Spreading Slack for Optimal Energy-Performance Tradeoffs for Multimedia Applications[*]

Christopher J. Hughes and Sarita V. Adve
Department of Computer Science
University of Illinois at Urbana-Champaign
{cjhughes,sadve}@cs.uiuc.edu

Abstract

Much research has recently been done on the adaptation of architectural resources of general-purpose processors to save energy at the cost of increased execution time (e.g., deactivating entries of the instruction window). This work examines control algorithms for adaptive general-purpose processors running real-time multimedia applications. The best previous algorithms are mostly heuristics-based and ad hoc. They require a huge tuning effort for the heuristics, making them impractical to implement.

Our goal is to take a more formal approach that does not require the application and resource specific tuning of previous approaches, and yet obtain energy savings at least as good as the best previous approach. We first pose control algorithm design as a mathematical constrained optimization problem: what configuration should we use 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 problem through 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, and use it to apply the solution.

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 algorithm provides very similar energy savings through a more formal approach that does not need such heroic tuning effort, making it practical to implement.

1 Introduction

Multimedia applications are an important workload for a variety of systems employing general-purpose processors [7, 8, 20]. Many of these systems are mobile, making battery life an important issue. This motivates increasing the energy efficiency of general-purpose processors running multimedia applications. A number of different hardware adaptation techniques have recently been proposed to increase the energy efficiency of general-purpose processors. A key to effective adaptation is the control algorithm, which must determine when to adapt

[*]This work is supported in part by an equipment donation from AMD, a gift from Motorola Inc., the National Science Foundation under Grant No. CCR-0096126, CCR-0209198, CCR-0205638, EIA-0224453, and the University of Illinois. Sarita V. Adve is also supported by an Alfred P. Sloan Research Fellowship and Christopher J. Hughes was supported by an Intel graduate fellowship.

and what resource to adapt. Previous work has referred to the former as the temporal granularity and the latter as the spatial granularity of adaptation [28].

1.1 Previous Control Algorithms

Previously proposed adaptation control algorithms for multimedia applications are based on the observation that there are two ways in which hardware can be energy-inefficient.

Slack for a frame: Hardware may run faster than needed, incurring a higher power dissipation than necessary. This is particularly true for real-time multimedia applications, which need to process discrete units of data, generally called frames, within a deadline; we need only meet the deadline, not beat it. The difference between the required and the actual execution time is referred to as slack. Techniques have been proposed to save energy by reducing slack.

In the context of real-time multimedia applications, slack is easily predictable at the granularity of a frame [16]. Therefore, previous algorithms that exploit slack for these applications operate at the frame granularity, picking one configuration for the entire frame – the configuration that will result in the least energy consumed, and slow the computation down by no more than the available slack. Thus, these algorithms are global at both the temporal and spatial granularity because they adapt once per frame and choose the configuration for all resources in an integrated way. We refer to them as GG.

One advantage of GG algorithms is that since they adapt infrequently they can control adaptive hardware even if it has a relatively large time overhead for adaptation (e.g., dynamic voltage and frequency scaling, or DVS, which may take up to 10μs to adapt [13]). A disadvantage is that the adaptations GG algorithms use must be applicable to full frames (e.g., they cannot deactivate all FPUs).

Varying resource utility within a frame: Some hardware resources may be active (consuming energy), but may not contribute sufficiently to performance (i.e., instructions per cycle, or IPC). Techniques have been proposed to recognize and react to this, deactivating such resources to save energy with little performance impact.

A resource's contribution to performance, which we call its utility, generally varies at a granularity much finer than a frame; thus, these algorithms track utility over small intervals. At this granularity, however, it is generally difficult to predict the impact on IPC from a change in a resource's configuration. It is even more difficult to predict the impact on IPC from changes in multiple resources. Therefore, these algorithms are reactive in nature; they use heuristics to control each resource independently and only try to keep the changes from having "much" impact on performance. These algorithms are local at both the temporal and spatial granularity because they adapt frequently during a frame and control resources independently (e.g., [9, 10, 24, 28]). We refer to them as LL.

In contrast to GG algorithms, an advantage of LL algorithms is that they can control adaptations that are not applicable for full frames. Similarly, a disadvantage of LL algorithms is that they cannot control high overhead adaptations due to their frequent adaptation. A key problem for heuristic-based LL algorithms is that they require a large amount of tuning effort to ensure that their performance impact is limited. This effort grows exponentially with the number of algorithms (i.e., adaptive resources) if they do not behave independently.

Combining frame slack and variable resource utility: GG algorithms do not exploit local resource utility variations and LL algorithms do not exploit slack. This motivated a combined GG and LL, or GG+LL, algorithm [28]. It uses performance and energy prediction to pick a global configuration at the frame granularity to minimize slack, and lets the LL algorithms react to behavior during a frame to further reduce resources without (ideally) further degrading execution time. This combination also allows both the use of high overhead adaptations and adaptations that are inappropriate for an entire frame.

This algorithm is successful at exploiting resource variability and removes most of the slack. It is the best previous adaptation control algorithm for multimedia applications. However, it has a key drawback – its use of the reactive LL algorithms necessitates the tuning process for the heuristics. Each heuristic periodically compares one or more execution statistics to a set of thresholds. The design space for the thresholds for the system evaluated here is very large (2^{57} points) due to interaction between the adaptive architecture resources. Also, it is hard to predict the change to performance and energy when moving from one design point to another. (Tuning is discussed in Section 2.2.1.) The difficulty in tuning LL algorithms makes an implementation practically challenging, especially for systems with many adaptive resources.

1.2 Our Contribution

Our contribution is to develop an algorithm that can provide at least the energy savings of GG+LL but without the enormous tuning effort. We do this by taking a more formal approach to the adaptation control problem. We start by taking a different view of the problem, merging the two sources of energy inefficiency. Previously, for multimedia applications, slack has been exploited at only the frame level because it was not known how to predict the performance and energy impact of adaptation at a finer granularity; thus, adaptation during a frame has been controlled in a reactive manner and tried to maintain performance. We show how to a priori predict the performance and energy impact of spatially global adaptation during a frame using properties of multimedia applications. This allows us to exploit slack within a frame, erasing the line between the two sources of inefficiency.

Armed with temporally fine-grained predictions of the impact of adaptation on performance and energy, we can view the problem formally as a constrained optimization problem: given that a frame is composed of many intervals, we want to choose one configuration for each interval to maximize total energy savings while using only the available slack. We solve this using formal optimization techniques. Our predictive approach and the use of formal optimization techniques leads to an algorithm that does not require the heroic tuning effort of the above previous algorithms.

Returning to the classification scheme for adaptation algorithms, this algorithm is temporally local and spatially global, so we call it LG. It fills out the design space previously established [28].[1]

A disadvantage of LG adaptation is that it cannot control high overhead adaptations due its temporally local nature. However, in the same way that GG+LL allows the use of high overhead adaptations, we propose a combined

[1]There is little basis for a temporally global and spatially local scheme, so we do not consider one. A temporally and spatially global scheme already exists; it will do as well or better since it runs at the same temporal granularity, but also considers interaction between resources (is spatially global).

		Spatial Granularity	
		Global	**Local**
Temporal Granularity	**Global**	Choose entire hardware configuration at beginning of frame. Exploits slack.	Choose configuration for each resource independently at beginning of frame. Little basis for this approach.
	Local	Choose entire hardware configuration at frequent intervals during frame. Exploits slack and variable resource utility.	Choose configuration for each resource independently at frequent intervals during frame. Exploits variable resource utility.

Table 1. Adaptation algorithm design space.

GG+LG algorithm where GG controls only the adaptations that LG cannot (just DVS in our system).

We evaluate the different algorithms, measuring energy savings when each algorithm is permitted to miss no more than 5% of the deadlines. We try systematic methods of tuning the LL algorithms, but they fail to find acceptable design points (far too many deadlines are missed). Therefore, we rely on hand-tuning, which is extremely time consuming. We find that LG (and GG+LG) provide very similar savings to the best (painstakingly) hand-tuned GG+LL (which is better than GG or LL alone). LG uses a more formal approach than GG+LL, and thus needs little effort to tune. Therefore, LG is a much easier to implement alternative to a well-tuned GG+LL, while achieving the same energy savings.

Our algorithm is closest to one proposed by Huang et al. [15]. That algorithm targets general applications (e.g., SPEC). It also spreads slack (i.e., a performance slowdown), but does so across multiple subroutines. The subroutine granularity is likely too coarse to fully exploit the variable resource utility we target, at least for the multimedia applications studied here. The algorithm also has limited support for adaptations that interact – for these, it uses an exhaustive search of all combinations of subroutine and configuration, which is not feasible when using small intervals instead of subroutines and when controlling many interacting adaptations. Instead, our LG algorithm uses properties of multimedia applications to apply the results of an efficient formal optimization algorithm, and thus avoids an exhaustive search. Finally, the previous algorithm does not integrate control of DVS. We discuss it further in Section 6.

2 Design Space and Previous Algorithms

Table 1 summarizes the adaptation control algorithm design space for processors running real-time multimedia applications previously established [28], and discussed in Section 1. The next few sections describe previously proposed LL, GG, and GG+LL algorithms. Section 3 discusses our LG algorithm.

2.1 Previous GG Algorithm

To our knowledge, the only previously proposed GG algorithm for multimedia applications that integrates architecture adaptation and DVS operates as follows [18]. At the beginning of each frame, it predicts the hardware configuration that will minimize energy for that frame without missing the deadline. The algorithm can control two types of adaptive hardware: adaptive architecture resources and dynamic voltage and frequency scaling (DVS). When adapting an architecture resource, a part of the resource is deactivated or reactivated (e.g., some entries of the instruction window are deactivated, effectively decreasing its size). When using DVS, the voltage and frequency of the processor are changed (e.g., the voltage is lowered to save energy, and this necessitates a frequency drop). A *hardware configuration* is a combination of the configuration for all architecture resources and the voltage and frequency. Thus, the GG algorithm chooses a single hardware configuration for each frame, thereby adapting all adaptive resources together.

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[2] for each architecture configuration A, at some base voltage and frequency. The algorithm collects the instructions per cycle (IPC_A) and average power (P_A) for each frame.

Previous work showed that for several multimedia applications and systems, for a given frame type, the average IPC and power for a configuration are roughly constant for all frames [16, 18]. Thus, the IPC_A and P_A values from the profiling phase can be used to predict the IPC_A and P_A of all other frames of that type. Previous work also showed that IPC is almost independent of frequency for the applications considered here, so IPC_A can be used to predict the IPC for architecture A at all frequencies. 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, the algorithm computes the most instructions executable within the deadline (I_{max}) as $I_{max_H} = deadline \times IPC_A \times f_H$. It also computes the energy consumed per instruction (EPI) for each configuration as $EPI_H \propto \frac{P_A \times V_H{}^2}{IPC_A}$. It then constructs a table with an entry for each configuration containing its I_{max} and EPI, sorted in order of increasing EPI.

After profiling is complete, the algorithm enters the adaptation phase. For each frame, it predicts the number of instructions, using a simple history-based predictor (takes the maximum of the last five frames 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 still meet the deadline, and so chooses it.

For processors that support a large number of frequency choices (e.g., XScale [19]), the number of configurations may be too large to use this table-based approach for choosing the configuration. A variation of this algorithm for such processors is given in [18]. We omit its description here for lack of space.

This GG algorithm can be implemented in software or hardware. It requires communication from the application or operating system to indicate when a new frame starts, the type of the frame, and the deadline. The application already needs to communicate the start of frame to the operating system (which already knows the deadline), so this requires little or no change to the application.

2.2 Previous LL Algorithms

A number of researchers have studied algorithms to control individual adaptive architectural resources, although 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) [6, 9, 10, 26, 28], changing the number of

[2] 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 profiles and adapts for each frame type separately.

functional units and/or issue width [3, 24, 28], and others [2, 4, 11, 14, 23]. In this study, we focus on changing the instruction window size and the number of active functional units (and issue width).

Sasanka et al. propose the best current LL algorithms for adapting both the instruction window size and the number of active functional units for multimedia applications [28]. For controlling the window size, the algorithm combines two heuristics: one each for determining when the size should be increased or decreased. The increase heuristic is based on estimating the lost performance from having some of the window deactivated (the details of the elaborate estimation method are not important for this study). If the performance loss in an interval is larger than a *threshold*, the window size is increased. The decrease heuristic is from [10]. It counts the number of instructions issued from the youngest part of the window. If at the end of an interval this number below a *threshold*, there is likely little benefit from having so large a window, and its size is decreased.

For controlling the number of active functional units, the algorithm again uses two heuristics: one each for determining when the number of active units should be increased or decreased. The same algorithm is used independently for both integer ALUs and FPUs. The increase heuristic is based on the number of times instructions are ready to be issued to a functional unit, but no unit is free (i.e., issue hazards). The larger the number of issue hazards, the larger the performance loss from deactivated units (most likely). If at the end of an interval the number of hazards is larger than a *threshold*, the number of active units is increased. The decrease heuristic is from [6]. It tracks functional unit utilization. If at the end of an interval, this is smaller than a *threshold*, there is likely little benefit from having so many units active, and the number is decreased.

The above algorithms operate at fixed time intervals (in our experiments, every 256 cycles). During an interval, each algorithm independently collects and computes statistics (e.g., the number of issue hazards), and at the end it compares the statistics to a set of *thresholds*. The outcome of these comparisons determines the configuration chosen for the next time interval. The LL algorithms consider changing the configuration of the targeted resource by only a single step per interval. For the instruction window, which is composed of equal sized segments, this means increasing or decreasing the window size by one segment. For functional units, this means increasing or decreasing the number of active units by one.

The LL algorithms must be implemented in hardware due to the nature of the heuristics used and the frequent adaptations. In addition to the hardware required to make each resource adaptive (needed by all adaptation algorithms), the LL algorithms need hardware added at each resource to collect and process statistics.

2.2.1 Tuning

The various thresholds of the LL algorithms 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 instruction window heuristics use two thresholds: maximum stall cycles from the shrunken window (maximum value is the interval length, which is 2^8 in our system) and minimum instructions issued from the youngest segment (maximum value is the interval length times the fetch width, which is 2^{11} in our system). Thus, for the system modeled here, there are 2^{19} possible design points for this algorithm. The functional unit heuristics also use two thresholds: maximum number of issue hazards (maximum value is the interval length times the fetch width) and minimum cycles utilized (maximum value is the interval length). Thus, for the system modeled here, there are 2^{19} possible design points for the functional unit algorithm as well (one set each for ALUs and FPUs).

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. 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 of the thresholds is neither linear nor monotonic. That is, it is hard to predict both the magnitude and direction of the change to energy when we change a threshold. 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} design points. In general, if all thresholds are tuned together, the tuning effort grows exponentially with the number of adaptive 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 heusristics 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 [28]. 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 slack and variable resource utility during a frame. However, it requires tuning due to the use of LL algorithms. In addition, the LL algorithms may need to be retuned once integrated with GG.

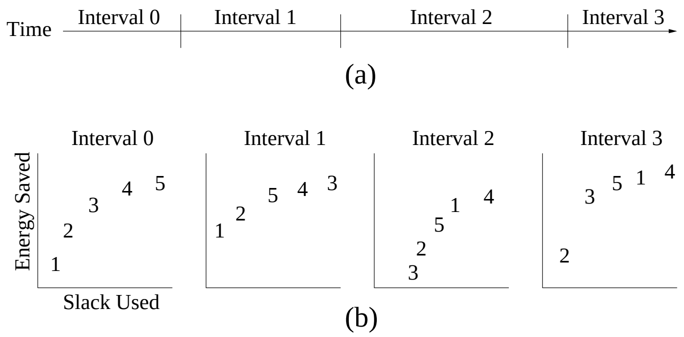

Figure 1. (a) Example frame composed of four intervals. (b) Energy-performance tradeoffs for the intervals. The numbers correspond to different configurations.

3 LG Algorithm

We now present our new algorithm for controlling architecture adaptation in a temporally local, spatially global, or LG, manner, and its integration with GG. Sections 3.1 through 3.5 describe LG control of architecture adaptation. Section 3.1 poses our problem as a constrained optimization problem, and Section 3.2 describes a solution to it assuming knowledge of the available slack versus energy tradeoffs, and the available slack for each frame. Section 3.3 discusses how the algorithm obtains the slack versus energy tradeoffs for real-time multimedia applications, and Section 3.4 explains how the algorithm obtains the slack for each frame. Section 3.5 summarizes the complete algorithm. Section 3.6 explains one way to reduce the profiling effort required by the algorithm. Section 3.7 gives the overheads of the algorithm, and Section 3.8 describes the design parameters for the algorithm. Section 3.9 discusses the limitations of the algorithm. Finally, Section 3.10 explains how we combine LG with GG to control high overhead adaptations like DVS.

3.1 The Optimization Problem

Each frame is divided into multiple intervals, each with the same number of instructions (1024 in our system). Figure 1 shows an example set of intervals, with the energy saved vs. slack used data for each architecture configuration (with respect to the base configuration). We want to find the set of configurations, one for each interval, that gives the most energy saved while using no more than the target (or available) slack for the frame, S_{target}. We discuss how to predict S_{target} for each frame in Section 3.4.

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, C_i is the configuration for interval i, $E_i(C_i)$ is the average energy saved per instruction (EPI-saved) compared to the base, non-adaptive configuration for interval i, and $S_i(C_i)$ is the average slack used per instruction (SPI-used) compared to the base configuration for interval i.

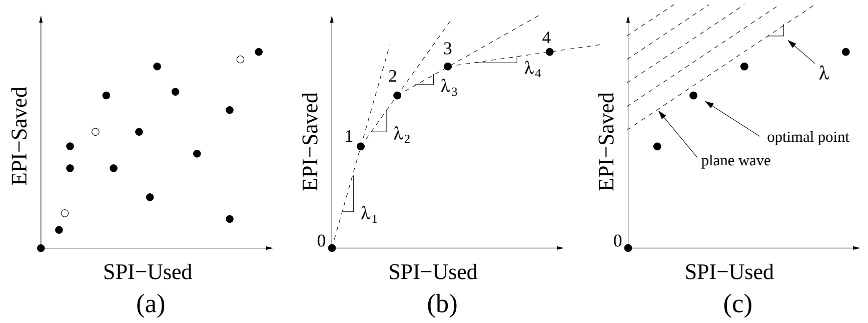

Figure 2. (a) EPI-saved versus SPI-used data for an example interval (each point corresponds to a configuration), and (b) the remaining configurations, with the λ_i, after extracting the convex envelope. The number above each point is the configuration. (c) A plane wave approaching the EPI-saved versus SPI-used curve of an interval.

3.2 The Solution

Our approach to solving this is inspired by work on an equivalent problem in the wireless comunications domain [21]. That work uses a standard technique for solving constrained optimization problems: the method of Lagrange multipliers. The key is that the function of interest (in our case $\sum E_i(C_i)$) is maximized when

$$\nabla \sum_{i=1}^{N} E_i(C_i) \;=\; \lambda \nabla \sum_{i=1}^{N} S_i(C_i)$$

where λ is the Lagrange multiplier. The set of configurations that maximizes total energy savings for each interval for a given λ is the optimal set of configurations for some target slack. The above equation is satisfied when

$$\frac{\partial E_i(C_i)}{\partial C_i} \;=\; \lambda \frac{\partial S_i(C_i)}{\partial C_i} \;\Longrightarrow\; \frac{\partial E_i(C_i)}{\partial S_i(C_i)} \;=\; \lambda \tag{1}$$

for all i. In other words, the total energy savings is maximized when the configuration for each interval is chosen such that its slope on the EPI-saved to SPI-used curve (if we were to create a curve from the discrete points) is λ. This λ is optimal for a target slack corresponding to the mean SPI-used for the selected configurations. However, we are interested in a specific target slack, S_{target}, which requires a search of the λ space. We discuss an efficient way to perform this search later.

One problem with using this solution is that it requires our configurations to form a convex EPI-saved to SPI-used curve for each interval, which they may not do. Figure 2(a) shows example EPI-saved versus SPI-used data for a single interval (this is more realistic than that in Figure 1(b)). To apply the above solution, we remove points not on the convex envelope. The slope between adjacent points of the curve must decrease with increasing SPI-used; points that violate this are pruned. Pruned points are likely suboptimal since there is probably a configuration with higher EPI-saved and lower SPI-used than them. However, some may not be suboptimal; examples of these are indicated with hollow points in the figure. Figure 2(b) shows the remaining points after the pruning process.

As mentioned earlier, a problem equivalent to ours has been solved in the wireless communications domain. The solution includes a way to efficiently determine the optimal configuration for each interval for a given λ [21].

From equation 1, the optimal configuration is the first one which a plane wave with slope λ would intersect the EPI-saved to SPI-used curve, as shown in Figure 2(c). To determine which point this is, the algorithm first computes the slope between adjacent configurations, or λ_i, as shown in Figure 2(b) (this can be done during pruning). The wave will first intersect configuration i if $\lambda_i \geq \lambda > \lambda_{i+1}$. For the endpoints of the curve, the wave will first intersect the configuration with smallest SPI-used if λ is larger than all λ_i. It will first intersect the configuration with largest SPI-used if λ is less than or equal to all λ_i. Thus, a simple table-based search through the λ_i will quickly determine the optimal configuration for each interval for a given λ.

To efficiently search the λ space for the λ optimal for S_{target}, the solution includes a bisection technique [21]. This solution for finding the optimal configuration for all intervals has a complexity of $O(N \log M)$, where M is the number of configuration choices. For comparison, an exhaustive search of all configuration and interval combinations has a complexity of $O(M^N)$, and thus is infeasible to use in practice.

3.3 Obtaining Energy-Slack Tradeoffs

The solution described above 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 us to profile all frames once per configuration before adapting. Instead, we do this for one frame, find the optimal configurations for that frame, and map the solution to other frames at run-time using some insights about real-time multimedia applications. Our technique is likely extendable to other application domains, but that is outside the scope of this paper.

Rather than obtaining perfect EPI-saved and SPI-used information for all frames, we do so for one frame. We profile a single frame once per configuration (i.e., the same frame multiple times). We then run the optimization algorithm on the profiled frame to obtain the optimal configuration for each interval. Finally, we define a key (some property of an interval) to map the intervals of other frames to the intervals of the profiled frame – the key for an interval tells it which configuration to use. Some choices for this key are IPC of the interval on the base configuration (IPC_{base}), starting program counter (PC) value for the interval, and branch history at the start of the interval. We use PC as the key for the following reasons (we also tried IPC_{base} – it gives slightly worse results, and using PC is easier).

Using PC as the key means that the algorithm will choose the same configuration for intervals with the same starting PC value both within a frame and across different frames. For the applications studied, intervals with the same starting PC value often have similar tradeoffs between slack and energy for each configuration, likely due to the repetitive nature of the application algorithms. Thus, within a given frame, the intervals with the same PC will often have the same (or a very similar) optimal configuration. Previously, it was reported that despite potentially large variability in per frame instruction count, the per frame IPC (and EPI) for real-time multimedia applications is almost constant across different frames [16]. 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 is relevant here in that it implies that the fraction of intervals with each starting PC value is likely to be almost the same across different frames. Thus, when we apply the configuration choices made for the profiled frame to other frames using PC as the key, we can expect the average SPI-used and EPI-saved across all intervals (in terms of cycles or energy per instruction) to be the same for the other frames as for the profiled frame. By extension, when adapting, the per frame IPC and EPI for the profiled frame should be almost the same as for other frames. We verified this by using this IPC and EPI as predictions for other frames (see Section 3.4).

Nevertheless, 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).

Since each configuration in the table may be a compromise between multiple intervals' optimal configurations, more slack may be used when adapting than anticipated, which could induce deadline misses. To correct for this, the algorithm uses the following procedure. It determines the configuration that would be used for each interval of the profiled frame. This gives it the slack used for each interval, and it simply sums these up to predict the slack used for the whole frame. If the predicted slack used is more than the target slack, the target is reduced by a small amount and the algorithm is rerun.

3.4 Obtaining Target Slack

Our algorithm assumes a target slack, S_{target}, (in cycles per instruction) as input. For the most likely way one would use our algorithm, this slack will vary from frame to frame because of execution time variability among frames and the presence of multiple frame types. Our approach is to run the optimization algorithm for multiple target slack values (we choose the range of zero to one cycle per instruction, with an interval of 0.01 cycles per instruction). At runtime, we predict the slack for a frame and choose the table for the closest slack (that does not exceed the prediction) for which we ran the optimization algorithm. The slack predictor is inspired by GG. It uses GG's instruction count predictor, and assumes that per frame IPC is almost the same for all frames of the same type. Thus, it predicts S_{target} to be $\frac{deadline}{instructioncount} - \frac{1}{IPC_{base}}$, where IPC_{base} is the IPC for the base configuration for the profiled frame.

We envisage our algorithm also being run in an alternative mode where the user or operating system specifies a target slowdown rather than a deadline. The target slack for this mode will be the same for all frames, and can be computed from the slowdown and a per frame IPC_{base} estimate. In this mode, the algorithm could also predict the energy savings for a given slowdown and let the user or operating system decide if the tradeoff is favorable. Although we omit an evaluation of this mode for lack of space, we observed that the algorithm is able to accurately predict energy savings for a wide range of target slowdowns.

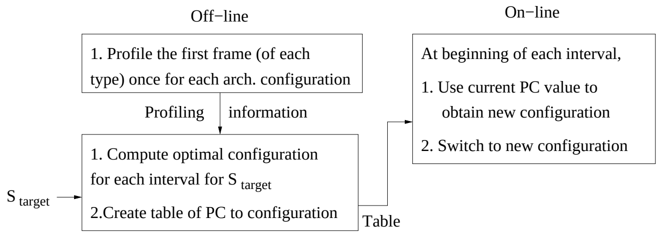
Off−line
1. Profile the first frame (of each type) once for each arch. configuration
Profiling information
1. Compute optimal configuration for each interval for S target
2.Create table of PC to configuration
S target
Table
On−line
At beginning of each interval,
1. Use current PC value to obtain new configuration
2. Switch to new configuration

Figure 3. The new LG algorithm. S_{target} is an input to the algorithm.

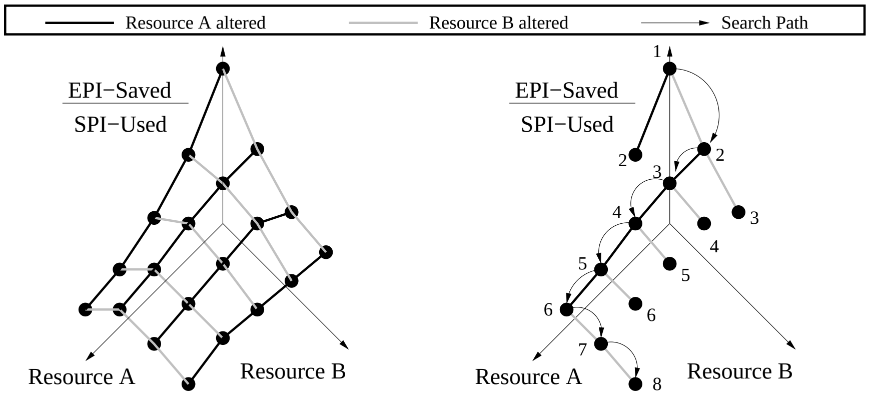

Figure 4. (a) An example adaptation design space. The x and y axes show *decreasing* amounts of the adaptive resources. $\frac{EPI-saved}{SPI-used}$ of the the base configuration is undefined ($\frac{0}{0}$), so we have assigned an arbitrary value to it. (b) The path the pruning algorithm would take and the configurations it would profile. The number by each point indicates the time step at which it is profiled.

3.5 Complete Algorithm

We now summarize the complete LG algorithm, also shown in Figure 3. It takes the target slack, S_{target}, as its input. It first collects profiling information and runs the off-line optimization algorithm with S_{target}. If the target slack is not known a priori (as in our experiments), the optimization algorithm is run with multiple S_{target}. For a given S_{target}, the optimization algorithm constructs a table of configuration choice for an interval indexed by starting PC value for the interval.

During execution, at the beginning of each interval, the algorithm uses the current PC value to obtain the configuration to use for the interval. If the target slack is unknown a priori, during execution the algorithm uses the PC to configuration table for the S_{target} closest to, but not exceeding, the target slack.

3.6 Reducing Profiling Effort

Our LG algorithm may require a lot of of profiling for a processor that supports many configurations. In general, this is not a significant drawback since profiling is done only once per application, and since the number of frames profiled is likely dwarfed by the number of frames processed in a single run of an application. However, if profiling for all possible configurations is undesirable, an alternative is to choose a subset of the configurations, at the cost of potentially reducing the energy savings. A naive way to pick a subset is to pick configurations uniformly over the design space. Instead, we explore a pruning algorithm for choosing configurations to profile based on the fact that

LG culls configurations not on the convex envelope of EPI-saved to SPI-used curves. Thus, the pruning algorithm attempts to profile only configurations that are on the convex envelope of enough intervals.

LG constructs the convex envelope of the EPI-saved to SPI-used curve for each interval by repeatedly choosing the point with largest $\frac{EPI-saved}{SPI-used}$ that is to the right of the last point chosen (i.e., has larger SPI-used). In general, when we deactivate a part of a resource, SPI-used increases; thus, as we travel along the convex envelope each configuration tends to be the same as the previous one with the smallest part of one of the adaptive resources deactivated (e.g., one ALU less).

Figure 4(a) shows example $\frac{EPI-saved}{SPI-used}$ data for a system with two adaptive resources (assume for now that this frame has only one interval). The pruning algorithm searches this space, starting at the base configuration and moving from one adjacent (and simpler) point to another (i.e., deactivates the smallest part possible of one resource at each step) until it reaches the simplest configuration. At each point in the search, the algorithm moves to the adjacent point with largest $\frac{EPI-saved}{SPI-used}$, and profiles the configurations adjacent to (and simpler than) the new point. Figure 4(b) shows the path the algorithm would take for the example data and all configurations that would be profiled. We now explain how we extend the algorithm for frames with multiple intervals.

When a frame has multiple intervals, at each point in the search, each interval may have a different configuration with largest $\frac{EPI-saved}{SPI-used}$. Thus, instead of choosing one "best" configuration, the algorithm computes the fraction of intervals for which each configuration has the largest $\frac{EPI-saved}{SPI-used}$, and chooses the ones whose fractions are above some threshold. We use $\frac{1}{\#\ of\ adaptive\ resources}$ as the threshold since the number of adaptive resources is the maximum number of configurations considered at each point in the search. This means the algorithm may follow multiple paths from the base configuration to the simplest one. If the number of configurations profiled is still too large then the algorithm can further limit the number of search paths taken, at the cost of reduced energy savings.

3.7 Required Support and Overhead

Our LG algorithm requires some special support for both the off-line and on-line portions. Off-line, LG profiles one frame (the same one) per architecture configuration considered. Applications can be restarted for each architecture to avoid modifying them. As discussed above, this is done only once for each application and the number of frames profiled is likely to be small compared to the number of frames processed during the run of the application. We also run the optimization algorithm off-line. Its complexity is relatively small, $O(N \log M)$, where N is the number of intervals in the profiling frame and M is the number of architecture configurations considered. Also, the algorithm is only run once per application. The off-line portion of LG needs to be implemented in software. Besides the ability to adapt the hardware, this requires no special hardware support.

On-line, our LG algorithm predicts the instruction count and S_{target} once per frame, for which the 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 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).

3.8 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 some design parameters that can affect its behavior. In particular, LG has the following parameters: the PC to configuration table size, the block size used when indexing the table, the interval length, and the instruction count predictor used to estimate target slack. Finding good values for these 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 but the instruction count predictor are relatively insensitive to the application (for those considered here). Finally, assuming we restrict the values of the table, block, and interval sizes 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. Larger interval sizes give less energy savings, and smaller ones give more missed deadlines (due to interaction between adjacent intervals).

The response of the algorithm to the final parameter, the instruction count predictor, is application-dependent. The fraction of frames for which the predictor under-predicts, and the amount by which it over-predicts for other frames, depends upon the behavior of the per frame instruction count for an application. Under-predictions may cause missed deadlines, but over-predictions will only reduce energy savings. We use the same predictor for LG as for GG, and it performs well for all of our applications (we did not tune it further). If necessary, it is possible to adapt the predictor to the application or to different missed deadline requirements. The predictor takes the maximum instruction count of the last five frames and adds some leeway. If too many deadlines are being missed, it could be made to take the maximum of more than the last five frames or to add more leeway. Since GG uses the same predictor, if this more complicated predictor is needed for LG, it would also be needed for GG.

3.9 Limitations

Despite LG's basis in an optimal algorithm, it may still give suboptimal results. The most important reasons for this are as follows. First, when constructing convex envelopes of the EPI-saved to SPI-used curves, we might prune the optimal configuration. Second, we sometimes choose a compromise configuration for entries in the PC to configuration table since intervals of the profiled frame with the same PC may have different optimal configurations. This results from our algorithm assuming that the EPI and IPC for an interval are dependent only on the configuration and the starting PC value for the interval. There are two sources of error associated with this: processor behavior for an interval will be affected by both the instructions of and the configuration used for the previous interval, and the control flow path taken within the interval may not always be the same. Third, as discussed earlier, there is a tradeoff between profiling overhead and configurations available to the algorithm. If

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

Table 2. Base (default) system parameters.

the number of possible configurations is very large, only a subset of them may be available for profiling.

Another limitation of LG is that is cannot control adaptive resources that have long switching times, due to its frequent adaptations. In the next section, we address this by combining it with GG.

3.10 Integrating GG and LG

A disadvantage of LG is that it cannot control adaptive resources that have long switching times, due to its frequent adaptations. However, such adaptations are possible to control at the coarser granularity of a frame. Therefore, to enable use of such adaptive resources we propose a combined GG+LG algorithm, where GG is used to control only those adaptive resources that LG cannot (including DVS). It works as follows.

LG runs the optimization algorithm for a set of S_{target} values, as before. For each S_{target} value, it estimates the per frame IPC and EPI that LG will give and gives this information to GG. LG estimates EPI by determining which 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. LG can be viewed as giving different tradeoffs between IPC and EPI for each value of S_{target}. Therefore, GG treats LG as an adaptive resource – the "configuration choices" for it are the S_{target} options. Otherwise, GG runs as described in Section 2.1. In choosing a configuration for each frame it will select the S_{target} to use for LG.

4 Experimental Methodology

4.1 Systems Modeled

We use the RSIM simulator [17] for performance evaluation and the Wattch tool [5] integrated with RSIM for energy measurement.

The base, non-adaptive, processor studied is summarized in Table 2. We model a centralized instruction window with a unified reorder buffer and issue queue, composed of eight entry segments. 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

App.	Type	Input Size		Base	Tight	Mean Slack on Base (%)	
		Time	Frames	IPC	Deadline	Tight	Loose
GSMdec	Speech codec	20s	1000	3.7	15μs	9.8	54.9
GSMenc		20s	1000	4.6	45μs	8.9	54.4
G728dec		0.63s	250	2.3	51μs	10.2	55.1
G728enc		0.63s	250	2.1	65μs	9.2	54.6
H263dec	Video codec	6s	150	3.4	450μs	15.6	57.8
H263enc		6s	150	2.3	18.78ms	25.1	62.5
MPGdec		8.33s	250	3.6	2.11ms	31.2	65.6
MPGenc		8.33s	250	3.0	55.3ms	38.7	69.4
MP3dec	Audio	26.1s	1000	3.0	495μs	22.3	61.1

Table 3. Workload, deadlines, and 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.

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 [19] as further discussed in [18].

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 [10]. 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 GG, LL, GG+LL, LG, and GG+LG algorithms. Since GG+LL was previously found to be better than GG or LL alone [28] we evaluate GG and LL for reference only, using the algorithm in [18] for GG and "Local" in [28] for LL. For GG+LL, we use the "Global+Local" algorithm from [28]. 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. LG also deactivates an FPU if none was used in the last interval, allowing it to deactivate all FPUs (LL has the same ability). We use an interval size of 1024 instructions for LG. Decreasing the interval size makes the interaction between intervals more significant and so increases deadline misses. Increasing the interval size keeps LG from exploiting as much opportunity for adaptation during a frame and so decreases energy savings.

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). For LG, we profile configurations according to the pruning algorithm given in Section 3.6. To test LG's sensitivity to the amount of profiling it does, we also evaluate the LG algorithm with profiling only the same 54 configurations profiled for GG and GG+LL. Detailed results are reported for LG using the pruning algorithm.

For temporally global adaptation (i.e., GG), 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 (i.e., LL and LG), 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 [28]). Other energy overheads for controlling temporally local adaptation are likely to be small, and so are ignored as explained in detail in [28].

4.2 Workload and Experiments

Table 3 summarizes the nine applications and inputs used in this paper. These were also used in [16, 18, 28] and are described in more detail in [16] (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 [16] for each frame here). We do not use multimedia instructions because most of our applications see little benefit from them 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 (the deadline). A deadline is missed if the processing of a frame exceeds the period, and up to 5% of deadline misses is acceptable. The energy savings, both absolute and relative, are sensitive to the slack, or leftover processing time, for the base processor.[3] Therefore, for our experiments we use two different 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 3 gives the tight deadlines and mean slack on the base processor for both sets of deadlines.[4] The table shows that some applications have a lot of 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.

We evaluate the control algorithms for systems both with and without DVS support (all experiments have architecture adaptation support). For experiments without DVS, we evaluate LG, but not GG+LG, and for experiments with DVS, we evaluate GG+LG, but not LG. For experiments with DVS, we use only the loose deadlines since there is 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.

5 Results

5.1 Tuning of LL and GG+LL

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

[3]The deadlines are assigned by the real-time scheduler which must consider the system load [1]. This interaction is beyond the scope of this study.

[4]Some of the deadlines are very short which might make context switch and DVS overhead too large. Although not evaluated here, one solution is to group (buffer) multiple frames, increasing the granularity of scheduling and temporally global adaptation.

Threshold	Description	Possible Range	Narrowed Range	Thresholds Used	
				Auto	Manual
iw1	Maximum stall cycles from shrunken instruction window	[1,255]	[16,255]	26	20
iw2	Minimum instructions issued from youngest instruction window segment	[1,2040]	[1,214]	42	10
alu1	Maximum number of ALU issue hazards	[1,2040]	[1,256]	6	50
alu2	Minimum cycles all ALUs utilized	[1,255]	[1,255]	51	250
fpu1	Maximum number of FPU issue hazards	[1,2040]	[1,65]	3	50
fpu2	Minimum cycles all FPUs utilized	[1,255]	[1,255]	227	250

Table 4. The thresholds for the LL algorithms, their possible values in our system, the ranges to which we initially narrow them down, and the final values for both an automated and a manual search.

We first attempt to automate the tuning process. As discussed in Section 2.2.1, there are a number of difficulties in designing an efficient and effective tuning algorithm. Therefore, our approach is to first narrow the range of values for each of the thresholds, culling values that keep almost all of a resource activated or deactivated. Table 4 lists the thresholds and the possible values for our system. Given a pruned design space, we then 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).

More specifically, by executing the fastest running application (G728dec) with different sets of thresholds we find a starting point for the rest of the automated tuning process. Our goal is to find a set of thresholds such that the mean configuration for each resource across all frames has at least one more than the minimum amount active and at least one less than the maximum amount active. This is relatively easy to find since a set with five of the six thresholds at their minimum value, and the sixth at its maximum, meets this criteria ($iw1 = 255$, and the others are all 1). We then use the following approach to narrow the range for each threshold, again using just G728dec.

The goal of narrowing the range for each thresholds is to find the maximum (and minimum) threshold value such that the mean configuration for each resource is just below the maximum configuration (or just above the minimum). When narrowing the range for a given threshold, we hold all other thresholds at their values from the starting point. We then perform two binary searches: one to find a reasonable maximum value and one for a reasonable minimum value. (Since the starting point has all thresholds at one extreme or the other of the possible reange, we actually only perform one binary search for each threshold.) We consider mean configurations that meet the following criteria to be far enough from having resources fully activated or deactivated: 126 $\geq$ instruction window size $\geq$ 18, 5.8 $\geq$ # of ALUs $\geq$ 1.2, and 3.8 $\geq$ # of FPUs $\geq$ 1.2. This process takes only 47 steps and 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. Table 4 gives the new range for each threshold.

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 (holding the others at their starting point values). Here, we try to find the best thresholds for each control algorithm independently, and then later 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.

Compared to tuning all thresholds together, this approach dramatically reduces the design space, but assumes independence of the algorithms. We choose 128 random points for each resource, and evaluate the four fastest applications (GSMdec, G728dec, G728enc, and H263dec). These applications are a mix of integer and floating-point applications, and have a range of sensitivities of performance and energy to the amount of each resource. Tuning with only a subset of applications more accurately models what would happen for a real system: tuning would be done for a test suite of applications, and other applications may behave poorly for the thresholds chosen. (2) Randomly choose all threshold values at the same time, and take the best set, as described above. This takes into account dependences between the algorithms, but gives a large design space for the search. We choose 128 random points, and evaluate each with the four fastest applications.

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%). These values are given in Table 4 as "Auto."

We also manually tune the thresholds. We start with thresholds used in a previous study (from [28]). Those thresholds were obtained through weeks of hand-tuning the LL algorithms [27]. Because of a difference in the adaptation interval length from the previous study, and because of small differences in the energy model, we re-tune the algorithms (together) to find a set of thresholds that gives good energy savings (the most we can find) while meeting our deadline miss criterion. We tune the LL algorithms for LL alone for all nine applications, and when we use them for GG+LL, they cause more than 5% deadline misses for one application, H263dec (7.4%). We evaluated all points adjacent in the design space (i.e., all combinations of changing each threshold by at most one), but none of them decreases the fraction of missed deadlines for H263dec. Due to the time intensive nature of manual tuning, we do not manually re-tune the algorithms for GG+LL. The values of the thresholds are given in Table 4 as "Manual."

5.2 LG Sensitivity to Profiling

For LG, with the pruning algorithm on average 77 architecture configurations are profiled (range is 50 to 126 across applications) out of 360 points in the design space. We also evaluated the LG algorithm with profiling only the same 54 configurations profiled for GG and GG+LL. The pruning algorithm increases savings over profiling the fixed set of 54 configurations by an average of 1% to 2% (depending on the deadline), and a maximum of 4%.

To test the sensitivity of LG to the profiling input (as opposed to the number of configurations profiled) we examined the energy savings for LG with four different profiling inputs for MP3dec. A previous study of variability in multimedia applications identified MP3dec as the application with largest IPC varability at the frame level of those studied here [16]. The range of energy savings across the four profiling inputs is narrow, 21.2% to 23.5%, indicating that the input used for profiling has only a small effect on the energy savings. Also, the profiling input with largest savings is not the one used for the online portion of the experiment. The previous study on variability

Application	Tight w/o DVS			Loose w/ DVS		
	GG+LL (Auto)	GG+LL (Manual)	LG	GG+LL (Auto)	GG+LL (Manual)	GG+LG
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 5. Missed deadlines (%) for automatically and manually tuned GG+LL and for LG and GG+LG. For loose deadlines and no DVS, $<$ 1% deadlines are missed for all applications and algorithms.

Savings from	Relative to Consumption of	Loose w/o DVS		Tight w/o DVS		Loose w/ DVS	
		Mean	Max	Mean	Max	Mean	Max
GG+LL (Auto)	Base	39	46	30	45	19	34
GG+LL (Manual)	Base	40	47	30	46	22	32
GG+LL (Manual)	GG	5	8	11	18	7	23
GG+LL (Manual)	LL	19	25	6	19	0	4
LG/GG+LG	Base	39	46	30	44	23	33
LG/GG+LG	GG+LL (Auto)	0	3	-1	5	4	24
LG/GG+LG	GG+LL (Manual)	-1	4	0	3	1	5

Table 6. Average relative energy savings (%) for different algorithm pairs, deadlines, and DVS support.

in these applications similarly found little difference across inputs in the IPC variability at the frame level.

5.3 Missed Deadlines

Table 5 gives the fraction of deadlines missed for GG+LL (with both Auto and Manual tuning) and for LG (or GG+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 (and GG+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.

5.4 Energy Savings

Figure 5 shows the energy consumption of Base (the non-adaptive processor), GG+LL with Auto tuning, GG+LL with Manual tuning, and LG (or GG+LG), all normalized to Base for each application. Table 6 gives the relative energy consumption for several pairs of processors, including some with GG or LL alone (detailed

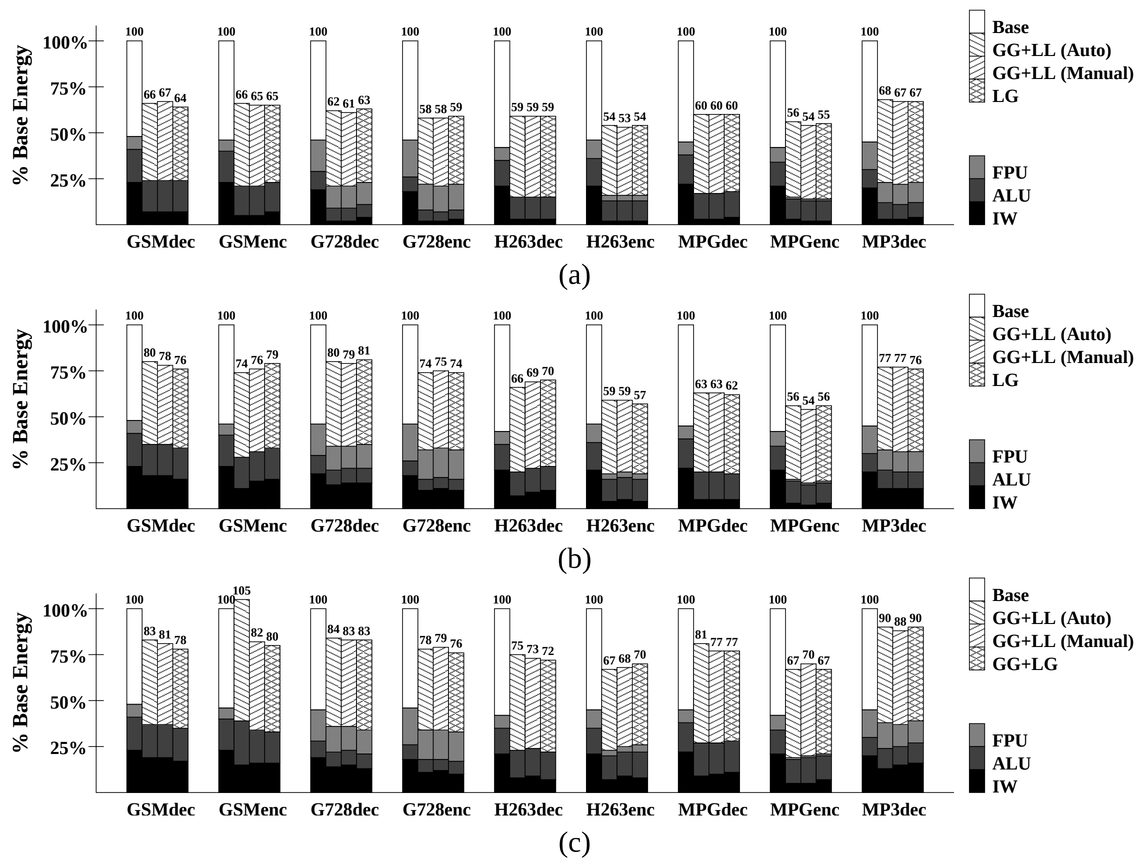

Figure 5. Energy consumption, normalized to Base. (a) With loose deadlines and no DVS. (b) With tight deadlines and no DVS. (c) With loose deadlines and DVS. The shaded parts of each bar indicate the energy consumed by the instruction window, ALUs, and FPUs.

results not shown).

For all combinations of deadlines and DVS, GG+LL with Auto tuning, GG+LL with Manual tuning, and LG (or GG+LG) give similar energy savings for all applications, with one notable exception explained below. Recall, however, that GG+LL with Auto tuning misses too many deadlines for some applications, so these 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. To help verify that our manual tuning of GG+LL is reasonable, we compare GG+LL with Manual tuning to both GG alone and LL alone. Our results agree with those previously reported: GG+LL with Manual tuning does better than both GG alone and LL alone, with relative savings depending on the amount of slack [28].

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 for GSMenc, GG+LL with Auto tuning chooses a much higher frequency for than GG+LG (645 MHz mean across all frames for GG+LL vs. 502MHz for GG+LG).

5.5 Summary and Discussion

GG+LL is difficult to practically implement due to the tuning effort required. We attempt to automate the tuning process, but it fails to find a set of thresholds that meets our maximum deadline miss requirement for even the applications with which it tunes. The set of thresholds chosen through the automated process also leads to almost 100% missed deadlines for an application outside the tuning suite, highlighting the unpredictability of such heuristics-based algorithms. Ignoring deadline misses, the energy savings using these thresholds is similar to LG (or GG+LG) in most cases, but in one case the energy consumption is worse than that for Base.

Since we take a more formal approach, our LG and GG+LG algorithms do not require the enormous tuning effort of GG+LL. Still, they give almost the same savings as GG+LL, the previous best adaptation control algorithm for real-time multimedia applications, even when GG+LL has been meticulously hand-tuned for best results with the applications studied. This is true across different applications and deadlines, and both without and with support for DVS. Although the results are not shown here, we found that LG can also accurately predict the energy savings it provides for a given amount of slack, allowing the operating system or user to decide whether the energy-performance tradeoff is favorable before running the application.

6 Related Work

Most of the adaptation control algorithms proposed in other studies have been for specific adaptive resources and have targeted either slack (for DVS) or variable resource utility (for architecture adaptation), but not both [2, 3, 4, 6, 9, 10, 12, 23, 24, 25, 26, 29]. We discuss work on controlling multiple adaptive resources below.

We have already discussed the algorithms we evaluate here [18, 28].

Huang et al. propose DEETM, a spatially global algorithm for general applications [14]. DEETM distributes slack evenly across all intervals (i.e., targets the same 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 (i.e., it does not support DVS) for general applications. It adapts at the temporal granularity of subroutines and can spread slack across different subroutines [15]. The algorithm associates a configuration with each subroutine that composes a large enough fraction of the application execution time, analagous to what we do for each PC value (but the effective interval size here is very large). It assumes that the adaptive resources are independent, so when determining which configuration to associate with each subroutine, it considers each resource 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. The algorithm differs from our LG algorithm in two important ways. First, it runs at a coarser temporal granularity, effectively running with a very large interval size. Second, it assumes that adaptations are independent of one another. These two simplifications of the problem reduce the search space enough that an exhaustive search (through all subroutine-adaptation pairs) becomes feasible to find the optimal configuration for each subroutine. 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. Even using an exhaustive search for our application with the smallest profiled frame (only 54 intervals) gives a search space of more than 10^{138} points. We avoid using exhaustive search by exploiting insights into multimedia applications to apply the solution provided by an efficient optimization algorithm. Furthermore, we integrate control of DVS into our algorithm.

Magklis et al. propose an algorithm for scheduling the voltage/frequency of each domain on a multiple clock domain processor (i.e., it does not control architecture adaptation) [22]. This algorithm runs at the subroutine level, and has both an off-line and on-line phase. The schedule is produced off-line (using a non-optimal "shaker" algorithm) through the use of profiling information and then applied on-line.

Dropsho et al. propose Local algorithms for the issue queues, load-store queue, ROB, register file, and caches [9]. They target general applications, and do not exploit slack. The algorithms for the caches use control feedback to obviate the need for tuning. However, they require accurate prediction and measurement of the 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. Therefore, the algorithms for the queues and ROB use heuristics based on thresholds.

7 Conclusions

In this paper, we study adaptation control algorithms for adaptive general-purpose processors running real-time multimedia applications. We take a more formal approach than previous studies by posing algorithm design as a constrained optimization problem. We want an algorithm that finds the optimal architecture configuration for each point in the program, given that at each point, each configuration has a different energy-performance tradeoff. We show how to apply a standard mathematical technique, the Lagrange multiplier method, to our problem. The solution, however, assumes knowledge of the energy-performance tradeoffs for the configurations at all points in each frame. Using properties of real-time multimedia applications, we develop a technique to estimate the tradeoffs, and use it to apply the solution. We call the resulting algorithm LG for its temporally local and spatially global approach to adaptation.

LG adapts frequently during a frame, making it hard to control high overhead adaptations (e.g., DVS). To solve this, we integrate LG with a previously proposed algorithm that adapts at frame boundaries, and use that algorithm to control only the high overhead adaptations.

We compare our algorithm's ability to make deadlines and save energy to the best previously proposed algorithm for real-time multimedia applications. Our algorithm finds the optimal solution for one frame and uses that solution to predict the best configuration for each part of other frames. On the other hand, the previous algorithm uses heuristics to choose a configuration for the next part of a frame based on processor behavior in the last part of the frame. These heuristics compare processor statistics to a set of thresholds, which require tuning to adjust the energy savings and missed deadlines.

While our algorithm meets the soft real-time requirements for our experiments, the previous algorithm's results depend on its tuning. When we rely on an automated tuning process, the previous algorithm is unable to always meet the requirements on missed deadlines, in one case missing nearly 100%. When we rely on hand-tuning the previous algorithm, 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 effort, saves almost the same energy as the previous one even when it is hand-tuned.

References

[1] S. V. Adve et al. The Illinois GRACE Project: Global Resource Adaptation through CoopEration. In *the Workshop on Self-Healing, Adaptive, and self-MANaged Systems (SHAMAN)*, 2002.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] T. M. Conte et al. Challenges to Combining General-Purpose and Multimedia Processors. *IEEE Computer*, December 1997.

[8] K. Diefendorff and P. K. Dubey. How Multimedia Workloads Will Change Processor Design. *IEEE Computer*, September 1997.

[9] 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.

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

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

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

[20] C. E. Kozyrakis and D. Patterson. A New Direction for Computer Architecture Research. *IEEE Computer*, November 1998.

[21] 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.

[22] G. Magklis et al. Profile-Based Dynamic Voltage and Frequency Scaling for a Multiple Clock Domain Microprocessor. In *Proc. of the 30th Annual Intl. Symp. on Comp. Architecture*, 2003.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] R. Sasanka, C. J. Hughes, and S. V. Adve. Private communication.

[28] 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.

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