results are for reference only. Although not shown for lack of space, the mean configurations chosen by all of these algorithms are similar in almost all cases, leading to similar energy savings.

For GSMenc with DVS, GG+LL with Auto tuning uses more energy than even Base (which uses DVS in this case). To compensate for the large IPC degradation it gives GSMenc, GG+LL with Auto tuning chooses a much higher frequency for than LG (645 MHz mean across all frames for GG+LL vs. 502 MHz for LG).

For reference, we also report average and maximum energy savings for LG relative to both GG and LL alone. LG does better than both GG alone and LL alone, with relative savings depending on the amount of temporal slack (up to 21% over GG and up to 23% over LL).

App	**GG+LL**	**LG**	**App**	**GG+LL**	**LG**
GSMdec	2%	6%	GSMenc	4%	4%
G728dec	5%	4%	G728enc	6%	16%
H263dec	3%	9%	H263enc	10%	11%
MPGdec	2%	7%	MPGenc	3%	9%
MP3dec	4%	16%	**Mean**	**4%**	**9%**

Table 5. Temporal slack spreading for GG+LL and LG, given as the standard deviation of per interval IPC degradation.

5.4 Temporal Slack Spreading

LG strategically chooses how much temporal slack to use for each interval based on the energy-performance tradeoffs for all intervals; thus, it “spreads” the slack across each frame. LL, in contrast, is not intended to use temporal slack; thus, GG+LL should not spread slack any more than GG.

We quantify this effect for LG and for GG+LL, for comparison. For one frame of each application, we use the architecture chosen by GG (as part of GG+LL) as a baseline; i.e., we run the chosen frame without adaptation using the architecture chosen by the GG part of GG+LL. We then compute, for each interval, the IPC degradation given by each algorithm relative to the above baseline. Finally, we compute the standard deviation of the per interval IPC degradation expressed as a percentage of the mean. This is a measure of how much temporal slack spreading LG and GG+LL provide. Table 5 shows the results.

We see that although LG spreads slack more than GG+LL in almost all cases, GG+LL *does* spread temporal slack – up to 10% in one case. Tuning GG+LL actually adjusts how it spreads slack. Thus, even though it spreads slack in a very ad hoc manner, with enough tuning it is able to achieve almost the same energy savings as LG, which inherently spreads slack strategically.

6 Related Work

Most of the adaptation control algorithms proposed in other studies have been for specific adaptive resources and have targeted either temporal slack (for DVS) or resource slack (for architecture adaptation), but not both [1, 2, 3, 5, 6, 9, 11, 14, 23, 24, 25, 26, 27, 28, 32].

There has been some recent work on controlling DVS optimally [14, 23]. However, to our knowledge, there is no such work for architecture adaptation.

In the architecture adaptation space, the most closely related work is by Huang et al. [15, 16]. They propose DEETM, a spatially global algorithm for general applications [15]. DEETM distributes temporal slack evenly across all intervals (i.e., targets the same temporal slack for each interval). The intervals are large (on the order of milliseconds), making this algorithm temporally global.

More recently, Huang et al. developed an algorithm for controlling architecture adaptation for general applications. It adapts at the temporal granularity of subroutines and can spread temporal slack across subroutines [16]. The algorithm associates configurations with subroutines, analogous to what we do for each PC value, but with much larger intervals. When mapping configurations to subroutines, it considers independent resources separately – for dependent resources it examines all possible combinations of their configurations. It chooses the set of subroutine-configuration pairs that saves the most energy across the entire application for a given amount of slack. This differs from our LG algorithm in two important ways. First, it runs at a coarser temporal granularity, and so does not exploit short term variability. Second, it assumes that most adaptations are independent of one another. Under these conditions the search space is small enough that an exhaustive search (through all subroutine-adaptation pairs) for an optimal solution becomes feasible. However, the adaptations we consider are interactive, and for our applications, using such a coarse temporal granularity would fail to capture the full energy-savings potential. Further, our application with the smallest search space has more than 10^{138} configuration-interval pairs, making exhaustive search infeasible. We avoid using exhaustive search by exploiting insights into multimedia applications to apply an efficient optimization algorithm. Furthermore, we integrate control of DVS into our algorithm.

Dropsho et al. propose LL algorithms for the issue queues, load-store queue, ROB, register file, and caches [6]. They target general applications, and do not exploit temporal slack. The algorithms for the caches use control feedback to obviate the need for tuning. However, they require accurate prediction and measurement of the temporal slack used for each interval. These are easy to obtain for the cache adaptation considered, but it is unclear how to obtain them for other resources.

There is also substantial work for adapting other parts of the system, and integrating it with the processor is a key part of future work. Most of this work is also thresholds-based and requires a lot of tuning [8, 12].

7 Conclusions

In this paper, we study adaptation control algorithms for adaptive general-purpose processors running real-time multimedia applications. To our knowledge, we are the first to take a formal approach to architecture adaptation control for saving energy. We pose control algorithm design as a constrained optimization problem and solve it using a standard technique, the Lagrange multiplier method. This technique assumes knowledge of energy-performance tradeoffs for the different hardware configurations at all points in all frames. We show how to estimate these tradeoffs using properties of real-time multimedia applications; however, our technique is likely extendible to other application domains by leveraging recent work on phase detection.

It is hard to control high overhead adaptations during a frame; thus, we do so at the frame granularity. At the beginning of each frame, we split the available temporal slack between DVS and architecture adaptation (close-to-optimally), choose a voltage/frequency for the entire frame, and adapt the architecture during the frame.

We compare our algorithm's ability to make deadlines and save energy to the best previously proposed algorithm for real-time multimedia applications. The previous algorithm uses heuristics instead of our more formal optimization approach. These heuristics compare processor statistics to a set of thresholds, which require tuning to adjust the energy savings and missed deadlines. While our algorithm tries to optimally "spread" the available temporal slack across each frame, the previous algorithm spreads slack in an ad hoc manner.

While our algorithm meets the soft real-time requirements for our experiments, the previous algorithm's results depend on its tuning. With an automated tuning process, the previous algorithm is unable to always meet the requirements on missed deadlines, in one case missing nearly

100%. With hand-tuning, a process which takes an impractically long time, it is able to meet the deadline requirements for all but one application. Our algorithm, which needs little tuning, saves almost the same energy as the previous one even when it is hand-tuned, making it practical to implement.

There are a number of avenues of future work, including integrating processor adaptation with adaptation in the rest of the system, handling interaction with the operating system, providing real-time guarantees, and extending this work to other application domains.

Acknowledgements

We would like to thank Douglas Jones for leading us to the Lagrange multiplier method and other members of the GRACE project.

References

[1] D. H. Albonesi. Selective Cache Ways: On-Demand Cache Resource Allocation. In *Proc. of the 32nd Annual Intl. Symp. on Microarchitecture*, 1999.

[2] R. I. Bahar and S. Manne. Power and Energy Reduction Via Pipeline Balancing. In *Proc. of the 28th Annual Intl. Symp. on Comp. Architecture*, 2001.

[3] D. Brooks and M. Martonosi. Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance. In *Proc. of the 5th Intl. Symp. on High Performance Comp. Architecture*, 1999.

[4] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proc. of the 27th Annual Intl. Symp. on Comp. Architecture*, 2000.

[5] A. Buyuktosunoglu et al. An Adaptive Issue Queue for Reduced Power at High Performance. In *Proc. of the Workshop on Power-Aware Computer Systems*, 2000.

[6] S. Dropsho et al. Integrating Adaptive On-Chip Storage Structures for Reduced Dynamic Power. In *Proc. of the Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2002.

[7] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and Predicting Program Behavior and its Variability. In *Proc. of the 12th Annual Intl. Symp. on Parallel Architectures and Compilation Techniques*, 2003.

[8] X. Fan, C. S. Ellis, and A. R. Lebeck. Memory Controller Policies for DRAM Power Management. In *Proc. of the Intl. Symposium on Low Power Electronics and Design*, 2001.

[9] D. Folegnani and A. González. Energy-Efficient Issue Logic. In *Proc. of the 28th Annual Intl. Symp. on Comp. Architecture*, 2001.

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

[11] K. Govil, E. Chan, and H. Wasserman. Comparing Algorithms for Dynamic Speed-Setting of a Low-Power CPU. In *Proc. of the 1st Intl. Conf. on Mobile Computing and Networking*, 1995.

[12] S. Gurumurthi et al. DRPM: Dynamic Speed Control for Power Management in Server Class Disks. In *Proc. of the 30th Annual Intl. Symp. on Comp. Architecture*, 2003.

[13] T. R. Halfhill. Transmeta Breaks x86 Low-Power Barrier. *Microprocessor Report*, February 2000.

[14] C.-H. Hsu and U. Kremer. The Design, Implementation, and Evaluation of a Compiler Algorithm for CPU Energy Reduction. In *Proc. of the SIGPLAN'03 Conf. on Prog. Language Design and Implementation*, 2003.

[15] M. Huang, J. Renau, S.-M. Yoo, and J. Torrellas. A Framework for Dynamic Energy Efficiency and Temperature Management. In *Proc. of the 33rd Annual Intl. Symp. on Microarchitecture*, 2000.

[16] M. C. Huang, J. Renau, and J. Torrellas. Positional Processor Adaptation: Application to Energy Reduction. In *Proc. of the 30th Annual Intl. Symp. on Comp. Architecture*, 2003.

[17] C. J. Hughes. *General-Purpose Processors for Multimedia Applications: Predictability and Energy Efficiency*. PhD thesis, University of Illinois at Urbana-Champaign, 2003.

[18] C. J. Hughes et al. Variability in the Execution of Multimedia Applications and Implications for Architecture. In *Proc. of the 28th Annual Intl. Symp. on Comp. Architecture*, 2001.

[19] C. J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM: Simulating Shared-Memory Multiprocessors with ILP Processors. *IEEE Computer*, February 2002.

[20] C. J. Hughes, J. Srinivasan, and S. V. Adve. Saving Energy with Architectural and Frequency Adaptations for Multimedia Applications. In *Proc. of the 34th Annual Intl. Symp. on Microarchitecture*, 2001.

[21] Intel XScale Microarchitecture. http://developer.intel.com/design/intelxscale/benchmarks.htm.

[22] B. S. Krongold, K. Ramchandran, and D. L. Jones. Computationally Efficient Optimal Power Allocation Algorithms for Multicarrier Communication Systems. *IEEE Trans. on Communications*, January 2000.

[23] J. R. Lorch and A. J. Smith. Operating System Modifications for Task-Based Speed and Voltage Scheduling. In *Proc. of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2003.

[24] Z. Lu, J. Hein, M. Humphrey, M. Stan, J. Lach, and K. Skadron. Control-Theoretic Dynamic Voltage and Frequency Scaling for Multimedia Workloads. In *Proc. of the 2002 International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, 2002.

[25] S. Manne, A. Klauser, and D. Grunwald. Pipeline Gating: Speculation Control for Energy Reduction. In *Proc. of the 25th Annual Intl. Symp. on Comp. Architecture*, 1998.

[26] R. Maro, Y. Bai, and R. Bahar. Dynamically Reconfiguring Processor Resources to Reduce Power Consumption in High-Performance Processors. In *Proc. of the Workshop on Power-Aware Computer Systems*, 2000.

[27] T. Pering, T. Burd, and R. Brodersen. Voltage Scheduling in the lpARM Microprocessor System. In *Proc. of the Intl. Symposium on Low Power Electronics and Design*, 2000.

[28] D. Ponomarev, G. Kuck, and K. Ghose. Reducing Power Requirements of Instruction Scheduling Through Dynamic Allocation of Multiple Datapath Resources. In *Proc. of the 34th Annual Intl. Symp. on Microarchitecture*, 2001.

[29] D. G. Sachs et al. GRACE: A Cross-Layer Adaptation Framework for Saving Energy. *IEEE Computer (a sidebar)*, Dec. 2003.

[30] R. Sasanka, C. J. Hughes, and S. V. Adve. Joint Local and Global Hardware Adaptations for Energy. In *Proc. of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2002.

[31] T. Sherwood, S. Sair, and B. Calder. Phase Tracking and Prediction. In *Proc. of the 30th Annual Intl. Symp. on Comp. Architecture*, 2003.

[32] M. Weiser et al. Scheduling for Reduced CPU Energy. In *Proc. of the 1st Symposium on Operating Systems Design and Implementation*, 1994.

[33] W. Yuan et al. Design and Evaluation of A Cross-Layer Adaptation Framework for Mobile Multimedia Systems. In *Proc. of the SPIE/ACM Multimedia Computing and Networking Conference*, 2003.