COMBINING INTRA-FRAME WITH INTER-FRAME HARDWARE ADAPTATIONS TO SAVE ENERGY

BY

RUCHIRA SASANKA

B.Sc., University of Moratuwa, Sri Lanka, 1998

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Scienc
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2002

Urbana, Illinois

Abstract

We consider the use of dynamic voltage/frequency scaling (DVS) and architectural adaptation for saving energy for realtime multimedia applications. Previous work has considered this problem by invoking architectural and voltage/frequency adaptations at the granularity of a full frame of multimedia applications. The proposed *inter-frame* adaptation algorithm exploited computation slack in each frame by using the lowest energy configuration that still meets the application deadline.

We make two sets of contributions: (1) We explore architectural adaptation *within* a frame, or *intra-frame* adaptations, using both old algorithms proposed in the context of non-real-time applications and new algorithms. While inter-frame architectural adaptations can slow the computation, the intra-frame algorithms attempt to save energy without affecting performance. (2) We show how intra-frame and inter-frame architectural adaptations can be combined, and compare pure inter-frame, pure intra-frame, and the combined approaches.

We find that intra-frame architectural adaptation is effective for realtime multimedia applications. However, the combined inter-frame and intra-frame architectural adaptation approach saves more energy than either technique alone for systems both without and with support for inter-frame controlled DVS. Without DVS, inter-frame architectural adaptation provides most of the benefits, but with less slack intra-frame is more beneficial. With DVS, intra-frame architectural adaptation provides most of the benefits. However, it is not best for all applications, and, furthermore, the overhead for implementing inter-frame architectural adaptation in a system with inter-frame DVS support is small. Therefore, for systems both without and with DVS, our results indicate that the combined architectural adaptation algorithm is a good design choice.

To my Father, Mother and Wife

Acknowledgements

This work is made possible by the contribution of many people, including my project partner Christopher J. Hughes who helped me obtain the results and my adviser Sarita V. Adve who guided me through the whole process. Thanks to all the past members of the RSIM group who developed the RSIM simulator and customized the applications which were used in this study.

Table of Contents

1 Introduction

Multimedia applications have become an important workload for a variety of systems, and general-purpose processors are being increasingly employed for these systems [3, 6, 8, 18, 19]. A large number of these systems are powered by batteries, making energy a first class resource constraint. To save energy, researchers have proposed the use of hardware adaptation, including dynamic voltage and frequency scaling (or DVS) [12, 7, 13, 20, 27, 25, 26, 23] and architectural adaptation (e.g., through speculation control [21], changing the instruction window size [10], changing the number of functional units and/or issue width [22, 2], switching issue strategy between in-order and out-of-order [11], changing operand widths [4], and shutting off parts of the cache [1]). There is, however, little work integrating DVS and architectural adaptation ([14, 16]), and little work on architectural adaptation that targets the special characteristics of multimedia applications, such as the real-time nature of many such applications ([16]).

To our knowledge, the most comprehensive work so far that integrates both DVS and architectural adaptation and targets multimedia applications is by Hughes et al. [16]. Two key questions must be addressed when designing adaptive hardware – when to adapt and what to adapt. The work by Hughes et al. uses the following observations to address these questions. Many multimedia applications are real-time and need to process discrete units of data, typically called a frame, within a deadline. If the processor completes a frame's computation early, it remains idle until the end of the deadline. This idle time, or slack, implies that the processor can be slowed to reduce energy without affecting user-perceived performance. Further, previous work showed significant variability in the execution time of different frames of some of the applications, motivating different amounts of slowdown for different frames [15].

Based on the above observations, Hughes et al. developed an adaptation control algorithm that addresses the questions of when to adapt and what to adapt as follows. For the first question, they chose to invoke adaptations at the start of each frame, to exploit the int*er*-frame execution time variability and slack. For the question of what to adapt, before the start of the next frame, the algorithm predicts the lowest energy hardware configuration (voltage/frequency and architecture) that can meet the deadline for the frame, and uses that configuration to execute the frame. A limitation of the algorithm by Hughes et al. is that it runs the entire frame with the same hardware configuration and does not exploit any execution variability within a frame.

This paper builds on the work by Hughes et al. to additionally exploit execution variability within a frame, or int*ra*-frame variability. As we will see later, it is hard to exploit execution time slack purely at the intra-frame level since it is hard to predict the performance impact of adaptations at this lower granularity.

We therefore use intra-frame adaptations in a way that will not affect execution time. To continue to exploit slack, we propose a combination of the inter-frame and intra-frame approaches in a complementary fashion.

Specifically, we make two sets of contributions. First, we study intra-frame architectural adaptation algorithms for multimedia applications, both with and without DVS. The basic intra-frame approach – adapting to save energy without significant reduction in performance – is similar to that previously proposed for work on non-real-time applications. Therefore, much of that work is applicable here as well [21, 10, 22, 2, 11, 4, 1]. The focus of our work is on the intra-frame adaptation control algorithms and their interaction with inter-frame architectural adaptation and DVS. We therefore choose to focus on two example architectural adaptations for this work – varying instruction window size and varying the number of active functional units (and the associated issue width of the processor). We study the best existing (intra-frame level) algorithms for these adaptations for multimedia applications and develop a new algorithm for each adaptation. We find that, for our system and suite of multimedia applications, all algorithms show significant energy savings without much reduction in execution time, both with and without DVS. The new algorithms are slightly better than the older ones.

For our second set of contributions, we develop an integrated approach that combines architectural adaptation at both intra-frame and inter-frame granularity and inter-frame DVS. We do not consider DVS at the intra-frame level due to its high overhead and impact on performance. We report results with and without DVS for (1) the new integrated algorithm, (2) purely inter-frame architectural adaptation (as in [16]), and (3) purely intra-frame architectural adaptation. We find that for our system and multimedia application suite, the new integrated algorithm performs the best in all cases. Without DVS, most of the benefits, however, are provided by inter-frame architectural adaptation due to its ability to exploit slack. With less slack (i.e., with tighter deadlines) intra-frame adaptation is more beneficial. With DVS, most of the benefits are provided by intra-frame adaptation because DVS removes most of the slack. However, inter-frame architectural adaptation is better for some applications, and worth implementing because of its low additional overhead. Therefore, for systems both without and with DVS, our results show that a combination of inter-frame and intra-frame architectural adaptation is a good design choice.

2 Background on the Inter-Frame Algorithm

The inter-frame adaptation control algorithm developed by Hughes et al. invokes adaptations at the granularity of a frame. At the beginning of each frame, the algorithm predicts the hardware configuration (i.e., the architecture and voltage/frequency) that will minimize energy consumption for that frame without missing the deadline. Two versions of the algorithm are proposed, depending on whether the system supports voltage/frequency scaling in discrete or (almost) continuous steps. Since this distinction is not important for this paper, we chose to study the more general continuous DVS system here. Our work is equally applicable to a discrete DVS system. The key aspects of the inter-frame algorithm based on continuous DVS are summarized in Figure 1 (taken from [16]) and described in more detail below.

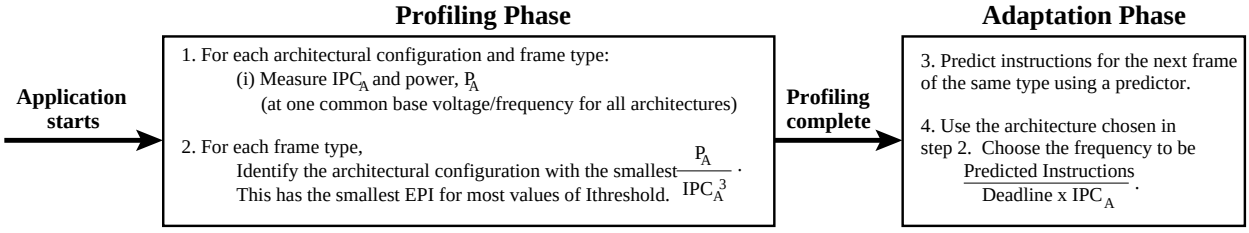

Figure 1 The algorithm proposed by Hughes et al. for choosing hardware configurations for a system with continuous DVS.

The inter-frame algorithm consists of two phases: a profiling phase at the start of the application followed by an adaptation phase. The profiling phase uses a fixed frequency/voltage throughout and profiles one frame of each type[1] for each possible architecture configuration. When profiling a frame for architecture A, the algorithm collects the average instructions per cycle (IPC_A) and average power (P_A) for the frame.

The results in [15] show that for a given application frame type and a given voltage/frequency, the average IPC and average power for an architecture are roughly constant for all frames of that type. Further, this IPC is roughly independent of the frequency/voltage. Thus, the values of IPC_A and P_A from the profile phase can be used to predict the IPC and average power of all other frames and hardware configurations. Using these results, Hughes et al. derive that for most cases in a continuous DVS system, a frame with a certain number of instructions I will execute within the deadline D and with approximately the lowest energy if it uses (1) an architecture A with the least value of $\frac{P_A}{IPC_A{}^3}$ and (2) a frequency of $\frac{I}{D \times IPC_A}$. Two exceptions to the above occur when the frequency calculated is the lowest or the highest frequency supported by the system. (We refer the reader to [16] to see how these exceptions are handled.)

[1] Some applications have multiple frame types (e.g., I, P, and B frames for MPEG-2 codecs). In such cases, the algorithm profi les and adapts for each frame type separately.

Based on the above, after profiling is complete, the algorithm computes $\frac{P}{IPC^3}$ for each architecture configuration and frame type. It chooses the architecture with the smallest such value to execute all frames of that type. Determining the execution frequency for a frame requires knowing the number of instructions in the frame. This is determined in the adaptation phase which follows next.

During the adaptation phase, before the start of the next frame, the algorithm first predicts the number of instructions in that frame using a simple history-based predictor [16]. Then the algorithm uses the expression given above to determine the frequency for execution. Since frame IPCs are only roughly constant, for a better approximation, instead of the profiled IPC_A, the algorithm uses the actual IPC of the previous frame of the same type and adds a small leeway,

3 Intra-Frame Algorithms

The inter-frame algorithm described in Section 2 takes advantage of the fact that the application can be slowed down, as long as it meets its real time guarantees. Prediction and measurement of the performance impact of adaptations at the inter-frame level is relatively straightforward. For adaptations at the *intra-frame* level, however, such prediction and measurement is difficult. Therefore, at this granularity we choose to examine adaptation algorithms that, ideally, do not affect execution time.

The motivation for intra-frame adaptations is that not all resources contribute to performance all the time. Thus, we can power down, or run at a "reduced level," under-utilized resources with little or no performance impact, but with significant energy savings. When we detect that a powered down resource is needed again, we must power it back up to avoid reducing performance. This type of adaptation is particularly well-suited for applications with resource requirements that change throughout the processing of a frame. Since resource requirements can change quickly, we are restricted to adaptations with little switching time (e.g., powering up a functional unit takes a few cycles).

In this paper, we examine two architectural adaptations: (1) changing the active instruction window size (Section 3.1), and (2) changing the number of active functional units, which also changes the issue width (Section 3.2). Although one might consider adapting voltage/frequency at the intra-frame level, the switching time is too large (on the order of microseconds), so we do not study this.

3.1 Intra-Frame Algorithm for Instruction Window Size Adaptation Control

Instruction window size adaptation assumes a design where the instruction window is broken into several (generally) equal segments, and where an arbitrary contiguous set of segments can be powered down at any cycle. We refer the reader to [10] for a detailed design of such an instruction window. The focus of this paper is on the control algorithm that determines the size of the active window at each cycle (and its interaction with the inter-frame algorithm). Recently, two such algorithms have been proposed [10, 9]. Both algorithms take a common approach of using the current state of the instruction window as a starting point, and then deciding whether to increase the size of the instruction window by one segment, reduce it by one segment, or leave the size unchanged. The decisions regarding decreasing and increasing the size are made independently, and are summarized in the top and bottom half of Table 1, respectively. The table also includes our new algorithm for increasing the window size. In principle, any of the algorithms for increasing the size can work with any of the algorithms for decreasing the size. The following discusses the different algorithms in detail.

Criterion for change	Advantages	Disadvantages
Decreasing the instruction window size		
Number of issues from the youngest segment is low (Folegnani et al. [10])	Can power down segments that do not contribute to IPC. Low overhead (1 bit per window entry).	Will keep the youngest segment powered up when non-critical instructions are issued from it.
Occupancy of the window is low (Ponomarev et al. [9])	Low overhead	This method is subsumed by the above method. Shuts down only when fetch mechanism fails to fill the window.
Increasing the instruction window size		
Periodic (Folegnani et al. [10])	Simple and low overhead	Can degrade energy since increase may be unnecessary. Can degrade IPC since increase may be too late.
Window is full (Ponomarev et al. [9])	Simple and low overhead	Can power up unnecessarily even if instructions are not issued from youngest segment
Reduced window size causes processor stalls	Increases only when IPC will benefit from increase.	Higher overhead – tag for each window entry.

Table 1 Strategies for adaptation of the size of the instruction window.

3.1.1 Algorithm by Folegnani et al.

The state-of-the-art algorithm, by Folegnani et al., decreases the size of the instruction window if the number of committed instructions that issued from the youngest segment during a fixed period is smaller than a threshold [10]. This has the advantage of powering down segments of the window that clearly do not contribute to the overall IPC, and has relatively low overhead (primarily 1 bit per instruction window entry). A disadvantage is that the youngest segment stays powered up even if the instructions that are issued are not on the critical path of the program; i.e., do not contribute to IPC.

The algorithm for increasing the size of the instruction window uses a simple, periodic strategy – the window is increased by a segment every fixed number of cycles. This algorithm is somewhat ad hoc since no attempt is made to determine if the extra instructions that will fit into a larger window will be useful (i.e., increase IPC). This could potentially waste energy. Conversely, for some cases, this scheme may be too conservative and lead to a high IPC degradation, since it may not react early enough to increased demands of an application.

3.1.2 Algorithm by Ponomarev et al.

The algorithm by Ponomarev et al. decreases the instruction window size if the youngest segment is unoccupied and increases it if the window fills up [9]. The strategy for decreasing is subsumed by the previous algorithm by Folegnani et al. Much like the previous algorithm, the strategy for increasing the window size

does not consider the utility of the instructions that will fill it.

3.1.3 A New Algorithm for Increasing Instruction Window Size

We propose a new algorithm, summarized in Figure 2, for increasing the size of the instruction window, based on an estimate of the resulting benefit in IPC. To obtain this estimate, we estimate the number of retirement stalls (henceforth referred to simply as *stalls*) that could have been avoided with a larger instruction window. An instruction I stalls if it reaches the head of the instruction window before completion. A larger instruction window can potentially avoid such a stall by providing more instructions ahead of I to overlap with I's latency. The key to our algorithm, therefore, is a technique to estimate this extra overlap that an instruction would have if the instruction window were fully powered up. Several aspects of the detailed design required making a tradeoff between improved accuracy or reduced hardware and energy overhead. We describe the design we chose next; other variations that improve accuracy or reduce overhead further are possible.

The algorithm keeps track of the average IPC over a fixed interval (400 cycles in our experiments). When an instruction is fetched into the youngest segment of the instruction window, it checks to see if its operands are already available. If so, a larger instruction window could potentially have allowed for more overlap for that instruction, as illustrated in Figure 3. We optimistically estimate that the additional potential overlap could have been $\frac{Number\ of\ powered\ down\ entries}{IPC}$ cycles,[2] and update a tag for this instruction, called *IWtag*, with this value. This update for IWtag is optimistic (but low overhead) because it assumes that even for the case of the larger instruction window, this instruction's operands would be available on fetch and the instruction would enter the youngest window segment. This estimate also ignores structural hazards on functional units.

A larger window could also benefit instructions that do not have their operands available on entry, if the large window enabled the producers of the pending operands to generate their results earlier. The earlier operand generation would provide the consumer a larger number of instructions to overlap its latency. The increased overlap would be the same as the increased overlap for the producer of the last operand, if the producer itself did not stall. If the producer stalls, then the overlap available for the consumer is reduced by the producer's stall cycles (i.e., by the cycles that the producer will use up for overlapping its own latency). Thus, when an instruction completes, we pass its IWtag to all instructions for which this instruction produced the last operand; if this instruction stalled before completion, then we reduce its IWtag by

[2] This value can be calculated in an approximate way, and is calculated only once every 400 cycles or when the instruction window is resized in our experiments.

On entry into the instruction window

```
for each instruction i inserted
    if( the operands of i are ready )
        i.tag = Current-Tag-Value;
```

Completion Stage

```
for each instruction i completed
    for each instruction j which consumed the result of i
        if( i produced the last arriving operand of j )
            if( i did not stall the processor )
                if( j.tag < i.tag )
                    j.tag = i.tag
            else if( i stalled the processor for c cycles )
                if( j.tag < i.tag - c )
                    j.tag = i.tag - c
```

Retirement Stage

```
for each instruction i retired
    if( i.tag > 0 )
        if( i stalled for k cycles )
            Stall-Counter += min( k, i.tag )
if( Stall-Counter > Stall-Thresh )
    increase the window size by 1 segment
    Stall-Counter = 0
```

At the end of each interval

```
Current-Tag-Value = Number-Of-Inactive-Segments /
                    Current-Average-IPC
Stall-Counter = 0
```

Figure 2 A new algorithm for increasing the instruction window size.

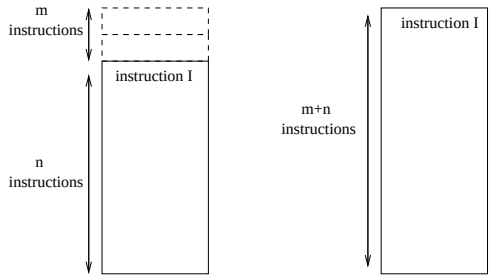

Figure 3 The exection ovelap lost due to the reduced window size. The left part shows an instruction window with n entries powered up and m entries powered down. Instruction I arrives at the top of the powered up portion of the window and finds all its operands available. Thus, I is overlapped with n instructions. If the instruction window were fully powered up (the right part) and I arrived at the top, it could have an additional m instructions for overlap.

$min(IWtag,\ stall\ cycles)$ before passing it to the consumers. Again, this is possibly an over-estimate of the possible overlap for the consumer because the producers of the other operands of the consumer may not be able to provide that much overlap with a larger instruction window.

The value of IWtag so far gives the additional overlap that an instruction could get with a larger instruction window. To determine how many stalls could be reduced from the additional overlap, we check the IWtag of each instruction that stalls the processor. If non-zero, we estimate that $min(IWtag, stall\ cycles)$ stalls could be avoided. We accumulate the avoidable stall cycles in a counter. When the counter exceeds a threshold, we increase the instruction window size by one segment and reset the counter. The counter is also reset periodically.

Our new algorithm potentially alleviates the disadvantage of the previous algorithms for increasing the

instruction window size because (1) it increases the instruction window size only when it estimates that the IPC will benefit, making it less wasteful of energy, and (2) it increases the window size as soon as it is possible for the IPC to benefit, limiting any IPC degradation from the adaptive hardware. In comparison, the previous algorithms are more ad hoc.

The disadvantage of the algorithm is in the higher hardware overhead. The primary overhead is in the bits for holding IWtag; however, we found that a small tag size (4 bits for our case) suffices.

3.1.4 Discussion and Choice of Algorithm

As mentioned earlier, it is possible to combine any of the algorithms that increase the instruction window size with those that decrease it. In this paper, we report results for the combination studied by Folegnani et al. as representing the state-of-the-art, and call this *PeriodicIW*. We also report results for the new algorithm combined with the algorithm by Folegnani et al. for decreasing the window size (since this subsumes the other algorithm for decreasing the window size), and call this *CriticalIW*. When combining the two algorithms, one of them must be given priority since both may simultaneously indicate that the instruction window size should change – and in opposite directions. We give priority to reducing the instruction window in the following manner. During a single period, we prevent an increase in the size of the window until we are certain that the algorithm for decreasing the size will not decide to decrease it for this period.

We also studied the algorithm by Ponomarev et al. for increasing the window size, but found that it always keeps significantly more of the instruction window powered up than the algorithm by Folegnani et al. on our system and application suite. We therefore do not report results for this algorithm.

It is also possible to design an algorithm for decreasing the instruction window size based on the new technique to increase the instruction window size. In particular, we could choose to reduce the size if the "avoidable stalls" counter for the new algorithm was below a certain threshold. The intuition is that the above condition indicates an increased window size would not improve IPC by much; therefore, it may be the case that an instruction window size reduced by one segment would also not degrade IPC by much. We experimented with this algorithm, but found that it did not perform as well as using the one by Folegnani et al. for decreasing the window size. A smarter algorithm would need to directly determine when instructions issued from the youngest segment of the instruction window are not critical instructions.

3.1.5 Parameters for the Algorithms

A key issue for intra-frame adaptation algorithms is that they use a number of different parameters which affect both energy savings and IPC degradation. A design space search must be performed to find the best

overall parameters for an algorithm. In many cases (all algorithms that we examined), the time required for an exhaustive search is prohibitive. For each algorithm, we evaluated several sets of parameters and found that energy savings are not as sensitive to the parameters as the IPC degradation (likely due to the energy savings being relatively small in most cases). Therefore, in our experiments we use parameters that were near the knee of the energy savings curve with the limitation that IPC degradation not be too large. For all the applications and systems, IPC degradation is less than 8%, and the average degradations for any single system are below 5%. We now discuss the parameters used for the two instruction window size adaptation algorithms for which we report results.

PeriodicIW considers reducing the instruction window size every 200 cycles (the period), and reduces the size by one segment if no instructions were issued from the youngest segment in the last period (i.e., this is the minimum value of the threshold). Larger threshold values give similar energy savings, but significantly increase IPC degradations. *PeriodicIW* increases the instruction window size by one segment every five periods (starting from the last time it either increased or decreased the size).

For comparison, Folegnani et al. chose 1000 cycles as the period for decreasing the instruction window size and also chose 5 periods for increasing the window size. We chose a smaller period because we found that the adaptation overhead was small, and choosing a smaller period allowed faster response to changing requirements of the application.

CriticalIW also considers reducing the instruction window size every 200 cycles. It reduces the instruction window size by a segment if the processor issued less than 40 instructions from the youngest segment in the last period. This threshold is much more aggressive than the one used for *PeriodicIW* because *CriticalIW* can more rapidly increase the instruction window size when needed. *CriticalIW* increases the instruction window size only when the number of stall cycles avoidable by the largest instruction window reaches 20 in a period (and at least 40 instructions have been issued from the youngest segment). *CriticalIW* uses 4 bit IWtag values.

3.2 Intra-Frame Algorithm for Functional Units and Issue Width Adaptation Control

Recently, several algorithms have been proposed to change the number of active (powered up) functional units and the consequent instruction issue width [22, 2], as summarized in Table 2. As in the case of instruction window adaptation, the algorithms for decreasing (top half of Table 2) and increasing (bottom half of Table 2) the number of functional units and issue width can be independent. We discuss the state-of-the-art below (Section 3.2.1) and then discuss new algorithms and combinations we tried (Section 3.2.2).

Criterion for change	Advantages	Disadvantages
Decreasing the number of active units		
Low utilization (Maro et al. [22])	Simple to implement	Does not power down if a unit is only utilized by non-critical instructions.
Low issue IPC (Bahar and Manne [2])	Simple to implement	Does not power down if a unit is only utilized by non-critical instructions. Many thresholds.
Increasing the number of active units		
High utilization (Maro et al. [22])	Simple to implement	High utilization of remaining units does not necessarily mean more units will be used. May increase for non-critical instructions.
High structural hazards for unit [22]	Can increase quickly when the hazard count is high	May increase for non-critical instructions
High issue IPC (Bahar and Manne [2])	Simple to implement	High issue IPC does not necessarily mean more units will be used. May increase for non-critical instructions.
Reduced number of units causes processor stalls	Increases only when IPC will benefit from increase	High overhead – tag for each instruction window entry

Table 2 Strategies for adaptation of the number of active functional units (and issue width).

3.2.1 State-of-the-art

Maro et al. proposed several algorithms for controlling the number of powered up functional units in a clustered architecture with two clusters, similar to the Alpha 21264 [22]. Their algorithms choose whether to have one or two clusters powered up, but they could be extended to provide finer control over the number of active functional units. The best algorithm proposed used functional unit utilization to determine whether to increase or decrease the number of active units. An advantage of this algorithm is that it is very simple to implement. The algorithm has three disadvantages: it does not power down a unit even if the instructions using it are not on the critical path, it may power up a unit that will not be used, and it may power up a unit even if the instructions that will use it are not on the critical path.

Maro et al. consider two other strategies for decreasing the number of active functional units: power down on low committed IPC, and power down when there are too many instructions waiting on data dependences in the instruction window. They found that neither of these performs as well as the utilization-based algorithm, so we do not explore them further.

Maro et al. also discuss, but do not evaluate, one other strategy for increasing the number of active functional units (also for a clustered architecture). The processor would track the number of structural hazards for each instruction (this would require a counter for each instruction window entry). At a given point in time, if enough hazards have been encountered by instructions in the instruction window, the algorithm powers up a cluster. This could be extended for finer control over the number of active functional units.

Bahar and Manne also studied a clustered architecture with two clusters, and developed an algorithm

to power down nothing, a full cluster, or half the ALUs and all of the FPUs in one cluster [2]. This could be extended to provide finer control. Their algorithm for powering up and powering down is based on the number of instructions issued per cycle (issue IPC) to each type of functional unit (ALU or FPU) [2]. After a certain period, the issue IPC is compared to both an upper and a lower threshold (the thresholds used depend on the current number of active units). If the upper threshold is exceeded, the number of active units is increased by one. If the issue IPC is below the lower threshold, the number of active units is decreased by one. Issue IPC, as a criterion for controlling functional unit adaptation, is very similar to utilization, having the same advantages and disadvantages. However, using issue IPC has one additional disadvantage: since the thresholds depend on the current number of units, a set of thresholds is required for each possible number of active units (a total of 8 thresholds in [2]. Choosing the right combination of thresholds to give this scheme the best showing would have required an inordinately large number of simulations. We therefore choose the utilization-based scheme (which is close to the above) as the state-of-the-art, and call it *UtilFU*.

3.2.2 New Algorithms and Combinations For Adapting Number of Active Units

A new algorithm for increasing the number of active functional units could be based on an estimate of the resulting benefit in IPC. Such an algorithm would be analogous to the new algorithm for increasing the size of the instruction window based estimated IPC benefit. We explored this option and found that the larger overhead for this algorithm, compared to the others, made this algorithm perform worse than most others.

We propose, and report results for, an algorithm that combines a utilization-based scheme for decreasing the number of active functional units with a scheme that tracks structural hazards to decide when to increase the number of active units. We call this scheme *HazardFU*. The algorithm for increasing the number of active units is very similar to one of those proposed by Maro et al. An extension of that algorithm for a non-clustered architecture would track the number of structural hazards from each type of unit encountered by instructions currently in the instruction window. We simplify this by using counters to track the number of structural hazards caused by each unit type over a time interval. If the counter exceeds a threshold before the end of a period, we increase the number of active units by one. The counter is reset at the end of the period.

In this paper, we also report results for an algorithm using utilization-based schemes for both increasing and decreasing the number of active units, as studied by Maro et al. (we call this *UtilFU*), as representing the state-of-the-art.

3.2.3 Parameters for the Algorithms

Parameters for the two functional unit adaptation algorithms for which we report results were chosen in the same manner as for the instruction window size algorithms (Section 3.1.5).

For both *UtilFU* and *HazardFU*, when all FP units are powered down, if an FP instruction is fetched, both algorithms immediately power up an FP unit. Note that instructions get issued to functional units in a prioritized manner; therefore, "the last unit" of each type will only be used when all other units are busy. This affects the unit utilizations.

UtilFU follows the LP1 scheme proposed in [22] for deciding when to increase the number of active functional units. If the last active unit of a given type has a utilization of at least 86% for the last period, *UtilFU* increases the number of active units of that type by one. *UtilFU* reduces the number of active units by one if the last unit is not used more than 4 cycles within the last period. However, *UtilFU* uses a different criteria for powering down the last FP unit when only one is active, as proposed in [22]. It powers down the last FP unit when that unit is not used for three cycles in a row. While [22] proposes using a small period – 16 cycles – we found that for our applications and system, this performed significantly worse than a larger period, regardless of thresholds. Therefore, we use a 200 cycle period.

HazardFU increases the number of active functional units for a given type by one if, during the last period, instructions faced at least 80 structural hazards from that type of unit. *HazardFU* reduces the number of active units by one when a unit of that type is not used more than 4 cycles within the last period (the same criteria *UtilFU* uses). *HazardFU* powers down the last available FP unit if it is not used at all within the last period. We use a 200 cycle period for *HazardFU* as well.

4 Integrating Inter-Frame and Intra-Frame Adaptation Control Algorithms

We combine the inter-frame and intra-frame adaptation control algorithms in a simple way, resulting in two parallel but integrated control loops in the system. Overall, the inter-frame algorithm enables exploiting per-frame slack while the intra-frame algorithm enables exploiting the execution variability within a frame.

- The inter-frame part of the integrated algorithm performs the profiling and adaptation phases as before, but with the following change. When profiling a specific candidate architecture for the inter-frame algorithm, we also invoke the intra-frame algorithms on this candidate architecture. Ideally, the inter-frame algorithm will now see lower average power numbers for each candidate architecture, with possibly little change in IPC, as compared to the case without the intra-frame adaptations. The profiling phase gives the IPC and power values for each candidate architecture with intra-frame adaptation turned on. As before, the inter-frame algorithm now picks the lowest energy architecture as the one with the lowest $\frac{P}{IPC^3}$.

  During the adaptation phase, again, the intra-frame algorithm is invoked on all frames. The inter-frame algorithm picks the voltage/frequency for executing the next frame as before, as a function of the deadline, the predicted instruction count for the frame, and the measured IPC of the last frame of the same type. The inter-frame algorithm will automatically compensate for any IPC degradation caused by the intra-frame algorithm. The inter-frame algorithm thus avoids possible missed deadlines caused by the intra-frame algorithm, but sees reduced benefits from the intra-frame algorithm.

- The intra-frame algorithm is always applied, including during both the profiling and adaptation phases of the inter-frame algorithm. The inter-frame algorithm picks the maximum configuration for each resource for the intra-frame algorithm. The intra-frame algorithm is only allowed to exercise adaptations that will power down components with respect to the configuration chosen by the inter-frame algorithm. That is, the intra-frame algorithm cannot increase the aggressiveness of any resource beyond what the inter-frame algorithm chose (since the goal is to not change IPC in either direction). Thus, if the inter-frame algorithm chooses aggressive configurations (i.e., they can exploit high ILP), then the intra-frame algorithm has substantial potential for exercising adaptations. However, if the inter-frame algorithm picks configurations that are significantly less aggressive (i.e., already "powered down" in many places), then the intra-frame algorithm does not have much potential to further adapt.

The inter-frame and intra-frame algorithms play different roles. In particular, the inter-frame algorithm has a large adaptation granularity, which makes it better able to predict execution behavior. The inter-frame algorithm (1) relies largely on profile information mostly collected at the beginning of the execution or at major changes in the application or system behavior, (2) considers a global hardware configuration at a time, (3) considers adaptations that will increase execution time to reduce slack, and (4) is well-suited for high-overhead adaptations (e.g., changing clock frequency or cache size) but not for adaptations that can only be enabled for parts of the frame execution (e.g., shutting off all floating point units).

In contrast, the intra-frame algorithms operate at a much finer granularity and it is difficult to foresee the impact of adaptations at this granularity. Therefore, the intra-frame algorithms are (1) fully dynamic relying on continuous monitoring of the system, (2) consist of several local control algorithms for adaptations of individual hardware structures that are continuously monitored and adapted, (3) only invoke adaptations that will not affect performance, and (4) are well-suited for adaptations that can only be enabled for parts of the frame execution but not for high-overhead adaptations. These differences are summarized in Table 3.

Properties	**Inter-frame adaptation**	**Intra-frame adaptation**
Basis of adaptation	Profiles mostly collected at start of application	Information from continuous monitoring
Hardware features controlled	Global configuration	Individual (or small groups of) hardware features
Impact on execution time	May increase	No impact (ideally)
Adaptations for which it is best	Adaptations with high overhead	Adaptations that cannot be invoked for the entire frame

Table 3 Comparison between inter-frame and intra-frame adaptation algorithms.

5 Experimental Methodology

5.1 Architectures

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 (Unifi ed)	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**	
Memory queue size	32 entries	L1 (Data) hit time (on-chip)	2 cycles
Branch Prediction	2KB bimodal agree, 32 entry RAS	L2 hit time (off-chip)	20 cycles
		Main Memory (off-chip)	102 cycles

Table 4 Base (default) system parameters.

The base processor studied is similar to the MIPS R10000 and is summarized in Table 4. We assume a centralized instruction window with a separate physical register file. We also study a version of the base processor with support for continuous dynamic voltage/frequency scaling (DVS). The voltages used for each frequency are the same as in [16]. They were derived from the information available for Intel's XScale (StrongArm-2) processor [17]. We allow the frequency to range from 100MHz to 1GHz.

We also study processors capable of adapting their instruction window size and/or the number of active functional units and issue width. The instruction window is broken into segments of 8 entries each, and at least two segments must always be active. The only restriction on the number of functional units active is that at least one integer ALU must always be active. The issue width of the processor is equal to the sum of all active functional units and hence changes when we change the number of active functional units.

We use the intra-frame adaptation control algorithms as described in Section 3. The algorithm parameters used for each experiment are given in Section 6. We model the energy overhead from the extra bits required in the instruction window for instruction window size adaptation. We also model a delay of 5 cycles to power up an inactive functional unit or instruction window segment (we observed that the results are not sensitive to this parameter).

We use the inter-frame adaptation control algorithm proposed in [16]. We increase the IPC leeway from 1% to 4% in order to make all of the applications have fewer than 5% missed deadlines on the base processor with DVS. This algorithm is used to control DVS in all cases, even when no inter-frame architectural adaptation is performed. As in [16], we ignore time and energy overheads for invoking inter-frame adaptation since it is invoked so rarely (once every frame). A more detailed justification for this assumption appears

in [16].

For inter-frame adaptation with DVS, we profiled all possible combinations of the following configurations (54 total): instruction window size $\in \{16,32,48,64,96,128\}$, number of ALUs $\in \{6,4,2\}$, and number of FPUs $\in \{4,2,1\}$. Of these configurations, only eleven were chosen by the pure inter-frame algorithm in the adaptation phase across all applications. When running the inter-frame algorithm alone and the integrated inter- and intra-frame algorithm, to keep simulation time for the profiling phase down, we limited the configurations available to the inter-frame part of the algorithm to be the above eleven. A real system may also choose to limit the configurations it uses for profiling. Nevertheless, we will also run with all configurations for the final version; we do not expect the results to change.[3]

For inter-frame adaptation without DVS, we allowed the adaptation control algorithm to choose from the following 11 hardware configurations, in the form of (instruction window size, number of ALUs, number of FPUs): (128,6,4), (96,4,2), (96,4,1), (64,4,2), (64,4,1), (48,4,1), (48,2,1), (32,4,1), (32,2,1), and (16,2,1). These configurations were picked to represent a spectrum of the available space and keep simulation time down. Since our results showed that in this case, the inter-frame and the integrated algorithm always did better than the intra-frame algorithm, pursuing a fuller set of configurations would only make this result stronger. Nevertheless, again, we will run experiments with the full set of configurations for the final paper.

One difference in the effect of instruction window adaptation as invoked in the inter-frame compared to the intra-frame algorithms is that the former is also able to change the number of active physical registers with instruction window size. A smaller instruction window requires fewer physical registers. Therefore, with the inter-frame algorithm, the number of active physical registers, of each type, is equal to the number of logical registers (64 for each type) plus the number of entries in the instruction window. We do not allow intra-frame adaptation to control the register file size because it is not clear how to do so with our processor model (reducing the size requires "garbage collecting" register contents during the course of the execution of a frame). This would be possible on other processor models (e.g., one with a combined physical register file and reorder buffer). This is not a problem with the inter-frame algorithm because the adaptations are invoked before the start of a frame; i.e., before any state is accumulated in the registers.

App.	Type	Frame Period	Input Size		Deadline	IPC
			Time	Frames		
GSMdec	Speech	20ms	20s	1000	50μs	4.0
GSMenc	codec		20s	1000	140μs	4.8
G728dec	Speech	625μs	0.63s	1000	60μs	2.4
G728enc	codec		0.63s	1000	70μs	2.2
H263dec	Video	40ms	4s	100	2.9ms	3.5
H263enc	codec		4s	100	40ms	2.5
MPGdec	Video	33.3ms	3.33s	100	6.3ms	3.8
MPGenc	codec		3.33s	100	66.6ms	2.7
MP3dec	Audio	26.1ms	13.05s	500	1.4ms	3.1

Table 5 Workload description.

5.2 Workload Description

Table 5 summarizes the nine applications and inputs used in this paper. These were also used in [15, 16] and are described in more detail in [15] (for some applications, we use fewer frames). The deadlines were chosen using the criteria for the tighter deadlines from [16]. For all but the video encoders, the deadline is three times the maximum processing time for a frame on the base processor. For the video encoders, we use longer deadlines since even the base processor is not able to meet this deadline in some cases – for H263enc, we use the full frame period and for MPGenc, we use twice the original frame period.

5.3 Performance and Energy Evaluation

We use the RSIM simulator [24] for performance evaluation. All applications were compiled with the SPARC SC4.2 compiler with the following options: *–xO4 –xtarget=ultra1/170 –xarch=v8plus*.

We use the Wattch tool [5] integrated with RSIM for energy measurement. We assume clock gating for all the components of the processor with 10% of its maximum power charged to a component when it is not accessed in a given cycle. To fairly represent the state-of-the-art, we also gate the wake-up logic for empty entries and ready entries in the instruction window as proposed in [10]. The gating is done by all processors (adaptive and non-adaptive). We also assume that the resources that are powered down by our adaptive algorithms do not consume any power.

Since we adapt the issue width of the processor with functional unit adaptation, we power down the selection logic corresponding to the functional units that are powered down. Also, when a functional unit is

[3] It may seem that profiling 54 configurations may be inordinate overhead for a real system. However, it is feasible since only one frame of each type need be profiled for each configuration, the total number of frames in a typical multimedia application is much larger than the number we need to profile (e.g., 30 frames a second for video), and the profiling can be done as part of the application's execution.

powered down, the corresponding part of the result bus, the wake-up ports to the instruction window, and write ports to the register file are also powered down.

6 Results

Sections 6.1 and 6.2 evaluate the intra-frame algorithms for adapting the instruction window size and functional units in isolation. Section 6.3 compares the pure intra-frame, pure inter-frame, and integrated inter-frame and intra-frame algorithms.

6.1 Intra-Frame Adaptation of Instruction Window Size

For instruction window size adaptation we evaluate three processors: one without adaptation support (*Base*), *Base* enhanced with an adaptive instruction window size controlled by *PeriodicIW*, and *Base* enhanced with an adaptive instruction window size controlled by *CriticalIW*. Figures 4(a) and 4(b) show the total processor energy normalized to *Base* for each application, without and with DVS, respectively. Each bar also shows the part of the energy dissipated by the instruction window.

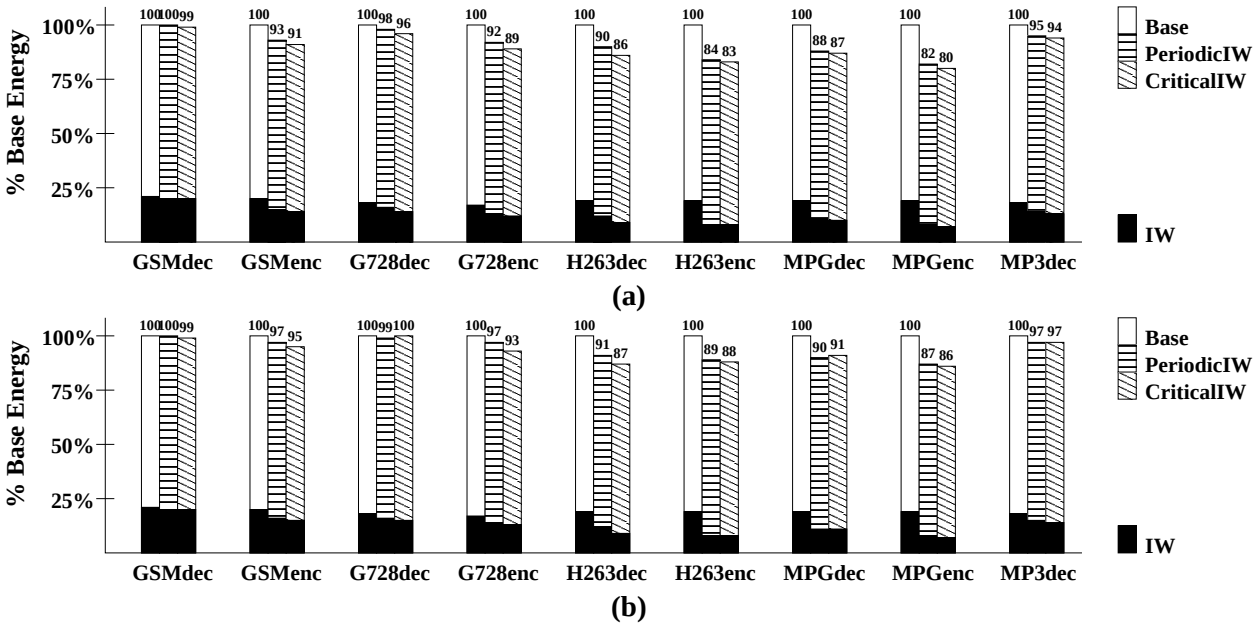

Figure 4 Energy consumption (normalized to *Base*) with instruction window size adaptation. (a) Without DVS. (b) With DVS. Each set of three bars represents *Base*, *PeriodicIW*, and *CriticalIW* respectively. The dark part in each bar shows the amount of energy spent in the instruction window.

Overall, we find that both instruction window adaptation algorithms are effective for multimedia applications, saving a significant amount of energy for some applications. As expected, the savings come primarily from the energy dissipated by the instruction window. *CriticalIW* saves more energy than *PeriodicIW*, but the difference is small.

Without DVS, *PeriodicIW* saves an average of 9% of *Base's* energy (maximum of 18%), while *CriticalIW* saves an average of 11% (maximum of 20%). With DVS, *PeriodicIW* saves 6% on average (maximum of 13%) over *Base*, and *CriticalIW* saves 7% on average (maximum of 14%). The processor with DVS increases the frequency to compensate for IPC degradations in order to meet the deadline, eroding some of the energy savings from adaptation.

Comparing *CriticalIW* and *PeriodicIW*, both without and with DVS, for all applications, *CriticalIW* saves as much or more energy than *PeriodicIW*. However, the difference is small (1% on average, 4% maximum). The mean IPC degradation for *PeriodicIW* and *CriticalIW*, without and with DVS is also similar – 3% and 4% respectively.

Table 6 shows the average size of the instruction window chosen by each algorithm. *CriticalIW* is able to reduce the size of the instruction window by as much or more than *PeriodicIW* for all applications. This difference is due to the more aggressive powering down of *CriticalIW*. Also, the periodic powering up strategy of *PeriodicIW* may hamper its ability to reduce the instruction window for as long as possible. Consequently, *CriticalIW* saves slightly more energy than *PeriodicIW*.

Application	No DVS		DVS	
	PeriodicIW	CriticalIW	PeriodicIW	CriticalIW
GSMdec	123	117	123	117
GSMenc	89	82	89	82
G728dec	113	99	110	98
G728enc	94	86	92	84
H263dec	76	55	76	54
H263enc	50	50	50	50
MPGdec	67	60	67	61
MPGenc	48	41	48	41
MP3dec	99	92	98	91

Table 6 Mean instruction window size selected by PeriodicIW and CriticalIW

6.2 Intra-Frame Adaptation for Number of Active Functional Units and Issue Width

For functional unit and issue width adaptation we evaluate three processors: *Base*, *Base* enhanced with adaptive functional units controlled by *UtilFU*, and *Base* enhanced with adaptive functional units controlled by *HazardFU*. Figure 5(a) and Figure 5(b) show the total processor energy for each application, normalized to *Base*, without and with DVS respectively. Each bar also shows the part of the energy dissipated due to the instruction window, ALUs, and FPUs. Energy savings for this type of adaptation come from many parts of the processor, as explained in Sections 3 and 5, because adapting the issue width allows powering down of parts of a number of structures.

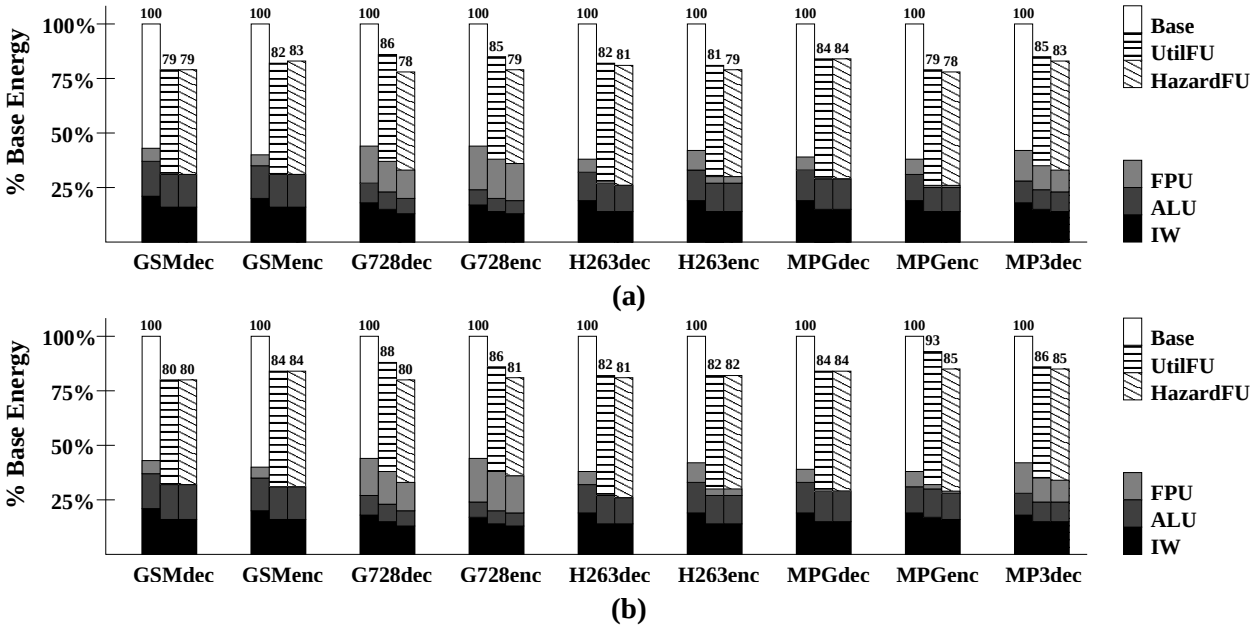

Figure 5 Energy consumption (normalized to *Base*) with functional unit and issue width adaptation. (a) Without DVS. (b) With DVS.

The results show that both algorithms are very effective for multimedia applications. There is negligible difference between the two algorithms in most cases. Exceptions are G728enc and G728dec, where *HazardFU* is superior.

More specifically, without DVS, *UtilFU* saves an average of 18% of the energy over *Base*, while *HazardFU* saves 20%. With DVS, *UtilFU* saves 15% on average over *Base*, and *HazardFU* saves 18%. For most cases, the difference between the two algorithms is negligible – on average, *HazardFU* saves 3% over *UtilFU* without DVS and 3% with DVS. For two applications (G728dec and G728enc), however, the difference is more significant. In these cases, *HazardFU* is superior and shows a maximum benefit of 9% over *UtilFU* (for G728dec without DVS). The IPC degradation for the two algorithms with or without DVS is 2% to 3% averaged across all applications.

Table 7 shows the average number of active functional units chosen by each algorithm. The number of active ALUs and FPUs are very similar for *UtilFU* and *HazardFU* for all applications except G728dec and G728enc, for which both the mean number of active ALUs and FPUs is smaller for *HazardFU*. These are the two applications for which *HazardFU* saves significantly more energy than *UtilFU*. *HazardFU* is able to keep more powered down because *UtilFU* activates an extra unit when the last unit is highly utilized, but the processor does not issue instructions to it (or there would be structural hazards and *HazardFU* would also power it up).

Application	No DVS				DVS			
	UtilFU		HazrdFU		UtilFU		HazardFU	
	ALU	FPU	ALU	FPU	ALU	FPU	ALU	FPU
GSMdec	4.5	0.0	4.5	0.0	4.5	0.0	4.5	0.0
GSMenc	4.9	0.0	5.0	0.0	4.9	0.0	5.0	0.0
G728dec	3.9	2.4	3.0	1.5	4.1	2.4	3.2	1.4
G728enc	3.5	2.7	2.5	2.0	3.4	2.7	2.5	2.0
H263dec	5.1	0.0	4.8	0.0	5.1	0.0	4.8	0.0
H263enc	4.4	0.6	4.1	0.5	4.4	0.6	4.1	0.5
MPGdec	5.4	0.0	5.3	0.0	5.4	0.0	5.3	0.0
MPGenc	3.5	0.2	3.8	0.1	3.5	0.2	3.8	0.1
MP3dec	4.3	1.7	4.0	1.2	4.3	1.7	4.0	1.2

Table 7 Number of functional units selected by UtilFU and HazardFU

6.3 Comparing Inter-Frame, Intra-Frame, and Integrated Inter+Intra Frame Algorithms

This section compares the inter-frame, intra-frame, and integrated inter+intra frame approaches for architectural adaptation, with and without DVS. For the experiments with intra-frame adaptation, we adapted both instruction window size and the number of active functional units (and issue width). Based on the results in Sections 6.1 and 6.2, we used the *CriticalIW* and the *HazardFU* algorithms respectively for these adaptations.

We report results for four processors: (1) *Base*, which is the base processor with no support for architectural adaptation; (2) *Inter*, which is *Base* enhanced with inter-frame adaptation as described in Sections 2 and 5; (3) *Intra*, which is *Base* enhanced with intra-frame adaptation as described above; and (4) *Inter+Intra*, which is *Base* enhanced with the integrated inter-frame and intra-frame algorithm described in Section 4, using the above individual inter-frame and intra-frame algorithms as its components. For each case, we evaluate both with and without DVS.

Figure 6(a) and Figure 6(b) show the total energy normalized to *Base* for all four processors, without and with DVS respectively. Each bar also shows the part of the total energy consumed by the instruction window, ALU, FPU, and the register file. Table 8 shows the mean relative savings in energy between different processor pairs. Each application missed less than 5% of its deadlines on all systems.

Savings from	Relative to	No DVS	DVS
Inter	*Base*	44%	18%
Intra	*Base*	29%	24%
Inter+Intra	*Base*	46%	27%
Intra	*Inter*	-26%	8%
Inter+Intra	*Inter*	5%	11%
Inter+Intra	*Intra*	24%	4%

Table 8 Mean relative energy savings for different processor pairs.

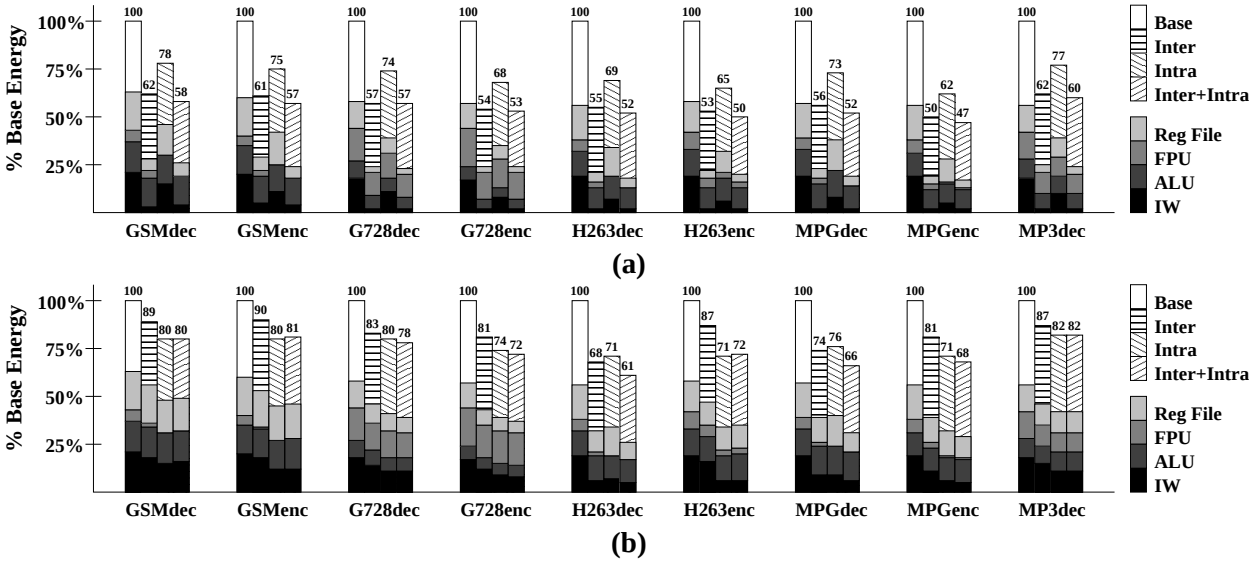

Figure 6 Energy consumption (normalized to *Base*) of processors capable of both inter-frame and intra-frame adaptation. (a) Without DVS. (b) With DVS.

6.3.1 Overall Results

Overall, Table 8 shows the following key results:

- *Intra vs. Base:* A purely intra-frame adaptation algorithm combining both instruction window and functional unit adaptation shows significant energy savings over *Base*, both without DVS (average of 26%) and with DVS (average of 24%). Compared to the results from Sections 6.1 and 6.2, we find that the benefits from the two adaptations are almost additive. (Inter-frame adaptation also shows high savings, but these were already known from previous work [16].)

- *Intra vs. Inter:* Without DVS, *Inter* shows significant energy savings over *Intra* for all applications (average of 21%). With DVS, however, the results are less clear cut. Overall, *Intra* shows modestly superior savings on average (8%), with a large maximum savings of 18% for H263enc. However, for two applications, *Inter* performs slightly better than *Intra* (by 3% for H263dec and 2% for MPGdec).

- *Integrated Inter+Intra vs. pure Inter and pure Intra:* In all cases, the integrated algorithm provides almost the same or better energy savings than the individual inter-frame or intra-frame approach. Without DVS, the additional benefit over *Inter* (the best individual approach) is relatively modest (average of 5%, maximum of 7%). With DVS, the average benefit over *Inter* is 11%, with a maximum of 17%. The average benefit over *Intra* is 4%, with a maximum of 14%.

6.3.2 Detailed Analysis

We next analyze the reasons for the relative performance of *Inter, Intra,* and *Inter+Intra* in more detail. Table 9 shows the mean number of active functional units and the instruction window size selected by *Inter*, *Intra*, and *Inter+Intra*. Figure 7(a) and Figure 7(b) show the computation slack remaining for *Base*, *Inter*, *Intra*, and *Inter+Intra* without and with DVS respectively.

Without DVS									
Application	**Inter-frame**			**Intra-frame**			**Inter+Intra**		
	IW	**ALU**	**FPU**	**IW**	**ALU**	**FPU**	**IW**	**ALU**	**FPU**
GSMdec	16	2.0	1.0	117	4.4	0.0	32	2.0	0.0
GSMenc	32	2.0	1.0	83	4.7	0.0	32	2.0	0.0
G728dec	16	2.0	1.0	99	2.8	1.4	16	1.7	1.0
G728enc	16	2.0	1.0	83	2.3	1.8	16	1.5	1.0
H263dec	16	2.0	1.0	56	4.4	0.0	16	2.0	0.0
H263enc	16	2.0	1.0	50	3.6	0.4	16	1.8	0.2
MPGdec	16	2.0	1.0	62	4.9	0.0	16	2.0	0.0
MPGenc	16	2.0	1.0	43	3.5	0.1	16	1.9	0.1
MP3dec	16	2.0	1.0	92	3.7	1.1	16	1.6	0.7
With DVS									
Application	**Inter-frame**			**Intra-frame**			**Inter+Intra**		
	IW	**ALU**	**FPU**	**IW**	**ALU**	**FPU**	**IW**	**ALU**	**FPU**
GSMdec	128	6.0	1.0	117	4.4	0.0	121	4.5	0.1
GSMenc	128	6.0	1.0	83	4.7	0.0	86	4.7	0.0
G728dec	100	4.4	1.8	100	3.1	1.3	85	2.6	1.2
G728enc	84	3.5	3.0	81	2.2	1.8	65	2.0	1.7
H263dec	48	6.0	1.0	56	4.4	0.0	40	4.2	0.0
H263enc	128	4.0	2.0	51	3.6	0.5	51	3.6	0.3
MPGdec	64	6.0	1.0	63	4.9	0.0	42	4.7	0.0
MPGenc	64	6.0	1.0	43	3.5	0.1	39	3.4	0.1
MP3dec	128	4.0	2.0	92	3.7	1.1	93	3.7	1.1

Table 9 Mean instruction window size, active ALUs, and active FPUs selected by *Inter*, *Intra*, and *Inter+Intra*.

Analysis without DVS

Without DVS, *Inter* saves more energy than *Intra* because *Inter* has the ability to exploit slack and sacrifice performance to save energy. Since the slack is relatively large for all applications, *Inter* selects the simpler (less aggressive) architecture configurations for all applications, as seen in Table 9. *Intra*, on the other hand, attempts to maintain performance despite the existence of so much slack. One could potentially design intra-frame algorithms capable of trading off performance for energy savings, but that is beyond the scope of this paper.

Inter+Intra is the same as or slightly better than *Inter* by itself in all cases, because it is able to shut down resources that *Inter* cannot, as seen in Table 9. These include the last FP unit and the second to last ALU (all configurations available for inter-frame adaptation had at least 2 ALUs). The absolute gains are

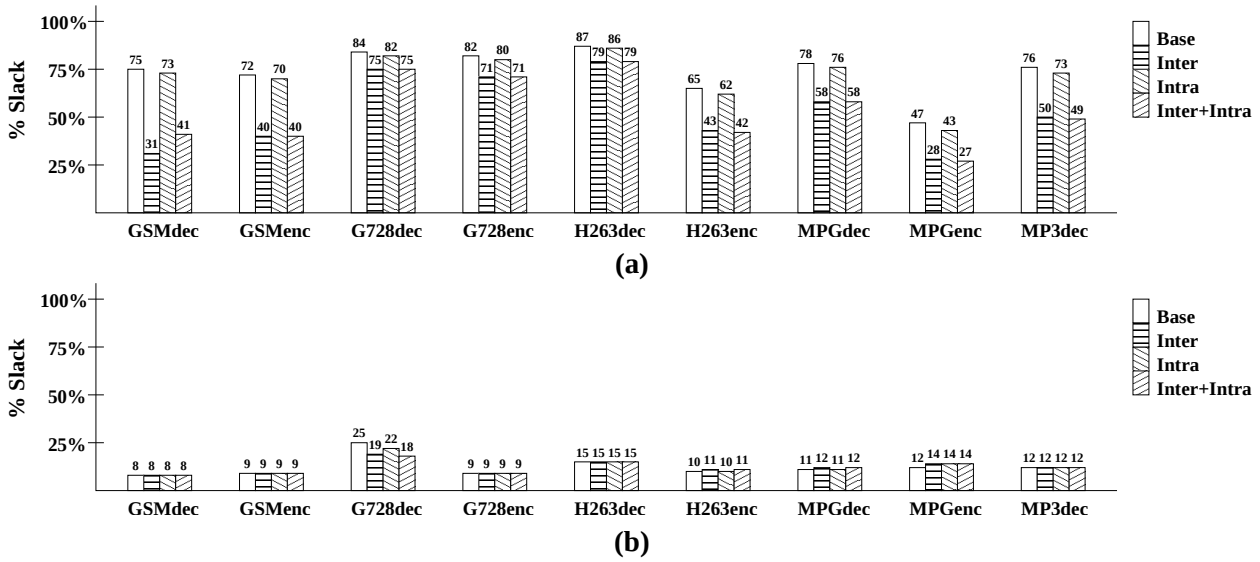

Figure 7 Remaining execution slack (as a percentage of the deadline) for processors capable of both inter-frame and intra-frame adaptation. (a) Without DVS. (b) With DVS.

modest, however, because the simple architectures picked by *Inter* offer limited opportunities for further adaptation by *Intra*. These gains might be further reduced if *Inter* were given a larger set of configurations to choose from. (Recall that we limited the configurations to save simulation time.)

Clearly, these results are sensitive to the slack available. When the slack is significantly smaller (i.e., the application deadlines are even tighter), the inter-frame algorithm chooses more aggressive architectures (as with DVS), and intra-frame algorithm saves more energy than the inter-frame algorithm as shown in Figure 8(a). The slack present for those cases is given in Figure 8(b). For these results, deadlines were chosen so that the frame with the longest execution with intra-frame adaptation will just meet the deadline. In all cases *Inter+Intra* is the best though most of the energy savings come from *Intra*.

Analysis with DVS

The ability of the processor to exploit slack through DVS reduces *Inter*'s advantage of exploiting slack over *Intra*. Further, as illustrated by Table 9, with DVS, in many cases, *Inter* tends to pick fairly aggressive architectures. That is, much of the slack is exploited through reduced frequency/voltage even with *Inter*, as explained in more detail in [16].[4] In these cases, since *Intra* can exploit intra-frame variability, it is able to power down more resources than *Inter* for parts of the execution, without losing much performance. As

[4]The aggressive architectures give high enough IPCs that it is most energy effi cient to choose them and exploit most of the slack through reduced frequency and voltage with DVS.

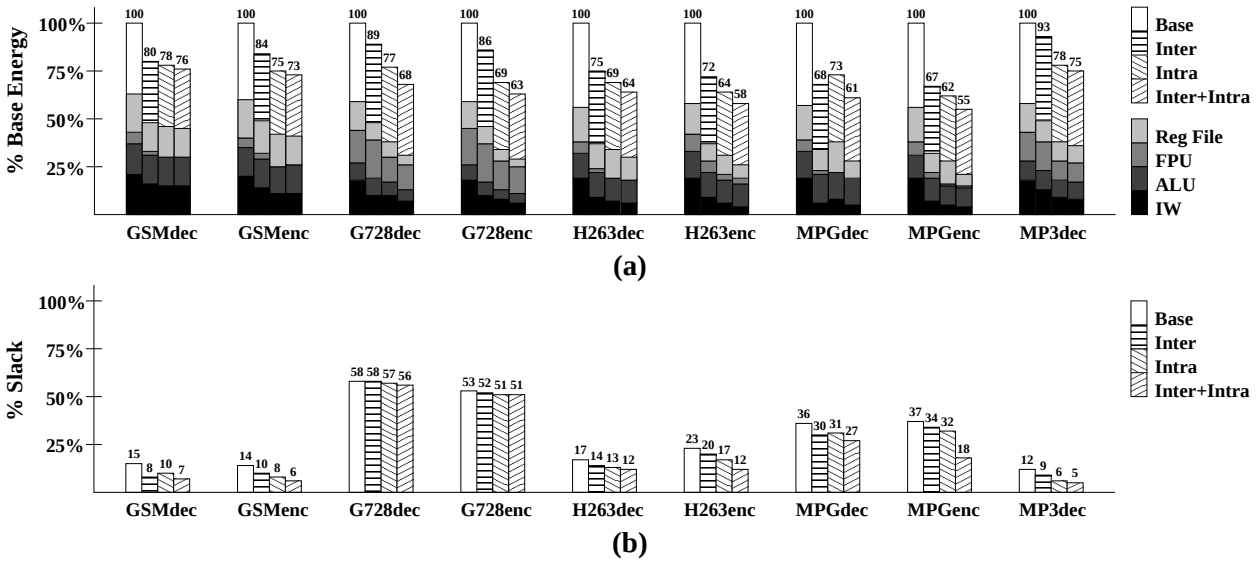

Figure 8 Energy and Execution Slack without DVS but with very tight deadlines. (a) Energy consumption (normalized to *Base*) of processors capable of both inter-frame and intra-frame adaptation. (b) Remaining execution slack (as a percentage of the deadline) for processors capable of both inter-frame and intra-frame adaptation.

a result, with DVS, *Intra* does better than *Inter* for seven of the nine applications, giving fairly significant gains for some of them.

For the other two applications (H263dec and MPGdec), *Intra* cannot save as much as *Inter* for two reasons. First, for both applications, *Intra* cannot shut down part of the register file like *Inter* can (Section 5). Second, for H263dec, *Inter* finds it beneficial to pick a simpler architecture to exploit slack. *Intra* is unable to reduce the instruction window size as much as *Inter* because that would adversely affect performance.

Inter+Intra has the benefits of both types of adaptation, and so saves almost the same or more energy than both *Inter* and *Intra*. For applications for which *Inter* is more effective than *Intra*, (e.g., H263dec and MPGdec), *Inter+Intra* is also able to trade off performance for energy savings and choose a less aggressive architecture. Furthermore, if parts of the frames for such applications require an even less aggressive architecture than that chosen by the inter-frame algorithm, *Inter+Intra* can save even more energy than *Inter*. This happens for both H263dec and MPGdec (*Inter+Intra* saves more than 10% relative to *Inter* for both). For applications for which *Intra* is more effective than *Inter*, *Inter+Intra* is also able to power down resources when they provide little performance. Furthermore, if the application requires an instruction window size smaller than the maximum for high energy efficiency (i.e., the inter-frame algorithm chooses less than maximum), there are two possible additional benefits for *Inter+Intra*. First, it will also be able to shut

down part of the register file, which *Intra* cannot do. This provides savings relative to *Intra* for MPGenc. Second, for parts of the frame which require a large instruction window for high performance, but for which a smaller instruction window has better energy efficiency, *Inter+Intra* will use a smaller instruction window (since the size is bounded by the inter-frame algorithm). This provides savings relative to *Intra* for G728enc.

6.3.3 Summary and Discussion

Overall, our proposed combination of inter-frame and intra-frame architecture adaptation works best across all configurations studied. Without DVS, for our applications and systems, our results clearly show that inter-frame architecture adaptation is beneficial and provides most of the benefits when there is significant amount of execution slack is present. However, when there is lower slack in the system, the intra-frame architecture adaptation becomes more beneficial. Given that the amount of slack available in a system is most likely a dynamic quantity (dependent on the total load on the system) and not predictable at design time, it could be beneficial to implement intra-frame architecture adaptation (integrated with inter-frame architecture adaptation) in a system without DVS as well.

For systems with DVS, our results clearly show that intra-frame architecture adaptation is beneficial for several applications, and provides most of the benefits. There are some applications, however, where inter-frame architecture adaptation works better. These are the applications where the inter-frame algorithm would pick relatively simpler architectures and are thoroughly discussed in [16]. We note that for a system that already implements inter-frame DVS and intra-frame architecture adaptation, adding inter-frame architecture adaptation does not introduce much additional hardware complexity (assuming that the intra-frame control algorithm, including the profiling phase, is implemented in software [16]). This again makes the case for supporting integrated inter-frame and intra-frame adaptation.

7 Conclusions

Hardware adaptation has been shown to be an effective technique for saving energy for real-time multimedia applications. We examine both dynamic voltage/frequency scaling (DVS) and architectural adaptation. Previously, these have been combined for real-time multimedia applications with a control algorithm operating at the granularity of a frame (inter-frame). That algorithm took advantage of computation slack to slow the processor down in order to save energy. We consider architectural adaptation at a finer granularity (intra-frame), where the control algorithms attempt to save energy by powering down under-utilized resources while maintaining performance.

In our first set of contributions, we evaluate previous adaptation control algorithms proposed in a non-real time context for intra-frame architectural adaptation for real-time multimedia applications. We consider adapting the size of the instruction window and the number of active functional units (and associated instruction issue width). We also propose new algorithms and ways of combining different algorithms for these adaptations. We find that intra-frame architectural adaptation is effective for real-time multimedia applications with and without DVS. The algorithms we propose do marginally better than the best previously proposed algorithms.

In our second set of contributions, we compare inter-frame, intra-frame, and a new combination of inter- and intra-frame architectural adaptation algorithms, both with and without inter-frame DVS. The new combination of intra- and inter-frame levels of adaptation exploits both the variability in resource utilization within a frame and computation slack at the frame granularity.

Our proposed combination of inter-frame and intra-frame architectural adaptation works best across all configurations studied. However, without DVS, inter-frame architectural adaptation provides most of the energy savings when there is a significant amount of slack. With less slack, intra-frame architectural adaptation provides more savings, making a stronger case for implementing the combined approach in a system without DVS. With DVS, intra-frame architectural adaptation provides most of the energy savings, but for some applications, inter-frame architectural adaptation is better. The overhead for adding inter-frame architectural adaptation to a system with inter-frame DVS and intra-frame architectural adaptation is small, making it beneficial to implement the combined inter-frame and intra-frame approach for the applications that can exploit it. Therefore, regardless of whether a system supports DVS or not, our results provide considerable justification for implementing a combined inter-frame and intra-frame architectural adaptation algorithm.

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] G. Blalock. Microprocessors Outperform DSPs 2:1. *Microprocessor Report*, December 1996.

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

[7] D.Grunwald, P.Levis, K.I.Farkas, C. III, and M.Neufeld. Policies for dynamic clock scheduling. In *Proc. of the 4th Symposium on Operating Systems Design and Implementation*, 2000.

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

[9] G. K. Dmitry Ponomarev 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.

[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] 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.

[16] 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.

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

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

[19] R. B. Lee and M. D. Smith. Media Processing: A New Design Target. *IEEE Micro*, August 1996.

[20] Y.-H. Lee and C. Krishna. Voltage-Clock Scaling for Low Power Energy Consumption in Real-Time Embedded Systems. In *Proc. of the 6th Intl. Conference on Real-Time Computing Systems and Applications*, 1999.

[21] 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.

[22] 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.

[23] M.Weiser, B.Welch, A.Demers, and S.Shenker. Scheduling for rediced CPU energy. In *Proc. of the 1st Symposium on Operating Systems Design and Implementation*, 1994.

[24] V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM Reference Manual version 1.0. Technical Report 9705, Department of Electrical and Computer Engineering, Rice University, August 1997.

[25] P. Pillai and K. G. Shin. Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems. In *Proc. of the 18th ACM Symp. on Operating Systems Principles*, 2001.

[26] J. Pouwelse, K. Langendoen, and H. Sips. Energy Priority Scheduling for Variable Voltage Processors. In *Intl. Symp. on Low-Power Electronics and Design*, 2001.

[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*, 2000.