Joint Local and Global Hardware Adaptations for Energy*

Ruchira Sasanka Christopher J. Hughes Sarita V. Adve

University of Illinois at Urbana-Champaign
Department of Computer Science
{sasanka,cjhughes,sadve}@cs.uiuc.edu

ABSTRACT

This work concerns algorithms to control energy-driven architecture adaptations for multimedia applications, without and with dynamic voltage scaling (DVS). We identify a broad design space for adaptation control algorithms based on two attributes: (1) when to adapt or *temporal* granularity and (2) what structures to adapt or *spatial* granularity. For each attribute, adaptation may be *global* or *local*. Our previous work developed a temporally and spatially global algorithm. It invokes adaptation at the granularity of a full frame of a multimedia application (temporally global) and considers the entire hardware configuration at a time (spatially global). It exploits int*er*-frame execution time variability, slowing computation just enough to eliminate idle time before the real-time deadline.

This paper explores temporally and spatially local algorithms and their integration with the previous global algorithm. The local algorithms invoke architectural adaptation within an application frame to exploit int*ra*-frame execution variability, and attempt to save energy without affecting execution time. We consider local algorithms previously studied for non-real-time applications as well as propose new algorithms. We find that, for systems without and with DVS, the local algorithms are effective in saving energy for multimedia applications, but the new integrated global and local algorithm is best for the systems and applications studied.

1. INTRODUCTION

Multimedia applications have become an important workload for a variety of systems employing general-purpose processors [7, 8, 21]. A large number of these systems are powered by batteries, making energy a first class resource constraint. To save energy, researchers have proposed hardware adaptation, including dynamic voltage and frequency scaling, or DVS (e.g., [13, 14, 25, 27]), and architecture adaptation (e.g., changing the instruction window size [6, 9, 11, 26], changing the number of functional units and/or issue width [3, 23], and others [2, 4, 12, 15, 16, 22]). This work concerns control algorithms for such hardware adaptations for multimedia applications, and is part of the Illinois GRACE project which seeks to coordinate system-wide hardware and software adaptations [1].

Two key questions must be addressed when designing adaptation control algorithms – (1) when to adapt, or the *temporal* granularity of adaptation, and (2) what structures to adapt, or the *spatial* granularity of adaptation. Our previous work developed an integrated control algorithm for DVS and architecture adaptation for multimedia applications and used the following observations to address the above questions [19]. 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. Since the slack may vary from frame to frame [17], the ideal hardware configuration may also vary from frame to frame, motivating inter-frame adaptation.

The previous algorithm, therefore, invokes adaptation at the start of each frame [19]. It predicts the lowest energy hardware configuration (voltage/frequency and architecture) that can meet the deadline for the next frame, and uses that configuration to execute the frame. The execution time and energy prediction exploits special characteristics of multimedia applications. Thus, this algorithm operates at the temporal granularity of a full frame and the spatial granularity of the entire processor configuration (it adapts all architecture components and the voltage/frequency together). We refer to this algorithm as having global temporal and spatial granularity, or as a *global* algorithm.

This paper explores *local* adaptation algorithms, both in time and space, and their combination with the previous global algorithm. The local algorithms seek to exploit int*ra*-frame variability in resource utilization, invoking adaptations periodically within a frame (local temporal granularity). It is difficult to precisely predict the performance impact of individual adaptations and their mutual interactions

*This work is supported in part by the National Science Foundation under Grant No. CCR-0096126, EIA-0103645, CCR-0209198, a gift from Motorola Inc., and the University of Illinois. Sarita V. Adve is also supported by an Alfred P. Sloan Research Fellowship and Christopher J. Hughes is supported by a Richard T. Cheng fellowship.

ASPLOS X 10/02 San Jose, CA, USA

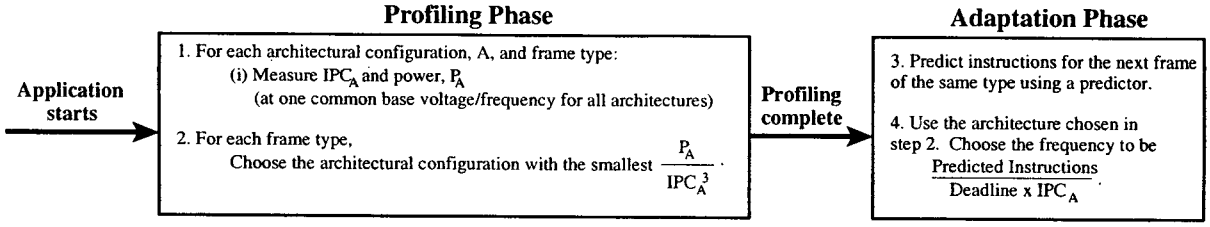

Figure 1: The previous global algorithm for choosing hardware configurations for a system with continuous DVS.

at the intra-frame granularity. The local algorithms therefore control individual hardware structures (i.e., local spatial granularity) and attempt to adapt without affecting execution time (i.e., they do not exploit slack). To continue to exploit slack, we propose an algorithm that integrates the global and local approaches.

We also identify the possibility of other control algorithms to fill the rest of the design space of temporal and spatial granularity; e.g., temporally local and spatially global, or temporally global and spatially local algorithms. These variations, however, are outside the scope of this paper.

This paper makes two sets of contributions. First, we study local architecture adaptation algorithms for multimedia applications, both without and with DVS. The basic approach – adapting individual components 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 [2, 3, 4, 6, 9, 11, 22, 23, 26]. The focus of our work is on the effectiveness of local adaptation control algorithms for multimedia applications and their interaction with global architecture adaptation and DVS. We focus on two architecture adaptations – 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 local algorithms for these adaptations and also propose new algorithms that are more intuitive. We find that, for our system and suite of multimedia applications, all algorithms show modest to significant energy savings without much increase in execution time, both without and with DVS. The new algorithms are slightly better than the older ones.

For the second set of contributions, we develop an algorithm to combine local and global architecture adaptation and global DVS. We do not consider local DVS because of its impact on performance. We report results without and with DVS for (1) purely global architecture adaptation (as in [19]), (2) purely local architecture adaptation, and (3) the new integrated algorithm. We find that for our system and multimedia application suite, the new integrated algorithm performs the best in all cases because it can exploit both computation slack at the frame granularity and variability in resource utilization within a frame. The amount of computation slack with the base architecture generally determines whether the global or local adaptation provides the majority of the benefit in the integrated approach.

2. PREVIOUS GLOBAL ALGORITHM

The previous global adaptation control algorithm [19] invokes adaptations at the granularity of a frame. At the beginning of each frame, it 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 flexible continuous DVS system here (e.g., Intel's XScale processor approximates such a system [20]). Our work would be equally applicable to a discrete DVS system. The key aspects of the global algorithm based on continuous DVS are summarized in Figure 1 (taken from [19]) and described in more detail below.

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

Previous work showed that for several multimedia applications and systems, 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 [17, 19]. 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, it can be derived 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, and are discussed further in [19].

Based on the above, after profiling is complete, the algorithm computes $\frac{P_A}{IPC_A{}^3}$ for each architecture configuration and frame type. It chooses the architecture with the smallest such value to execute all frames of that type. Choosing the execution frequency for a frame requires knowing the number of instructions in the frame. This is determined during the adaptation phase, using simple history-based predictor [19] before the start of a frame. The expression $\frac{I}{D \times IPC_A}$ can then be used to determine the frequency. Since frame IPCs are only roughly constant, 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.

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

The global control algorithm can be implemented in software or hardware. A hardware implementation requires communication from the software to indicate when a new frame starts, the type of the frame, and the deadline.

3. LOCAL ALGORITHMS

The global algorithm exploits the fact that a real-time application can be slowed as long as it meets its deadline. Prediction of the performance impact of adaptations is straightforward at the global (inter-frame) level, but is difficult at the local (intra-frame) level. Therefore, local algorithms explored here consider adaptations that, ideally, do not affect execution time. In particular, not all resources always contribute significantly to performance. Thus, we can deactivate under-utilized resources with little or no performance impact, but significant energy savings. Since resource utilizations can change quickly, we consider local adaptations with small switching times and assume hardware implementations of the local algorithms. As an indicator of intra-frame variability of the execution profile, we measured the standard deviation of the IPC over 200 cycle intervals for a sample frame on each application and the base architecture in Section 5. The standard deviation ranged from 20% to 35% as a percentage of the mean IPC of the full frame.

In this paper, we examine two local architecture 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 we study systems with DVS, we do not consider local DVS control because DVS necessarily affects performance.

All the local algorithms discussed here take a common approach of monitoring certain statistics over a pre-determined *period* (e.g., every 200 cycles), and using them to determine whether to increase or decrease the adapted structure (e.g., instruction window size, number of functional units, issue width) for the next period. The decisions for decreasing and increasing the structure are made independently. In principle, any pair of algorithms for increasing and decreasing a structure can be used together.

3.1 Adapting Instruction Window Size

We assume the instruction window is divided into several equal segments, and a contiguous set of segments can be deactivated at any cycle [6, 11]. Several local algorithms have been proposed to control the active size of such a window and have been evaluated for non-real-time applications [6, 9, 11, 26]. Section 3.1.1 discusses the state-of-the-art and Section 3.1.2 discuss a new algorithm for increasing the size.

3.1.1 State-of-the-art

We evaluate an algorithm by Folegnani et al. as representing the state-of-the-art [11]. This algorithm decreases the instruction window size by one segment if the number of committed instructions that issued from the youngest segment during the previous period is smaller than a threshold [11]. Thus, segments of the window that clearly did not contribute to the overall IPC are deactivated, with relatively low overhead (primarily 1 bit per instruction window entry). A disadvantage is that the youngest segment stays activated even if the instructions issued from it are not on the program's critical path. Issuing such instructions early may not contribute to IPC, but this is not detected.

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 IPC will benefit from the additional instructions that fit into a larger window, potentially wasting energy. Conversely, in some cases, this may potentially degrade IPC, if the increase does not occur early enough to meet the increased demands of the application.

Other algorithms proposed are based on window occupancy [6, 9, 26], which we found to be less effective for the applications and architecture studied here (Section 5). The algorithm by Ponomarev et al. decreases the window size based on the average occupancy during the previous period [26]. Dropsho et al. propose an extension that decreases the size based on an approximation of the distribution of window occupancy rather than the average [9] (this algorithm supercedes the one in [6]). Both algorithms are fundamentally more conservative than the one by Folegnani et al. – they can consume more energy without benefitting IPC when instructions are present in the youngest segment of the window but are not able to issue.[2] The occupancy-based algorithms increase the window size if there are enough dispatch stalls due to the instruction window being full (window overflows). However, for several of the applications and the architecture we evaluate (e.g., with unified reorder buffer and issue queue, discussed in Section 5), we find that the instruction window is full for much of the execution without necessarily contributing to IPC, making such a scheme less effective.

3.1.2 New Algorithm for Increasing Window Size

We propose a new algorithm (Figure 2) for increasing the size of the instruction window, based on a prediction of the resulting benefit in IPC. To obtain this prediction, we estimate the number of (retirement) stall cycles that could have been avoided with a larger instruction window. We say an instruction I is *stalled* if it is incomplete and at the head of the instruction window [24]. A larger instruction window can potentially avoid such a stall by providing more instructions ahead of I to overlap with I's latency, as illustrated in Figure 3. 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 activated. Several aspects of the design required making a tradeoff between accuracy and hardware and energy overhead. We describe the design we chose next; other variations that improve accuracy or reduce overhead further are possible.

The algorithm computes the IPC over a fixed period. When an instruction is fetched into the youngest segment of the window, it checks to see if its operands are already available. If so, a larger window could potentially have allowed for more overlap for that instruction (Figure 3). We optimistically estimate that the additional overlap could have been $\frac{Number\ of\ deactivated\ entries}{IPC\ from\ last\ period}$ cycles, and set a tag for

[2]The occupancy-based algorithms [9, 26] can reduce the instruction window size by multiple segments at a time. The other algorithms ([11] and our new algorithm in Section 3.1.2) can be similarly extended, but we do not evaluate that extension here.

On entry of instruction I to instruction window:	if (the operands of I are ready) I→IWtag = MaxOverlap

On completion of instruction I:	for each instruction J consuming I's result
	if (I produced the last operand of J)
	if (I did not stall)
	J→IWtag = I→IWtag
	else if (I stalled for S cycles)
	J→IWtag = max(0, I→IWtag – S)

On retirement of instruction I:	if (I→IWtag > 0)
	if (I stalled for S cycles)
	Counter += min(I→IWtag, S)
	if (Counter > Threshold) {
	Increase the window size
	Counter=0

At the end of each period:	MaxOverlap = Deactivated Entries / IPC from last period Counter=0

Figure 2: A new algorithm for increasing the instruction window size. An instruction is said to stall if it reaches the head of the instruction window before completion.

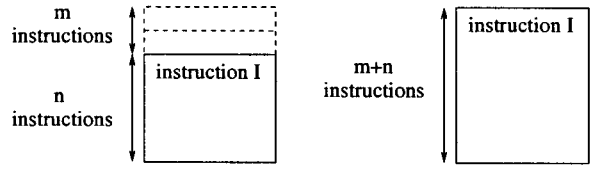

Figure 3: Additional overlap from a larger window. The left part shows an instruction window with n entries activated and m entries deactivated. Instruction I arrives at the top of the activated portion of the window and finds all its operands available. Thus, I is overlapped with $n-1$ instructions. If the instruction window were fully activated (the right part) and I arrived at the top, it could have an additional m instructions for overlap.

this instruction, called *IWtag*, with this value. This computation is optimistic 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 provide increased overlap to an instruction that does not have its operands available on entry, if the larger window enabled its operands to be generated early. The increased overlap would be the same as that for the producer of the last pending operand of the consumer, but 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[3]; if this instruction stalled before completion for S cycles, then we reduce its IWtag by $min(IWtag,\ S)$ 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.

[3]The idea of passing tags was inspired by the token passing along "last arriving edges" of Fields et al. [10]

The value of IWtag gives the additional overlap that an instruction could get with a larger window. If a tagged instruction stalls the processor for S cycles, then we estimate $min(IWtag,\ S)$ as the number of stall cycles that could be avoided by the additional overlap. We accumulate the avoidable stall cycles in a counter, and reset the counter at the end of the period. If the counter exceeds a threshold, we increase the instruction window size by one segment, reset the counter, and start a new period.

Our new algorithm potentially alleviates the limitations of the previous algorithm by Folegnani et al. 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.

A potential disadvantage of the new 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. Other overheads include the logic for calculating the tags, logic for calculating the maximum possible overlap at the end of a period (*MaxOverlap* in Figure 2), a shifter to compute IPC, and some counters. The energy consumed by the tag computation, shifter, and counters is likely to be negligible. Computing *MaxOverlap* may require a more expensive energy-hungry divide; however, this value can be calculated less frequently and in an approximate way to reduce the overhead. In our experiments, we calculated it once every two periods or when the instruction window is resized.

Finally, it is also possible to design an algorithm for decreasing the instruction window size based on the new technique to increase the size. For example, we could decrease the size if the number of "avoidable stalls" in a period is below some threshold. We experimented with this algorithm, but found that it did not perform as well as that by Folegnani et al. for decreasing the window size. A smarter algorithm would need to more precisely determine when instructions issued from the youngest segment of the instruction window are not critical instructions.

3.1.3 *Algorithms Evaluated and Parameters Used*

As discussed earlier, we evaluate the algorithm by Folegnani et al. (for both increasing and decreasing the window size) as representing the state-of-the-art, and call this *PeriodicIW*. We also report results for the new algorithm for increasing the window size combined with the algorithm by Folegnani et al. for decreasing the window size, and call this *StallIW*.

A key issue for local adaptation algorithms is that they use a number of different parameters that 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%. The specific parameters are as follows.

PeriodicIW considers reducing the 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 degradation. *PeriodicIW* increases the instruction window size by one segment every five periods (starting from the last time it changed the size).

For comparison, Folegnani et al. chose 1000 cycles as the period for decreasing the window size and also chose five periods for increasing it. We chose a smaller period because the overhead is small, and a smaller period allowed a faster response to changing requirements of the application.

StallIW also considers reducing the instruction window size with a period of 200 cycles. It reduces the 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 for *PeriodicIW* because *StallIW* can more rapidly increase the instruction window size when needed. *StallIW* increases the 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, to give priority to the decreasing algorithm).

3.2 Adapting Functional Units and Issue Width

Several algorithms have been recently proposed to change the number of activated functional units and the consequent instruction issue width, and have been evaluated for non-real-time applications [3, 23]. Section 3.2.1 discusses the state-of-the-art and Section 3.2.2 discusses new algorithms and combinations explored for this work.

3.2.1 State-of-the-art

The algorithms below were proposed for an architecture with two clusters (e.g., Alpha 21264), where the activation and deactivation of functional units is performed at the granularity of a half or full cluster. Extensions for finer control of the functional units in a non-clustered architecture such as modeled in this paper (Section 5) are straightforward.

Maro et al. considered several algorithms to control whether to have functional units in one or two clusters activated [23]. The best algorithm (LP1 in [23]) reported uses mean functional unit utilization over a period to determine whether to increase or decrease the number of active units (i.e., increase if utilization is high and vice versa). The algorithm is simple to implement but has some disadvantages: (1) it activates and deactivates a functional unit without consideration of whether the instructions using it are on the critical path, and (2) it may activate a unit that will not be used.

Maro et al. consider two other strategies for deactivating functional units: deactivate on low committed IPC, and deactivate 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. discuss, but do not evaluate, another strategy for activating functional units. This scheme tracks the number of structural hazards for each instruction. If the total number of hazards for the instructions currently in the instruction window is over some threshold, the algorithm activates a cluster. This algorithm can quickly increase the number of active units in the event of a burst of hazards.

Bahar and Manne developed an algorithm to deactivate a full cluster or half the ALUs and all of the FPUs in one cluster [3]. Their algorithm is based on the number of instructions issued per cycle (issue IPC) to each type of unit (ALU or FPU) [3]. 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. If the issue IPC is below the lower threshold, the number of active units is decreased. 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 [3]). 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 (for both increasing and decreasing the number of active units), and call it *UtilFU*.

3.2.2 New Algorithms and Combinations

A new algorithm for increasing the number of active functional units could be based on an estimate of the resulting benefit in IPC, analogous to the new algorithm for increasing the size of the instruction window in Section 3.1.2. We explored this option and found that its larger relative overhead made it perform worse than most others.

We propose, and report results for, an algorithm that combines a utilization-based scheme and a structural hazard based scheme for respectively decreasing and increasing the number of active units. We call this scheme *HazardFU*. The algorithm for increasing the number of active units is similar to the structural hazard based scheme proposed (but not evaluated) by Maro et al. Rather than tracking the number of hazards seen by each instruction in the instruction window, we track the total number of hazards (for each type of unit) seen by all instructions within a period. If the total exceeds a threshold before the end of the period, we increase the number of active units by one. Such a scheme is likely to better anticipate functional unit usage than a mean utilization-based scheme. The overhead for this scheme is simply some counters to track hazards and unit utilization. The energy consumed by these is likely negligible.

3.2.3 Parameters for the Algorithms

In our experiments, one integer ALU is always active for both *UtilFU* and *HazardFU*. Also for both, when all FP

Properties	Global adaptation	Local adaptation
Source of benefits	Per-frame execution time slack	Variability in resource usage within a frame
Temporal granularity of adaptation	Entire frame	Small intervals within a frame
Hardware features controlled	Global configuration	Individual (or small groups of) hardware features
Impact on execution time	May increase	No impact (ideally)
Basis of adaptation	Profiles mostly collected at start of application	Information from continuous monitoring
Implementation	Hardware or software	Mostly hardware
Inapplicable adaptations	Adaptations that cannot be invoked on the entire frame (e.g., shutting off all floating point units)	Adaptations with high overhead or that impact execution time (e.g., DVS)

Table 1: Comparison between global and local adaptation algorithms.

units are deactivated, if an FP instruction is fetched, an FP unit is activated. 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.

For *UtilFU*, due to the large number of parameters, we could not perform a full design space search, and use some from the LP1 scheme of [23]. Using our previously described criteria for choosing parameters (Section 3.1.3), *UtilFU* reduces the number of active units by one if the last unit is not used more than 4 cycles in the last period. However, *UtilFU* uses the LP1 criteria for deactivating the last FP unit; it does so when that unit is not used for three cycles in a row. Also as for LP1, if the last active unit of a given type has a utilization of at least 86% for the previous period, the algorithm increases the number of active units of that type by one. While [23] 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.

We chose all parameters for *HazardFU* in the same manner as described in Section 3.1.3. *HazardFU* increases the number of active 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 as for *UtilFU*). An exception is for the last available FP unit, which is deactivated only if it is not used at all within the last period. We also use a 200 cycle period for *HazardFU*.

4. JOINT GLOBAL/LOCAL ADAPTATION

We combine the global and local adaptation control algorithms in a simple way, resulting in two parallel but integrated control loops in the system.

The global part of the integrated algorithm performs the profiling and adaptation phases as before, but with the following change. During profiling, the local algorithms are also invoked. Ideally, the global algorithm will now see lower average power for each candidate architecture with little change in IPC (vs. without local adaptations). As before, the algorithm estimates the lowest energy architecture to be the one with the lowest profiled $\frac{Power}{IPC^3}$. During the adaptation phase, again, the local algorithms are invoked on all frames. The global 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. Thus, the global algorithm automatically compensates for any IPC degradation caused by the local algorithms. This avoids missed deadlines but at the cost of reduced energy savings from the local algorithms.

The local part of the integrated algorithm is always invoked and is mostly oblivious to the global part, except for the following. The global algorithm establishes the maximum configuration for each resource for the local algorithms. A local algorithm must respect this maximum since its goal is to minimize energy without changing the IPC obtained by the globally chosen configuration. Thus, if the global algorithm chooses aggressive architectural configurations, then the local algorithm has substantial potential for exercising adaptations. However, less aggressive global choices may leave little potential for local adaptation.

The global and local algorithms play different roles. The global algorithm exploits per-frame execution time slack, and adapts at the granularity of a full frame and the full hardware configuration. The local algorithms exploit variability in resource usage within a frame, and adapt at the granularity of small intervals within a frame and control individual (or small groups of) hardware features. In particular, the larger temporal granularity of adaptation for the global algorithm makes it better able to predict future execution behavior than the local algorithm. These distinctions lead to significant differences in the design and effectiveness of the algorithms, as summarized in Table 1. These differences include impact on execution time (slowdown vs. ideally no impact), techniques to determine the right adaptation (use of profile information vs. continuous monitoring), whether the algorithm can be implemented in software (as for global) or must be in hardware (as for local), and the applicability of different adaptations (adaptations that cannot be invoked on a full frame are inapplicable to the global algorithm while adaptations with high overhead and impacting execution time are inapplicable to local algorithms).

5. EXPERIMENTAL METHODOLOGY

5.1 Systems Modeled

We use the RSIM simulator [18] for performance evaluation and the Wattch tool [5] integrated with RSIM for energy measurement.

The base processor studied is similar to the MIPS R10000 and is summarized in Table 2. We assume a centralized instruction window with a unified reorder buffer and issue queue but a separate physical register file. Experiments with DVS assume a continuous frequency range from 100MHz to 1GHz and corresponding voltage levels derived from information available for Intel's XScale processor [20] as further discussed in [19].

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

Table 2: Base (default) system parameters.

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

Table 3: Workload description.

Experiments with instruction window adaptation assume eight entry instruction window segments and that at least two segments must always be active. A smaller instruction window requires fewer physical registers. Since we model a physical register file separate from the instruction window, reducing the register file size during execution requires "garbage collecting" register contents [9]. This is straightforward with global adaptation (since resizing occurs infrequently at the start of a frame which is typically a new context with no prior register state). We therefore deactivate one integer and one floating point physical register with each deactivated instruction window entry with global adaptation. With local adaptation, however, we do not change the register file size since it would be too much overhead.

Experiments with functional unit adaptation assume that the issue width is equal to the sum of all active functional units and hence changes with the number of active functional units. Consequently, when a functional unit is deactivated, the corresponding instruction selection logic is also deactivated. Similarly, the corresponding parts of the result bus, the wake-up ports of the instruction window, and ports of the register file are also deactivated.

We assume clock gating for all the components of all the processor configurations (adaptive and non-adaptive). If a component is clock gated (i.e., not accessed) in a given cycle, we charge 10% of its maximum power. To fairly 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 [11]. We assume that the resources that are deactivated by our adaptive algorithms do not consume any power. Thus, deactivating an unused component saves 10% of the maximum power of the component (i.e., the remaining power after clock gating).

We use the local adaptation control algorithms as described in Section 3. For global adaptation, we use the algorithm in [19] except that we increase the IPC leeway from 1% to 4% to make all applications have fewer than 5% missed deadlines on the base processor. The global algorithm is used to control DVS in all cases, even when no global architecture adaptation is performed. Further, for global adaptation, we profile all possible combinations of the following configurations (54 total): instruction window size $\in \{128,96,64,48,32,16\}$, number of ALUs $\in \{6,4,2\}$, and number of FPUs $\in \{4,2,1\}$. It may seem that profiling 54 configurations is inordinate overhead for a real system. However, it is feasible since only one frame of each type need be profiled for each configuration, a typical multimedia application executes many more frames (e.g., 30 frames a second for video), and profiling can be done as part of the application's execution. Nevertheless, we also performed experiments where we profiled a smaller (representative) subset of the possible configurations, and found similar results.

Regarding adaptation overheads, as in [19], we ignore time and energy overheads for invoking global adaptation since they are incurred only once per frame. A more detailed justification for this assumption appears in [19]. For local adaptation, we model the extra bits required in the instruction window for instruction window size adaptation (one bit for *PeriodicIW* and four bits for *StallIW*). As mentioned in Sections 3.1.2 and 3.2.2, other overheads, such as for control logic and counters, are likely to be small. We model a delay of 5 cycles to activate all deactivated components (we observed that the results are not very sensitive to this).

5.2 Workload Description

Table 3 summarizes the nine applications and inputs used in this paper. These were also used in [17, 19] and are described in more detail in [17] (for some applications, we use fewer frames). We do not use multimedia instructions in this study because many of our applications see little benefit from them and we lack a power model for multimedia enhanced functional units.

For the application deadlines, by default, we use the deadlines referred to as "tight" deadlines in [19].[4] For all but the video encoders, this deadline is three times the maximum processing time for a frame on the base processor. For the video encoders, we use longer deadlines (the full frame period) since even the base processor is not able to meet this deadline in some cases – for MPGenc, we additionally double the frame period.

[4]The deadlines are affected by interactions with the real-time scheduler which must consider all the applications in the system. This interaction is beyond the scope of this study.

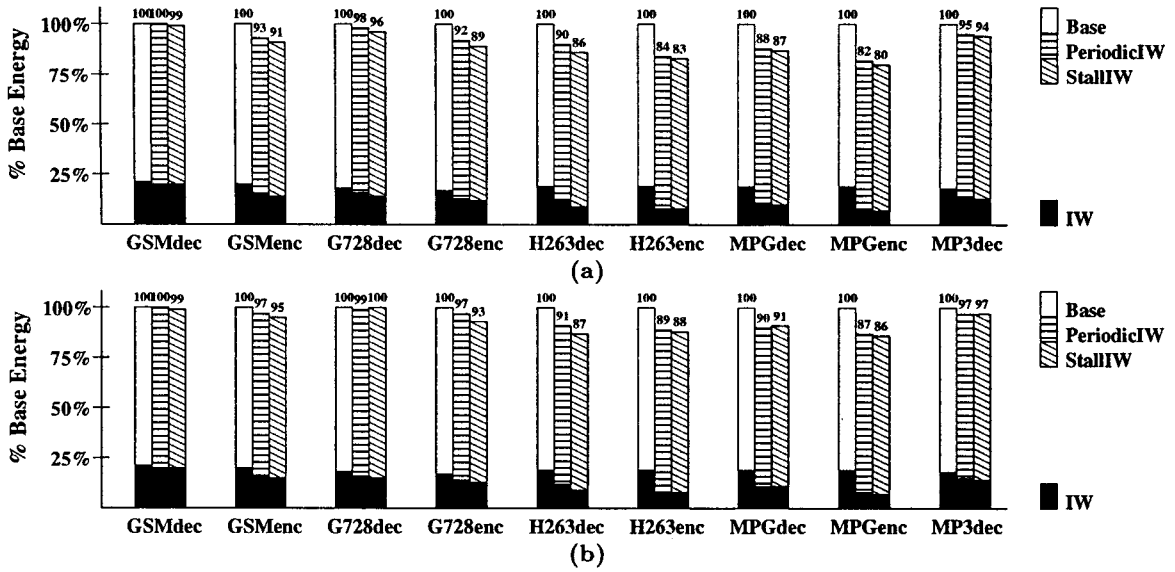

Figure 4: Energy consumption (normalized to *Base*) with instruction window size adaptation. (a) Without DVS. (b) With DVS. Each set of three bars represents *Base*, *PeriodicIW*, and *StallIW* respectively. The dark part in each bar shows the energy spent in the instruction window (IW).

The benefits of global adaptation are highly dependent on the original computation slack in a frame (i.e., remaining time from the end of the frame processing until the deadline). The default deadlines leave significant slack in all cases in the base (non-adaptive) processor without DVS (average slack for each application is ≥49% of the deadline). Therefore, when considering systems without DVS with global adaptation, we consider a set of tighter deadlines as well. These are set to be the longest time taken by a frame with local adaptation (on the base architecture without DVS). With these deadlines, significantly less slack remains on the base architecture without DVS (≤ 23% except for G728 and MPG).[5] Since DVS removes most slack from the system even with the default deadlines, we use only these deadlines with DVS (average slack is 8% to 16% for 8 of the 9 applications),

App.	No DVS		DVS	
	PeriodicIW	*StallIW*	*PeriodicIW*	*StallIW*
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 4: Mean instruction window size selected by *PeriodicIW* and *StallIW*.

6. RESULTS

Sections 6.1 and 6.2 respectively evaluate the local algorithms for adapting the instruction window size and functional units in isolation. Section 6.3 compares the global, local, and integrated global and local algorithms.

6.1 Local Instruction Window Adaptation

We evaluate three architectures for instruction window size adaptation: the default, non-adaptive *Base*, and *Base* enhanced with an adaptive window size controlled by *PeriodicIW* and by *StallIW*. Figures 4(a) and 4(b) show the total 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.

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

Without DVS, *PeriodicIW* saves an average of 9% energy over *Base* (maximum 18%), while *StallIW* saves an average of 11% (maximum 20%). With DVS, *PeriodicIW* saves 6% on average (maximum 13%) over *Base*, and *StallIW* saves 7% on average (maximum 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 *StallIW* and *PeriodicIW*, both without and with DVS, for all applications, *StallIW* saves about as much or more energy than *PeriodicIW*. However, the difference is small (1% on average, 4% maximum). The mean IPC degradation for *PeriodicIW* and *StallIW* without and with DVS is also similar – 3% and 4% respectively.

Table 4 shows the mean instruction window size chosen by each algorithm. *StallIW* 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 deactivation of *StallIW*. Also, the periodic activation

[5]The average slack for G728 and MPG is still high since these applications have multiple frame types and some frame types require much less execution time than the selected deadlines.

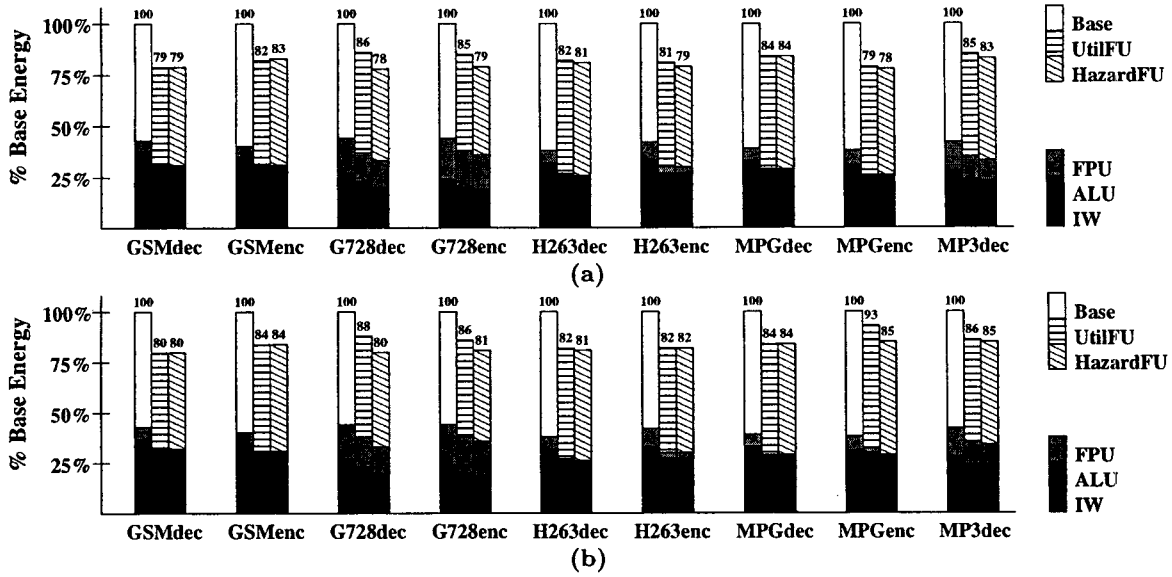

Figure 5: Energy consumption (normalized to *Base*) with functional unit and issue width adaptation. (a) Without DVS. (b) With DVS. Each set of three bars represents *Base*, *UtilFU*, and *HazardFU* respectively. The dark parts in each bar show the energy spent in the instruction window, ALUs, and FPUs.

App.	No DVS				DVS			
	UtilFU		*HazardFU*		*UtilFU*		*HazardFU*	
	A	F	A	F	A	F	A	F
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 5: Number of ALUs (A) and FPUs (F) selected by *UtilFU* and *HazardFU*.

strategy of *PeriodicIW* may hamper its ability to reduce the window size for as long as possible. Consequently, *StallIW* saves slightly more energy than *PeriodicIW*.

6.2 Local Functional Unit Adaptation

We evaluate three architectures for functional unit adaptation: the default, non-adaptive *Base*, and *Base* enhanced with adaptive functional units controlled by *UtilFU* and by *HazardFU*. Figures 5(a) and 5(b) show the total energy for each application, normalized to *Base*, without and with DVS respectively. Each bar also shows the part of the energy 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 Section 5, because adapting the issue width allows deactivation of parts of a number of structures.

The results show that both algorithms are very effective for multimedia applications. There is negligible difference between the two algorithms in most cases.

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

Table 5 shows the average number of active functional units chosen by each algorithm. The number of active ALUs and FPUs is very similar for *UtilFU* and *HazardFU* for all applications except G728dec and G728enc, where *HazardFU* saves more energy than *UtilFU*. *HazardFU* is able to deactivate more units because *UtilFU* activates an extra unit when the last unit is highly utilized, but the processor does not issue instructions to it. For MPGenc, with DVS, *HazardFU* saves 9% energy over *UtilFU*, but Table 5 shows that the savings are not from deactivating more units. Instead, *UtilFU* degrades the IPC more for this application, and DVS exposes this difference.

6.3 Global, Local, and Joint Adaptation

This section compares the global, local, and integrated global and local approaches for architecture adaptation, without and with DVS. For the experiments with local adaptation, we adapted both instruction window size and the number of active functional units. Based on the results in Sections 6.1 and 6.2, we used the *StallIW* and the *HazardFU* algorithms respectively for these adaptations.

We report results for four architectures: (1) the default, non-adaptive *Base*; (2) *Global*, which is *Base* enhanced with global adaptation as described in Sections 2 and 5; (3) *Local*, which is *Base* enhanced with local adaptations as described in Sections 3 and 5; and (4) *Global+Local*, which is *Base* enhanced with the integrated global and local algo-

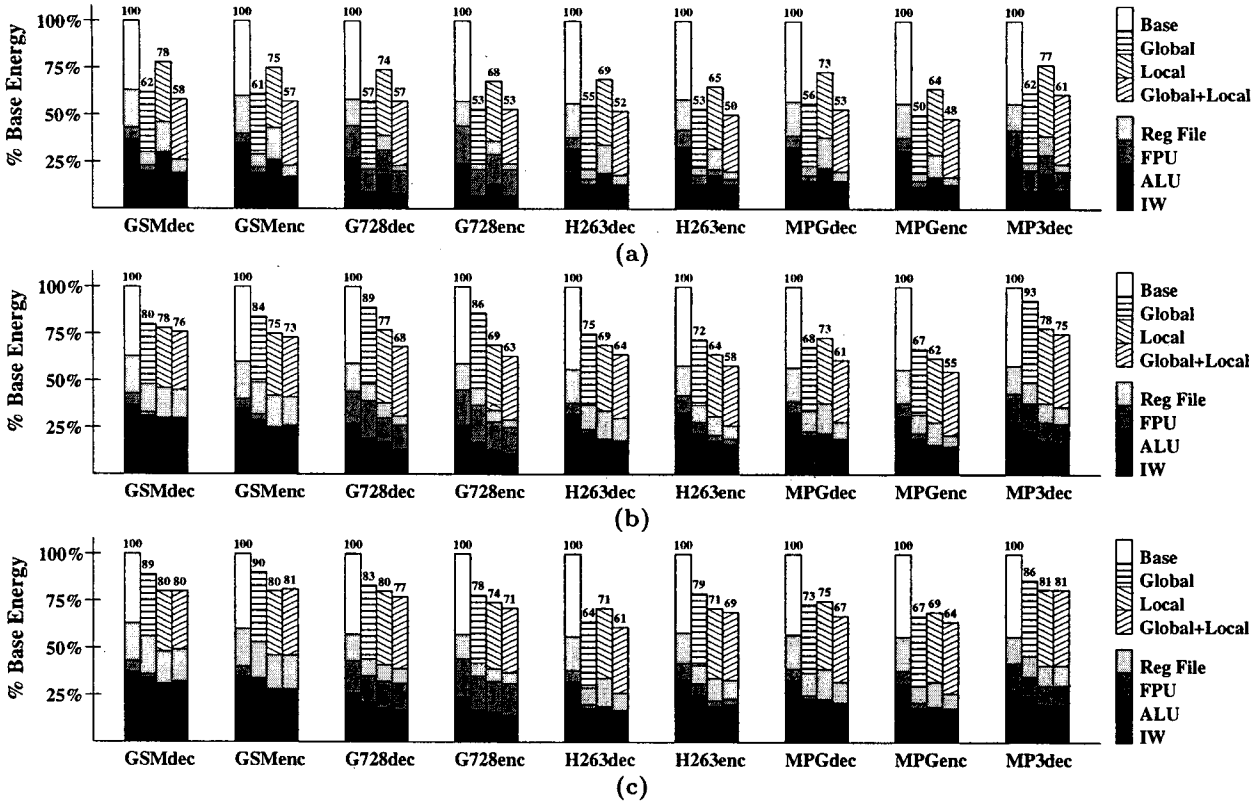

Figure 6: Energy consumption (normalized to *Base*) of processors capable of both global and local adaptation. (a) Without DVS, default deadlines (high slack). (b) Without DVS, tighter deadlines (low slack). (c) With DVS, default deadlines (low slack). Each set of four bars represents *Base*, *Global*, *Local*, and *Global+Local* respectively. The dark parts in each bar show the energy spent in the instruction window, ALUs, FPUs, and the register file.

rithm described in Section 4. For each, we evaluate three combinations of slack and DVS as discussed in Section 5 – no-DVS/default-deadlines (high slack), no-DVS/tighter-deadlines (low slack), and DVS/default-deadlines (low slack).

Figures 6(a), 6(b), and 6(c) show the total energy normalized to *Base* for each application and architecture for the different slack and DVS combinations. Each bar also shows the part of the total energy consumed by the instruction window, ALUs, FPUs, and the register file. Table 6 shows the mean relative savings in energy between different architecture pairs. For detailed analysis, Table 7 also shows the mean number of active functional units and the instruction window size selected by *Global*, *Local*, and *Global+Local* for the high slack case and one low slack case (DVS/default-deadlines). Each application missed less than 5% of its deadlines on all systems. The average IPC degradation for *Local* (i.e., when both local adaptations are combined) without DVS is 7% (maximum 11%).

We divide the results into the case with high slack (Section 6.3.1) and the cases with low slack (Section 6.3.2) followed by a summary and discussion (Section 6.3.3).

6.3.1 Results with High Slack

Our discussion of the high slack case below follows the architecture pairs listed in Table 6.

Global vs. Base: *Global* shows high energy savings over *Base*, as already known from previous work [19].

Savings from	Relative to	High Slack	Low Slack	
		No-DVS	No-DVS	DVS
Global	*Base*	44%	21%	21%
Local	*Base*	29%	29%	24%
Local	*Global*	-26%	8%	3%
Global+Local	*Base*	46%	34%	28%
Global+Local	*Global*	4%	13%	8%
Global+Local	*Local*	24%	5%	5%

Table 6: Mean relative energy savings for different architecture pairs and DVS/slack combinations.

Local vs. Base: *Local* also shows significant savings (29% average). Comparing with the results from Sections 6.1 and 6.2, the benefits from the two local adaptations are almost additive.

Local vs. Global: With high slack, *Global* shows significant energy savings over *Local* for all applications (average of 21%). This is because *Global* has the ability to exploit slack, sacrificing performance to save energy. Given the large slack, *Global* selects the simpler (lower power and IPC) architecture configurations for all applications, as seen in Table 7. *Local*, on the other hand, attempts to maintain performance despite the existence of so much slack. One could potentially design local algorithms that trade off performance for energy savings, but that is beyond the scope of this paper.

Without DVS									
App.	Global			Local			Global+Local		
	I	A	F	I	A	F	I	A	F
GSMdec	16	2.0	1.0	117	4.4	0.0	31	2.0	0.0
GSMenc	32	2.0	1.0	83	4.7	0.0	16	1.8	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	2.4	0.3
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	1.9	0.8
With DVS									
App.	Global			Local			Global+Local		
	I	A	F	I	A	F	I	A	F
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	4.8	0.0
G728dec	88	3.8	1.6	100	3.1	1.3	66	2.5	1.1
G728enc	80	2.5	2.5	81	2.2	1.8	58	2.0	1.7
H263dec	48	4.0	1.0	56	4.4	0.0	40	4.2	0.0
H263enc	96	4.0	2.0	51	3.6	0.5	47	3.4	0.4
MPGdec	64	6.0	1.0	63	4.9	0.0	50	4.9	0.0
MPGenc	48	4.1	1.1	43	3.5	0.1	31	3.3	0.1
MP3dec	128	4.0	2.0	92	3.7	1.1	93	3.7	1.1

Table 7: Mean instruction window size (I), active ALUs (A), and active FPUs (F) selected by *Global*, *Local*, and *Global+Local* with the default deadlines.

Global+Local vs. others: *Global+Local* provides the same or better energy savings than the individual global or local approaches, although most of the benefit comes from global adaptation. The additional benefit of *Global+Local* over *Local* is a significant 24% on average, for the same reason that *Global* alone is superior to *Local* alone. The additional benefit over *Global* is a modest 4% on average (maximum of 7%). *Global+Local* sees some benefit over *Global* because it is able to shut down resources that *Global* cannot, as seen in Table 7. These include the last FP unit and the second to last ALU (all configurations available for global adaptation had at least 2 ALUs). The absolute gains are modest, however, because the simple architectures picked by *Global* offer limited opportunities for further adaptation by *Local*. These gains might be further reduced if *Global* was given an even larger set of configurations from which to choose.

6.3.2 *Results with Low Slack*

With low slack, again, both the pure global and pure local adaptations show high energy savings versus *Base* (>20% average without and with DVS). As is expected, the savings of *Global* are considerably lower than for the high slack case, since there is less slack for it to exploit. *Local* is not sensitive to slack.

Local vs. Global: With low slack, *Local* shows higher energy savings than *Global* in most cases. While the maximum savings is high (20%), the average benefit of *Local* over *Global* is a modest 8% without DVS and a small 3% with DVS. Compared to the high slack case, the lower slack reduces *Global*'s advantage over *Local*. With low slack and no DVS, *Global* must, in many cases, choose a fairly aggressive architecture in order to make the deadline. With DVS, *Global* also frequently picks fairly aggressive architectures (as shown in Table 7). As explained in more detail in [19], aggressive architectures give high enough IPCs for many applications so that it is most energy efficient to choose them and exploit most of the slack through DVS. Since *Local* can exploit intra-frame variability, it is able to deactivate more resources than *Global* for parts of the execution, without losing much performance. As a result, with low slack, *Local* does better than *Global* for all but one application without DVS and for six of the nine applications with DVS.

Global saves more energy than *Local* in some cases for two reasons. First, unlike *Local*, *Global* shuts down part of the register file corresponding to instruction window adaptation. Second, *Global* still exploits slack in some cases. For MPGdec without DVS, sufficient slack remains for *Global* to choose a simpler architecture. For H263dec with DVS, *Global* finds that an architecture less aggressive than *Local*'s average choice is most energy efficient, and exploits some slack through architecture adaptation rather than DVS.

Global+Local vs. others: *Global+Local* again saves almost the same or more energy than either *Global* or *Local* alone because it enjoys the benefits of both types of adaptation. It saves 13% and 8% on average over *Global* without and with DVS respectively (vs. 4% with high slack). As explained earlier, the global algorithm picks fairly aggressive architectures, leaving more room for local adaptation. *Global+Local* saves 5% on average over *Local* for both systems (maximum 16% with no DVS and 14% with DVS). Thus, with low slack, local adaptations provide (modestly) higher benefit over global adaptations in the combined algorithm in all but a few cases.

6.3.3 *Summary and Discussion*

Overall, our proposed combination of global and local architecture adaptation works best across all configurations studied. The relative benefits of global adaptation are higher when the base architecture exhibits higher slack.

With DVS, for our systems and applications, the base architecture exhibited low slack and the integrated algorithm showed (modestly) higher benefits from local adaptations than from global adaptations on average. Nevertheless, for some applications, global adaptation showed higher benefit even with DVS, by deactivating the register file and exploiting some slack through architecture adaptation. For a system that already implements global DVS and local architecture adaptation, adding a global architecture adaptation algorithm (in software) does not introduce much additional hardware complexity. Therefore, with DVS, the integrated global and local algorithm appears a good design choice.

Without DVS, with high slack, for our applications and systems, global architecture adaptation clearly provides most of the benefits. However, when there is little slack in the system, local architecture adaptation becomes more beneficial, outperforming global adaptation (sometimes quite significantly) in all but one case. Given that the amount of slack available is most likely a dynamic quantity (dependent on the total load on the system) and not predictable at design time, again, the integrated global and local architecture adaptation algorithm would be the best implementation choice for systems without DVS.

7. CONCLUSIONS

Hardware adaptation, including DVS and architecture adaptation, has been shown to be effective in saving energy for real-time multimedia applications. Previously, DVS and architecture adaptation have been combined for real-time multimedia applications with a control algorithm operating at a global granularity in both a spatial sense (i.e., all resources adapted together) and a temporal sense (i.e., adaptation occurs once per frame). That algorithm took advantage of computation slack at the end of an application frame to slow the processor down to save energy.

This paper considers spatially and temporally local architecture adaptation and its integration with global adaptation. With local adaptation, a separate algorithm controls each resource (or small group of resources). It attempts to save energy while maintaining performance by deactivating under-utilized components periodicially within a frame. We explore adapting the size of the instruction window and the number of active functional units (and associated instruction issue width).

In our first set of contributions, we evaluate previous local adaptation control algorithms originally proposed for non-real-time applications, and also propose some new local algorithms. We find that local architecture adaptation is effective for real-time multimedia applications without and with DVS. All algorithms evaluated provide modest to significant energy benefits without much reduction in performance, and the new algorithms are marginally better than the best previously proposed algorithms.

In our second set of contributions, we compare pure global, pure local, and integrated global and local architecture adaptation algorithms, both without and with global DVS. The combination of global and local adaptation exploits both computation slack at the frame granularity and variability in resource utilization within a frame. The combination therefore works best across all configurations studied. The source of the majority of the benefits in the combination varies depending on the computation slack and DVS support.

There are several avenues of future work. We would like to explore the remaining part of the design space identified here for adaptation control algorithms. In particular, spatially global but temporally local architecture adaptation is promising. This would exploit both inter-frame and intra-frame execution variability, perhaps obviating the need for two types of control algorithms and integrating local DVS. However, it requires a method to predict execution time and energy impact of adaptations and their mutual interaction over short time intervals (hundreds of cycles). We would also like to explore the interaction among multiple applications running on the system and the real-time scheduler as well as adaptations in other parts of the system.

8. REFERENCES

[1] S. V. Adve et al. The Illinois GRACE Project: Global Resource Adaptation through CoopErating. 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] B. Fields, S. Rubin, and R. Bodík. Focusing Processor Policies via Critical-Path Prediction. In *Proc. of the 28th Annual Intl. Symp. on Comp. Architecture*, 2001.

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

[12] 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.

[13] 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.

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

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

[16] M. C. Huang. *Managing Processor Adaptation for Energy Reduction and Temperature Control.* PhD thesis, University of Illinois at Urbana-Champaign, 2002.

[17] 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.

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

[19] 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.

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

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

[22] 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.

[23] 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.

[24] V. S. Pai, P. Ranganathan, S. V. Adve, and T. Harton. An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors. In *Proc. of the 7th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 1996.

[25] T. Pering, T. Burd, and R. Brodersen. Voltage Scheduling in the lpARM Microprocessor System. In *Proc. of the Intl. Symp. 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] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for Reduced CPU Energy. In *Proc. of the 1st Symp. on Operating Systems Design and Implementation*, 1994.