JOINT PROCESSOR-MEMORY ADAPTATION FOR ENERGY FOR GENERAL-PURPOSE APPLICATIONS

BY

RITU GUPTA

B.Tech, Indian Institute of Technology, Bombay, 2002

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2004

Urbana, Illinois

ABSTRACT

Much work has been done to reduce the energy consumption of the processor or memory using adaptation algorithms for general-purpose systems. This paper develops new adaptation algorithms that combine the benefits of multiple time scales of adaptation and joint processor-memory adaptation to save more energy than previous algorithms for general-purpose applications. Specifically, our final algorithm for joint adaptation of processor and memory has the following attributes that have not previously been available for general-purpose adaptations. First, the algorithm can trade off a specified amount of performance for energy savings. In contrast, previous work on processor adaptation has focused on saving energy without "much" performance loss – our work not only allows more energy savings but also provides a performance guarantee. Second, previous processor adaptation algorithms for general-purpose applications adapt at either a fine or coarse time scale. The new algorithm allows adaptation at both time scales, exploiting both short term and long term variability. Third, previous work has considered processor and memory adaptation separately. Our algorithm is the first to jointly adapt both processor and memory, and shows that such joint adaptation can provide significant energy savings over adapting either component alone.

ACKNOWLEDGMENT

I would like to express my sincere gratitude to my advisor, Sarita Adve, for her invaluable guidance, support, and motivation during the course of this thesis.

I would also like to thank Xiaodong Li, with whom I have had the distinct pleasure of working during the course of this thesis. Special thanks go to the members of the RSIM group with whom I have interacted during this time.

I would also like to thank Prof. Y. Y. Zhou for her valuable ideas and comments in our weekly meetings.

TABLE OF CONTENTS

List of Tables

List of Figures

CHAPTER 1

INTRODUCTION

1.1 Motivation

Energy consumption is an important design-time concern across a large spectrum of computer systems. These systems include mobile systems such as laptops and cellular phones where battery life must be maximized, as well as high-end servers in data centers where the energy and cooling bill must be minimized. The last few years have seen significant research in the use of adaptive architectures to save processor energy [1–10]. More recently, researchers have recognized that memory is also a key consumer of energy. For example, a study shows that for fully configured IBM server systems [11], memory energy could be as high as 150% of processor energy. Commercial memory chips now contain multiple power modes [12] and recent work has proposed control algorithms to adapt between these different modes [13, 14].

To our knowledge, there is no work that considers general-purpose systems where both the processor and memory are adaptive. Traditionally, for general-purpose applications, the focus has been on adapting the processor to reduce energy without any impact on performance. Recent work on memory system adaptation, however, makes the case for trading off some (previously agreed upon) performance to reduce energy. Much work has been done in exploiting such a tradeoff in multimedia applications on mobile systems where the applications lend themselves easily to different levels of quality of service (with different performance demands) and battery life is a predominant constraint. However, such tradeoffs are also becoming reasonable in the context of data center environments where customers may sign service-level agreements depending on desired performance and attendant energy costs. Given an environment where some known amount of

performance slowdown is acceptable, it is unclear how to exploit this jointly through processor and memory adaptations.

This thesis represents the first work that considers reducing energy consumption in the processor and memory subsystems *together* for general-purpose applications. The focus of our work is on algorithms to control the different adaptation modes provided by the processor and memory, with the goal of minimizing energy while incurring no more than a specified performance slowdown. This work requires advances on two fronts – in processor adaptation algorithms and algorithms for joint adaptation of multiple hardware components (processor and memory in our case).

1.2 Challenges in Processor Adaptation

As mentioned, most current work on processor adaptation for general-purpose applications aims to reduce energy without "much" loss in performance, but without bounding this loss. In most of this work, the control algorithms are developed using heuristics that are painstakingly hand tuned to avoid performance loss. Even so, these algorithms may incur unpredictable performance losses for situations outside the training set of the tuning. In contrast, we seek to develop control algorithms that can exploit a known amount of performance slowdown to further reduce energy, while providing a guarantee of not exceeding this specified slowdown. The acceptable slowdown may be different for different applications and system environments. The heuristics-driven hand tuning approach for all such possible slowdowns and environments is not practical. Instead, a more algorithmic approach that can predict the energy and performance impact of different adaptations is required. While there have been a few studies that have considered incurring a known slowdown for saving energy [15, 16], we are not aware of other work that provides a performance guarantee for processor adaptations.

To solve this problem, we bring together concepts developed in several recent works in different contexts. Sherwood et al. [17] have shown that at a large scale (millions of instructions), programs repeat their behavior, and it is possible to predict the occurrence and performance characteristics of these repeated *phases*.

Once we establish the presence of such predictable phase boundaries, we can leverage energy saving techniques by Hughes et al. [18,19] in the context of multimedia applications. That work ex-

ploits the natural frame boundaries of multimedia applications and prior results that show that the performance and energy for the next frame are predictable. Extending the ideas from Sherwood et al. [17] and Hughes et al. [19], we develop an algorithm for general-purpose applications that adapts at phase boundaries to choose the lowest energy configuration that stays within the specified performance target. The guarantee on performance is ensured even in the presence of mispredicted phase identifications – to achieve this guarantee, we adapt some ideas from the performance guarantee algorithm previously proposed for memory adaptation [14].

In summary, our final processor adaptation control algorithm unifies ideas from several prior studies. The result is an algorithm with a combination of significant attributes that are not all available in any prior algorithm for general-purpose applications.

1.3 Challenges in Joint Processor-Memory Adaptation

Given that both the processor and memory adaptations seek to minimize energy while incurring a targeted performance slowdown, we need to ensure that these adaptations are adequately coupled and do not conflict with each other. For example, a memory adaptation that slows down the memory's response to the processor's request has an impact on the optimal energy-efficient configuration of the processor. Similarly, the configuration of the processor has an impact on the rate at which memory sees requests and the resultant optimal memory configuration. It is unclear how to couple the processor and memory adaptation algorithms so that we can achieve configurations that are energy-optimal in a system-wide sense and respect the overall performance target. Furthermore, coupling the algorithms too tightly for the perfect optimal incurs the danger of being too inefficient – a completely coupled algorithm would need to consider the cross-product of the configuration space possible for the processor and memory, which is too large for our case.

We show that while there is a need to perform the processor and memory adaptation in a cooperative way, a tight coupling is not required. We demonstrate that the key to successful cooperation is to determine an optimal distribution of the performance slowdown that should be exploited by the processor and the memory system. Once such a distribution is determined, the processor and memory adaptations can take place independently of each other (while respecting their own slowdown targets). We show that it suffices to determine the distribution at phase

boundaries, and that it can be determined efficiently. Specifically, the algorithm does not need to examine the configuration space cross-product and so does not suffer from the configuration space explosion.

1.4 Summary of Contributions and Key Findings

Overall, our new joint processor-memory adaptation algorithm advances the state-of-the-art in the following ways:

- *Energy-performance tradeoff.* Our new algorithm allows processor adaptations to exploit a targeted performance slowdown to minimize energy. Previously, all processor adaptation algorithms for general-purpose applications have aimed to provide energy savings without "much" loss in performance.

- *Processor adaptation at multiple time scales.* The new algorithm invokes processor adaptation at multiple time scales, exploiting the benefits of local and global variability. Previous algorithms for general-purpose applications mostly adapted at the local time-scale, with the exception of one preliminary work that adapts at only the global time-scale.

- *Joint processor-memory adaptation.* The new algorithm is the first to perform both processor and memory adaptation in a cooperative way.

- *Performance guarantee.* The algorithm provides a performance guarantee even in the face of mispredictions of phase identifications. We are not aware of any previous processor adaptation algorithms that provide such a guarantee.

Our simulation-based experiments on seven SPEC benchmarks show that our new algorithm is effective on all of the above counts. Specifically, our algorithm is able to maintain the performance target for all cases. Regarding processor adaptation, we show that combining global and local adaptation is best. In particular, it saves up to 25% (average 12%) more processor energy than global adaptation alone and up to 30% (average 11%) more than local adaptation alone. Regarding joint processor-memory adaptation, we show that it is beneficial to adapt both processor and memory. Joint adaptation provides a savings of about 30% on average over processor adaptation

alone and over 40% on average over memory adaptation alone. Finally, our results show that this joint processor-memory adaptation must be done in a cooperative way, with an intelligent and application-specific choice of slowdown distribution between processor and memory.

1.5 Organization of the Thesis

This thesis is organized as follows. Chapter 2 describes the processor adaptation algorithms and presents the results for processor only adaptations. Chapter 3 presents our joint processor and memory adaptation algorithm. Chapter 4 presents related work on processor and memory adaptations not already discussed in Chapters 2 and 3. Chapter 5 summarizes our findings and presents future work.

CHAPTER 2

PROCESSOR ADAPTATION ALGORITHMS

Section 2.1 describes the most closely related processor architecture adaptation algorithms proposed for multimedia applications. (Chapter 4 provides a complete comparison with related work.) Section 2.2 describes the new algorithms derived for general-purpose applications. Section 2.3 describes the experimental setup. Section 2.4 presents energy savings due to the different processor adaptation algorithms.

2.1 Previous Algorithms for Multimedia Applications

2.1.1 Global algorithm

Multimedia applications are typically frame-based, where the execution of each frame must meet a certain *deadline*. Previous work showed that for several multimedia applications studied, for a given architecture configuration, the average IPC and average power for a frame stay roughly the same from frame to frame [18, 19]. This is because the nature of the work done stays roughly the same. The amount of work, however, differs from frame to frame, resulting in differing instruction counts for different frames. Nevertheless, since the amount of work changes slowly across frames, the instruction count is highly correlated with the previous few frames and so is easy to predict [18].

Using the above observations, Hughes et al. [19] proposed the following algorithm for architecture adaptation at the frame granularity for multimedia applications. The goal for the algorithm is to find, for each frame, the lowest energy architecture configuration that will execute the frame

within the time allocated (referred to as the deadline). The algorithm consists of two (on-line) phases – a profiling phase and an adaptation phase.

The profiling phase runs successive frames using the different architecture configurations available (one configuration for an entire frame). For each architecture configuration, it records its frame's average instructions per cycle (IPC) and power. This information also gives the average frame energy per instruction (EPI) for the architecture (which also remains roughly constant across frames, for a given architecture). For each architecture configuration, the algorithm then uses the measured IPC and EPI to determine the maximum number of instructions (I_{max}) that can be executed within the assigned deadline. The algorithm then constructs a table with an entry for each architecture configuration containing I_{max}, and sorted in order of increasing EPI.

In the adaptation phase, before executing a frame, the algorithm predicts the number of instructions to be executed by that frame (using a simple predictor based on instruction count of the last few frames). It then searches the above table to find the first hardware configuration with $I_{max} > \textit{predicted instructions}$ – this configuration is expected to execute the next frame in the assigned deadline with the lowest EPI and is used for the next frame.

2.1.2 Local algorithms

The Local algorithms adapt frequently (e.g., every few hundred cycles, referred to as the *local interval*) in response to variability in the usage of the targeted microarchitectural resources. Since it is difficult to predict the performance impact of adaptations at this fine granularity, these algorithms attempt to reduce energy without "much" impact on performance by shutting down resources that are predicted to not contribute to performance. Sasanka et al. studied local adaptation of the instruction window size and active functional units (and the consequent issue width) for multimedia applications, and found the following algorithms to work the best [19]. For the instruction window, they proposed a new algorithm that combines two heuristics: one each for increasing and decreasing the instruction window size. The increase heuristic is based on estimating the loss in IPC from having part of the instruction window deactivated over the interval. If the IPC loss is greater than some threshold, the window size is increased. The decrease heuristic is from [7]. It counts the number of instructions issued from the youngest part of the window over the interval – if the

number of committed instructions is smaller than a threshold, the window size size is decreased.

For controlling the number of active functional units, the algorithm again uses two heuristics: one each for increasing or decreasing the number of functional units, based on the work in [20]. The algorithm for decreasing the units measures the mean functional unit utilization over an interval – if it is less than a threshold, then it decreases the number of functional units. The algorithm for increasing the units tracks the total number of structural hazards for each type of unit seen by all instructions within an interval. If the total exceeds a threshold before the end of the period, the number of active functional units is increased.

2.1.3 Joint Global+Local algorithm

Sasanka et al. combined the above two classes of algorithms to get the benefits of both for multimedia applications [21]. The joint Global+Local algorithm can be viewed as a Global algorithm but where each architecture configuration also invokes Local adaptations. Thus, the algorithm comprises an initial profiling phase, where each architecture is profiled over the period of a frame. During this time, the architecture also invokes local adaptations to react to variability during the frame to further reduce resources without (ideally) further degrading execution time. The Local adaptations in this case are not allowed to increase resources beyond those provided by the architecture configuration being profiled. After all profiling is done, an EPI-sorted I_{max} table is constructed as before. Subsequently, the adaptation phase begins and at the start of each frame, the lowest EPI configuration with the appropriate I_{max} is selected, similar to the Global algorithm.

2.2 New Processor Algorithms for General-Purpose Applications

2.2.1 Global algorithm

The multimedia global processor adaptation algorithm adapts at the frame granularity and saves energy by slowing down the frame execution just enough to meet the deadline. General-purpose applications are different in that there is neither a direct notion of a frame nor a deadline. However, recent work has shown that many general-purpose programs execute as a series of repeated phases – each phase may be fairly different from the others, but repeats over the course of the execution

with similar performance and power metrics averaged across the phase [17, 22, 23]. We can use the notion of a *phase* as the granularity for adaptation, analogous to a multimedia frame. Instead of using a deadline to determine the allowed slowdown, we assume that the user specifies an allowable percentage slowdown or *slack*, relative to the base performance without any adaptations (e.g., a 10% slack).

We use the technique described in [17] to track and classify phases after every 10 million instructions. This technique is based upon code execution frequencies and is independent of the architecture configuration used to run the phase. At the end of each group of 10M instructions (referred to as a *phase interval*), the phase classification technique assigns a unique phaseID corresponding to the tracked phase. We refer to a phase interval classified as a given phase as an *occurrence* of that phase. The algorithm also requires a phase predictor to predict the phaseID of the next phase interval.

For simplicity, we first describe the algorithm assuming a perfect phase predictor. The goal of the algorithm is to slow down each occurrence of a phase by the specified slack. (Note that such an equal distribution of slack across all phases may not be optimal, but other distributions are more complex.) Given the correspondence between a phase and a frame, we adapt the multimedia global algorithm as follows. Again, we have a *profiling part* where the initial occurrences of each phase are run with different architecture configurations (one configuration for an entire phase occurrence). These profiles give, for each architecture configuration and phaseID, the execution time and energy taken by the phase. This time and energy is predicted to be the same across all occurrences of the phase. Unlike the multimedia algorithm, all occurrences of a phase have the same instruction count; therefore, we can directly collect the execution time (instead of IPC) and energy (instead of EPI). After all the profiles are collected for a given phase, the algorithm can simply determine the architecture configuration with the lowest energy such that its execution time is within the targeted slack (i.e., the slowdown relative to the base architecture is less than the user-specified constraint). Subsequent occurrences of the phase are now run at this chosen architecture configuration. The total number of intervals used for profiling with this algorithm equals *Number of phases* $\times$ *Number of architectural configurations*.

The above algorithm is simple, but assumes that the phaseID for the next phase interval can

be predicted perfectly. However, this is not always the case. A phase misprediction could result in choosing an architecture that violates the performance constraint by using up too much slack. To accommodate this case, we modify the algorithm to track the slack used in each phase interval. If too much slack is used, then different configurations are chosen in subsequent intervals. These configurations use up less than the user-specified slack and are used until the previous over-use is compensated for. To achieve this, at the end of the profiling phase for a given phaseID, the algorithm builds a table for that phaseID, with an entry corresponding to each architecture configuration and sorted in increasing order of energy (analogous to the multimedia algorithm). The entry stores the execution time for the architecture for that phaseID.

Now consider the execution of a phase interval i with a certain phase P, which may have been predicted incorrectly and run with an inappropriate architecture $Arch$. We use the following terms:

- T^P_{Arch} : Execution time for P with the architecture $Arch$ as measured.
- T^P_{base} : Execution time for phase P with the base architecture (as determined from the above table).
- $Slack$: Target user slack (specified as a fraction).
- $UsedSlack_i$: Absolute slack (in terms of time) used in interval i
  $UsedSlack_i = T^P_{Arch} - T^P_{base}$.
- $RemainingSlack_i$: Unused slack (in terms of time) at the end of interval i
  $RemainingSlack_i$ = (Absolute slack available at start of interval i) - $UsedSlack_i$
  $= T^P_{base} * (Slack/100) + RemainingSlack_{i-1} - (T^P_{Arch} - T^P_{base})$.

If the remaining slack at the end of interval i is negative, then the algorithm calculates a new desired execution time for the next interval $i+1$ with phaseID P' as $T^{P'}_{base} * (1 + Slack) + RemainingSlack_i$, where $T^{P'}_{base}$ is the execution time with the base architecture for the predicted phase for interval $i+1$.

The algorithm looks up the table for the predicted phase for interval $i+1$ to determine the first architecture configuration in the table that has an execution time value less than the above. This configuration is then chosen for the next phase interval. Assuming there are enough phase intervals

remaining to execute, any over-use of slack is compensated for, and a performance guarantee is maintained.

We use three different phase predictors in our experiments: (1) perfect predictor; (2) a predictor which predicts the phase of the next interval to be the same as the current interval, referred to as *Simple* predictor; and the (3) RLE Markov Predictor proposed in [17].

2.2.2 Local algorithms

The Global algorithm adapts only at large-scale phase boundaries and it can only exploit the large scale variability among phases, referred to as interphase variability. Ideally we would like to adapt frequently in response to all variations in microarchitectural resources within a phase. The local adaptation algorithms we consider are derived from the multimedia algorithms described earlier. Specifically, we examine two local adaptations: changing the size of *instruction window* and changing the number of *Active Functional Units* which also changes the *Issue Width*.

Previous local algorithms present the following challenges. First, at the intraphase granularity, it is difficult to predict the impact of adaptations on performance. The local algorithms considered in the literature use a variety of heuristics and take a long time to tune the thresholds. For the multimedia algorithms, the authors state that it took weeks of hand tuning. Using the same parameters as those for the multimedia algorithms, we found that the processor performance degradation for some phases was as high as 60%. This highly application-dependent nature of local heuristics-based algorithms makes them less desirable. Nevertheless, we study them here since much of the past literature on processor adaptation is focused on such algorithms, and we would like to explore the potential for exploiting variability at multiple time scales in general-purpose applications.

The second challenge for local algorithms arises because of our goal for providing cooperative adaptation for both processor and memory as in Chapter 3. As mentioned earlier, for ideal cooperative adaptation, we need to decouple the impact of local processor adaptations from those of memory. However, directly using some of the heuristics in Sasanka et al. makes this difficult. Specifically, memory adaptations may slow down memory (i.e., increase memory response time) to provide energy savings [14]. This could result in reduced utilization of processor resources. Since some of the heuristics of the local algorithms are directly related to utilization of microarchitectural resources, this can lead to decreasing the resources. This in turn slows down the processor which

can result in further reduced memory activity. This can cause the memory to switch further to an even lower power mode, further reducing memory response time. Hence, an unending loop is formed, which could lead to a huge performance degradation. To prevent the above performance degradation loop, we need to modify the algorithm so that the local processor adaptations are not sensitive to memory latency and its impact on processor resource utilization.

We performed some, but not much, tuning of the local adaptation algorithms to address both of the above challenges. With the original algorithms, the adaptation interval was 256 *cycles*. We found that using an interval of 100 branch *instructions* instead worked much better. Our inspiration for this change came from the phase classification method used in the global algorithm, where the branches and the number of instructions between them form the signature for a phase. This change limited the performance degradation to within about 15% for all phases of all applications. Further, it also provided sufficient decoupling for the memory and processor algorithms – changing the interval from a cycle-based size to an instruction-based size made the thresholds less sensitive to resource utilization.

2.2.3 Global+Local algorithm

The Global+Local algorithm seeks to exploit both interphase and intraphase variability by combining the global and local algorithms. This combination is very similar to that in the multimedia work [21]. The Global algorithm performs adaptations at the granularity of a phase interval and the local algorithms perform adaptations at the granularity of an interval (which in our case is 100 graduating branch instructions).

The Global part performs the profiling and adaptation phases as before, but additionally also performs local adaptation throughout. Ideally, the global part would now see a lower average energy for each architecture with little change in the execution time (versus without local adaptations). As before, the algorithm chooses the lowest energy architectural configuration which can meet the execution time requirement governed by the available slack. The Global part compensates for any performance degradations due to the Local algorithms and helps to keep the execution time within user defined constraints. The Local parts are independent of the Global part except for the fact that Global establishes a maximum configuration for each resource for the Local algorithms. Any Local algorithm operates within these maximum resource constraints.

2.2.4 Overheads

The Global and Global+Local algorithms have a phase detection overhead, profiling overhead, and adaptation time overhead.

For phase detection we use an architecture similar to the one used in [17]. The space and time overhead incurred due to the phase detector and classifier has already been discussed in [17]. The phase detection architecture consists of an accumulator with 32 entries. The branch PC is used to hash into the accumulator table and the corresponding counter entry is incremented by the total number of instructions executed in between the two successive branches. This is done every 10M instructions. We also need to save the signatures of all phases for phase classification. This is done by using a history table which stores the signatures. The hardware structures used in the accumulator and the history table lead to space overhead. Unlike [17], we choose to keep the signatures for all phases. For benchmark *ammp*, 35 phases contribute to 98% of the program run. Phase detection and classification also has an energy overhead. The energy overhead includes keeping a count of number of instructions executed between branches. This is done for every instruction. The other energy overheads are for each phase interval. These include a table lookup for hashing branch PC and incrementing accumulator entries. Our experiments do not account for these overheads but they will be negligible in comparison to the energy savings.

The adaptation part also incurs some overhead. Here, the global algorithm is invoked at the beginning of each phase interval. It needs to predict the phaseID of the next phase interval and perform a table lookup to find the lowest-energy configuration which meets the performance guarantee. The prediction overhead is O(1) in case of both simple and RLE Markov predictors. The table lookup will require N_{config} number of comparisons in the worst case, where N_{config} is the total number of architectural configurations. These overheads are incurred every phase interval (10 million instructions) and are negligible.

The local adaptations suffer from hardware overheads. These overheads include the logic for calculating tags, logic for calculating the maximum possible overlap at the end of the period, a shifter to compute IPC, and some counters. The local algorithms for changing the number of functional units require additional hardware circuitry to power on and off the functional units. The overheads associated with Local algorithms are higher since they are invoked after small intervals

(100 branch instructions in our work). We model the energy overheads due to these hardware changes. This has been discussed in more detail in [21].

For all processor algorithms, namely, Global, Local, and Global+Local, we need to calculate the actual execution time in order to calculate the incurred slack. This is done every 10M instructions and can be done in hardware.

2.3 Methodology

This section describes the general-purpose processor architecture that we study, along with the applications and inputs and our simulation infrastructure.

2.3.1 Workload and architectures studied

Our workload consists of seven SPEC2000 CPU benchmarks which are summarized in Table 2.1. Of these seven benchmarks, *mcf*, *gzip*, and *twolf* are SPECint and *ammp*, *equake*, *art*, and *mesa* are the SPECfp CPU benchmarks. This workload consists of applications which are either memory intensive or computation intensive or both. All these applications have a small initialization phase which is not representative of the complete program behavior [24]. In our simulation, we fast forward each benchmark to skip the initialization phase based on the numbers from Sherwood et al. [24].

The base architecture consists of an out-of-order superscalar processor, and its characteristics are described in Table 2.2. Several variations of the base architectures are also studied, and are described in the corresponding sections. We use the RSIM simulator [25] for most of our experimental evaluation. RSIM is a user-level execution-driven simulator that models the processor and memory in detail including the contention for all resources. The memory model used in RSIM is the RDRAM memory model. There are a total of eight 256 MB RDRAM chips in our memory configuration. Operating system and I/O functionality is emulated, not simulated. The applications in the benchmark were compiled using the SPARC SC4.2 compiler.

The base processor studied is similar to MIPS R10000. In our simulations we assume a centralized instruction window with a unified reorder buffer and issue queue but a separate physical register file.

The Wattch tool [26] has been integrated with RSIM for energy measurements. In our simulations, we model this tool with parameters of 0.18-μm technology. To model power for architectural adaptation, we generate separate power models for each possible architecture, as if each were a separate processor. We assume that when an architecture other than the base is selected, the components not available in that architecture are powered down, consuming no energy.

In the base processor we assume a centralized instruction window with a unified reorder buffer and issue queue but a separate physical register file. We allow adaptations of issue width, the instruction window, and the number of functional units as in [19]. All of these have significant impact on the execution time and energy dissipation in the applications we are considering.

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. This is straightforward with global adaptation. We deactivate one integer physical register and one floating point physical register with each deactivated instruction window entry with global adaptation. With local adaptation, we do not change the register file size since it would be too much overhead.

Experiments with functional unit adaptations assume that issue width is equal to the sum of the functional units and hence changes with the number of 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 components of all the processor configurations (adaptive and nonadaptive). 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-art, we also gate the wake-up logic for empty and ready entries in the instruction window as proposed in [7]. We assume that the resources 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 gating).

We use Local algorithms as described in Section 2.2.2. The threshold values used for various heuristics are given in Table 2.3. For Global adaptations described in Section 2.2.1 and Global+Local algorithm described in Section 2.2.3, we evaluate energy savings for user slack of 5%, 10%, and 20% over base execution time. We profile all possible combinations of the following configurations (54 total): instruction window $\in$ {128,96,64,48,32,16}, number of ALUs $\in$ {6,4,2} and number of FPUs $\in$ {4,2,1}. This set of 54 architectural configurations is denoted as G_{54}.

In order to reduce the profiling effort we also run simulations where we reduce the number of the architectural configurations from G_{54} to a much smaller set. Since SPECInt and SPECfp benchmarks behave differently in terms of the requirements of functional units, we have two different reduced sets of architectural configurations. These sets of architectural configurations are referred to as G_i and G_f for SPECint and SPECfp benchmarks, respectively. The G_i and G_f configurations are given in Tables 2.4 and 2.5, respectively.

The rationale behind obtaining the reduced set of architectural configurations G_i and G_f is as follows:

- SPECInt benchmarks only have integer operations so we do not need more than one FP unit. Hence, we do not consider architectural configurations with 2 or 4 FPUs in the G_i set.
- For SPECfp benchmarks, the dominant form of computation is due to floating point operations. Hence, we do not consider architectural configurations with 6 ALUs in G_f set.
- The simulation results show that adjacent instruction window sizes, say 32 and 16, with the same number of functional units are quite close to each other in terms of energy consumed. Hence, we first consider the instruction window set {128,64,32}. Since the base processor is a dynamic, out-of-order execution processor, we would like to have a larger instruction window if more functional units are available. This helps in increasing or decreasing the instruction window size by 16 for each configuration depending on the availability of functional units.

Since the applications take an extremely long time to run, we perform the following approximation. We first collect a trace of the phase behavior of all the applications for their entire length. For each experiment, we simulate the application long enough to ensure that we collect the necessary

profiling data for each phase, and subsequently, each phase occurs at least ten times for adaptation[1] (This part alone takes a month for *ammp.*). The execution times and energy across these last 10 occurrences of each phase are averaged to give the phase execution time and energy. This average value is then fed into the phase trace initially collected to determine the energy and execution time of the entire application for that experiment. In the results reported, energy is reported for all phases except *minor* phases. We do not include the energy consumed by the program when it is executing in a *minor* phase. Minor phases contribute towards less than 2% of program run for most of the applications except *gzip* where they contribute towards 20% of the program run. For the application *gzip*, the minor phases constitute 20% of the program run for the input, webserver log, which we use in our simulations. For other inputs, minor phases contribute to about 5% of program run.

2.4 Results

We evaluate six different processor architectures :

- **Base:** This is the default nonadaptive architecture.
- **Global:** This is the Base architecture enhanced with global adaptations using the complete set of architectural configuration, i.e., (G_{54}) as described in the previous section.
- **Reduced Global:** This is the Base architecture enhanced with global adaptations using the reduced set of architectural configuration, i.e., (G_i/G_f) as described in the methodology section.
- **Local:** The Base architecture enhanced with local adaptations which adapt the instruction window and the functional units as described in the previous section.
- **Global+Local**: This is the Base architecture enhanced with joint Global+Local adaptations where the processor architectural configuration set is G_{54}.
- **Reduced Global+Local**: This is the Base architecture enhanced with joint Global+Local adaptations where the processor architectural configuration set is G_{54}.

[1]If the same phase occurs too often through this run, it is fast forwarded, while ensuring that the simulation is adequately warmed up for the next phase that needs to be measured.

Section 2.4.1 presents the energy consumption. Section 2.4.2 presents the architectural configurations selected for the five longest occurring phases of each application by the algorithm. Section 2.4.3 presents the performance degradation incurred by different applications. Section 2.4.4 presents an overall analysis. Section 2.4.5 presents a comparison between different processor algorithms. Section 2.4.6 presents the comparison between the architectural sets, G_{54} and G_i/G_f.

2.4.1 Energy consumption

Figures 2.1 and 2.2 show the processor energy consumption with perfect and simple predictors, respectively, for systems with processor architectures *Global*, *Global+Local*, and *Local* normalized to *base*. The RLE Markov predictor provides only a small improvement in energy savings compared to the simple predictor, but is much more complex. We therefore omit detailed results for that predictor.

Each bar in the figure also shows the (normalized) energy distribution among the different processor components – instruction window (IW), ALU, FPU, and the rest of the processor. Table 2.6 summarizes the data in Figures 2.1 and 2.2– for each algorithm pair, it shows the average (overall applications), minimum, and maximum energy savings for each user slack and predictor type. When comparing with Local, we do not include points where Local fails to meet the targeted performance slack (Global and Global+Local always meet the target).

2.4.2 Configurations used

Tables 2.7 and 2.8 show the architectural configuration chosen by the *Global* and *Global + Local* architectures, respectively.

As can be seen from the tables, as we increase user slack from 5% to 20%, both Global and Global+Local algorithms tend to choose much simpler architectural configuration. However, two exceptions are *mcf* and *art*. These applications are memory intensive and very small user slacks are enough to adapt to simple optimal configurations.

Table 2.9 presents the mean instruction window size and the average number of functional units chosen by the local adaptation algorithms for the top five phases of each application. As can be seen from the table, the local adaptations are able to successfully turn off the floating point unit for integer benchmarks, *gzip*, *mcf*, and *twolf*. Local algorithms are invoked at the granularity of an

interval. Table 2.9 also gives the basic interval information for an interval at which local algorithms are invoked. We present the total number of instructions and processor cycles in each interval. We also present the standard deviation of the IPC of the interval as a percentage of the IPC of the phase interval.

2.4.3 Performance degradation

Table 2.10 shows the performance degradation for each case for both the perfect(P) and simple(S) predictors

2.4.4 Overall analysis

The data in Sections 2.4.1 and 2.4.3 illustrates the following high-level results.

- All of the processor adaptation algorithms give significant energy savings over the base case for all slack values.
- For each benchmark, slack value, and predictor, the performance degradation for Global and Global+Local is within the allowed value. For Local, there are some cases with 5% slack where the performance degradation exceeds the target. This is to be expected since Local does not use any tight performance guarantee algorithm. (The reason the degradation is not much worse is because we hand-tuned the algorithm.) When comparing Local with the other algorithms below, we do not consider the points where its performance degradation exceeds the user specified slack since these are unacceptable points.
- For Global and Global+Local, increasing the slack increases the energy savings for some, but not all, benchmarks.
- Comparing the different algorithms, we find that for each slack value and predictor, the combined Global+Local algorithm uses the least energy or close to the least energy (within 6% of the best). Global+Local provides significant benefits over Global alone and Local alone for several applications. These results are consistent with those previously reported for multimedia applications [19].

- Comparing the two predictors, we find that the simple predictor performs remarkably close to the perfect predictor for most of the applications except *art*. In terms of absolute savings, Global+Local is within 4% for *mcf, mesa, gzip, ammp, equake*, and *twolf* .

2.4.5 Detailed analysis

Global vs. Base: Global always gives substantial energy savings over the base. The average energy savings increase from 42% to 50% as we increase the slack from 5% to 20%. For multimedia applications, the average energy savings of global algorithms over base are 44% and 21% for high and low user slacks, repectively. For some benchmarks, increasing slack does not increase energy savings substantially. For example, *mcf* and *art* are memory intensive benchmarks with a very low IPC and use lower configurations even with the 5% slack case; increasing slack does not change the architectural configuration. On the other hand, for other applications, increasing slack does buy significant energy benefits. For example, for *equake*, increasing slack from 10% to 20% improves the energy savings over base from 45% to 52%. This indicates that the trade-off between energy and slack should be made in an application specific way, and is an important direction for future work. Overall, we find that for most applications, Global is able to exploit the majority of the available slack (Table 2.10).

Finally, we note that the architectural configurations chosen by Global do differ among different phases for some, though not all, applications for 20% slack. This indicates it is beneficial to exploit phase-level variability and perform interphase adaptation (as opposed to choosing a single configuration for the entire application).

Local vs. Base: Local gives substantial energy savings over the base. The average savings is about 43%, ranging from about 23% for *ammp* to 69% for *mcf*. The energy savings obtained for the multimedia applications using the local algorithms was about 29% [19]. Table 2.9 shows that there is significant variability in IPC across local adaptation intervals within a given phase, indicating the potential for local adaptation even over the relatively large intervals that we chose. Nevertheless, as can be seen from Table 2.9, for some applications, the local algorithms are not able to reduce the Instruction window size (e.g., *equake*).

Global vs. Local: On average, Global uses less energy than Local for higher slack like 10% and 20%. Global can exploit the extra slack to slow the processor down by choosing simpler

configurations whereas Local seeks to maintain base performance. For low slack, Global does not have this advantage and Local often does better because it can exploit short-term variability and, for some applications, it has higher program coverage as discussed in Section 2.3 (e.g., *gzip*). In some cases, Local also does better at higher slack for two reasons in addition to its higher coverage. First, Local has the ability to shut off all floating point units while Global cannot do so (e.g., for *mcf* and *gzip*). Second, for cases such as *mcf*, as discussed above, Global cannot exploit further slack and so loses its advantage over Local.

Global+Local vs. Others: Global+Local saves more energy or is within 6% of the best algorithm for all cases. This is because the combined algorithm gets the advantages of both Global and Local discussed above. Compared to both Global alone and Local alone, Global+Local sees significant benefits for several applications; e.g., 7% more savings than Global alone for *gzip* and 22% more savings than Local alone for *equake*.

In some cases with 5% user slack, Global+Local does worse (within 6%) than Global (e.g., *gzip*,*mesa* and *equake*). In these cases, for some phases, the Local adaptations cause more than 5% performance degradation and so the Global+Local algorithm is forced to use the base configuration. The Global algorithm in contrast can judiciously use slightly simpler configurations which use less than 5% slack (the Global+Local algorithm does not have a choice of these configurations because its profiling happens with Local adaptations always turned on).

2.4.6 Results for reduced architectural configuration set

In this section, we present the results for profiling with reduced architectural set G_i/G_f. Since we are interested only in the relative energy savings we present results only for perfect predictor.

Figure 2.3 shows the total processor energy consumption for systems with processor architecture *Global* and *Reduced Global* normalized to *Base* processor energy for an perfect predictor. Figure 2.4 shows the total processor energy consumption for systems with processor architectures *Global+Local* and *Reduced Global+Local* normalized to *Base* for a system with perfect predictor. Table 2.11 presents the energy savings averaged over all applications for *Reduced Global* and *Reduced Global+Local* architectures. As can be seen from the table, the *Reduced Global* processor architecture gives on an average 14% less energy savings than the *Global* architecture. The energy savings gap is reduced for *Reduced Global+Local* architecture where *Reduced Global+Local* gives on

an average 4% less energy savings than *Reduced Global+Local*. This is due to the ability of Local algorithms to exploit the variability in the usage of microarchitectural resources.

Tables 2.12 and 2.13 show the average instruction window size and the mean number of functional units used by the five longest phases of each application by the Global and Global+Local algorithms for G_i/G_f architectural set, respectively. In case of G_i/G_f set of architectural configurations, we do get substantial energy savings with respect to *Base* processor architecture; however, we do not see much variation in the architectural configurations as we increase the user slack from 5% to 20%. This is because the spatial granularity is too coarse.

2.5 Figures and Tables

Table 2.1 Workload description.

Benchmarks	
SPECInt	**SPECfp**
gzip	ammp
mcf	equake
twolf	art
	mesa

Table 2.2 Base system parameters.

Base Processor Parameters	
Processor speed	1 GHz
Fetch/Retire rate	6 per cycle
Functional units	6 Int, 4 FP, 2 Add. gen.
Integer FU latencies	1/7/12 add/multiply/ divide(pipelines)
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	2 KB bimodal agree, 32 entry RAS
Base Memory Hierarchy Parameters	
L1 (Data)	64 KB, 2-way associative, 64 line, 2 ports, 12 MSHRs
L2 (Instr)	32 KB, 2-way associative
L2(Unified)	1 MB, 4-way associative, 64 B line, 1 port, 12 MSHRs
Main Memory	16 B/cycle, 4-way interleaved
Base Contentionless Memory Latencies	
L1(Data) hit time (on-chip)	2 cycles
L2 hit time (off-chip)	20 cycles

Table 2.3 The thresholds used for Local algorithms.

Threshold	Description	Thresholds Used
iw1	Maximum stall cycles from shrunken instruction window	20
iw2	Maximum instructions issued from youngest instruction window segment	10
alu1	Minimum cycles all ALUs utilized	250
alu2	Maximum number of ALU issue hazards	50
fpu1	Minimum cycles all FPUs utilized	250
fpu2	Maximum number of FPU issue hazards	100

Table 2.4 Reduced set of architectural configurations for SPECInt: Size of instruction window (IW), number of ALUs (A), and number of FPUs (F).

G_{int}			
Config ID	IW	A	F
0	128	6	4
1	128	6	1
5	128	4	1
11	96	2	1
17	96	2	1
23	64	4	1
29	48	6	1
35	48	2	1
41	32	4	1
44	32	2	1

Table 2.5 Reduced set of architectural configurations for SPECfp: Size of instruction window (IW), number of ALUs (A), and number of FPUs (F).

G_{fp}			
Config ID	IW	A	F
0	128	6	4
3	128	4	4
4	128	4	2
6	128	2	4
12	96	4	4
16	96	2	2
22	64	4	2
24	64	2	4
30	48	4	4
34	48	2	2
40	32	4	2
42	32	2	4
52	16	2	2

Table 2.6 Relative processor energy savings (%) for each algorithm pair (G: Global; L: Local; G+L: Global+Local) for all user slack and predictors, averaged over all applications for different predictor types (S: simple, P: Perfect). Minimum and maximum savings are given in brackets. For comparisons with Local, we do not include points where Local fails to meet the target slack; these are indicated in bold.

Savings	**Relative**	Slack (Predictor Type)					
from	**to**	5% (P)	5% (S)	10% (P)	10% (S)	20% (P)	20% (S)
G+L	Base	46 [27, 68]	39 [26, 67]	51 [35, 68]	49 [35, 67]	54 [43, 69]	52 [42, 68]
G	Local	**-3 [-29, 16]**	**-12 [-42, 12]**	2 [-25, 17]	-5 [-36, 14]	8 [-25, 27]	0 [-36, 22]
G+L	G	**8 [-5, 20]**	**10 [-3, 25]**	10 [0, 17]	11 [2, 22]	10 [2, 21]	10 [2, 24]
G+L	L	**10[-3, 11]**	**6 [-6, 22]**	12 [-3, 30]	8 [-6, 23]	10 [0, 30]	11 [-3, 25]

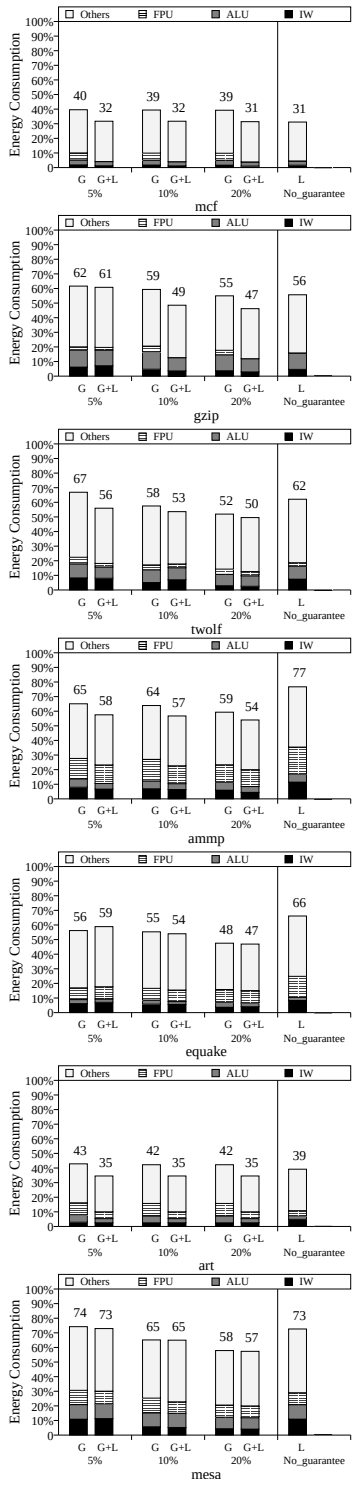

Figure 2.1 Processor energy consumption (normalized to base) for the processors capable of *Global*, *Local*, and *Joint Global + Local* adaptations with perfect predictor.

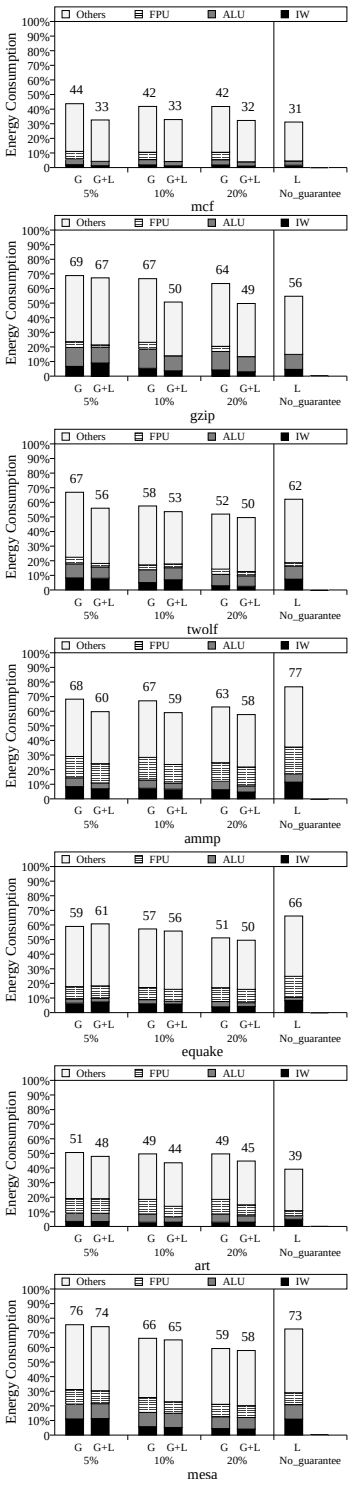

Figure 2.2 Processor energy consumption (normalized to base) for the processors capable of *Global*, *Local*, and *Joint Global + Local* adaptations with simple predictor.

Table 2.7 Instruction window size and number of ALUs and FPUs for the five most occurring phases of each application for *Global* adaptations.

mcf										
PhaseID	%	Slack 5%			Slack 10%			Slack 20%		
		IW	A	F	IW	A	F	IW	A	F
45	26	16	2	1	16	2	1	16	2	1
11	18	16	2	1	16	2	1	16	2	1
9	11	48	2	1	48	2	1	48	2	1
10	10	64	2	1	64	2	1	64	2	1
33	9	48	2	1	48	2	1	48	2	1
gzip-log										
PhaseID	%	Slack 5%			Slack 10%			Slack 20%		
		IW	A	F	IW	A	F	IW	A	F
31	22	48	4	2	48	4	2	64	2	1
54	20	48	4	1	48	4	1	32	2	1
8	17	32	4	1	16	2	1	16	2	1
13	17	64	6	1	64	4	1	32	4	1
57	9	64	6	1	64	4	1	32	4	1
twolf										
PhaseID	%	Slack 5%			Slack 10%			Slack 20%		
		IW	A	F	IW	A	F	IW	A	F
0	57	64	4	1	64	2	1	32	2	1
8	43	96	4	1	48	4	1	32	2	1
ammp										
PhaseID	%	Slack 5%			Slack 10%			Slack 20%		
		IW	A	F	IW	A	F	IW	A	F
9	25	96	2	1	64	2	2	64	2	1
10	18	64	6	1	64	6	1	64	6	1
14	10	48	2	2	48	2	2	48	2	2
24	9	96	2	2	96	2	2	64	2	4
13	8	96	2	2	96	2	2	64	2	1
equake										
PhaseID	%	Slack 5%			Slack 10%			Slack 20%		
		IW	A	F	IW	A	F	IW	A	F
0	36	96	2	1	96	2	1	48	2	1
140	5	96	2	2	96	2	1	48	2	1
138	5	96	2	1	96	2	1	48	2	1
13	4	96	2	1	96	2	1	64	4	1
141	4	128	2	1	96	2	1	48	2	1
art-110										
PhaseID	%	Slack 5%			Slack 10%			Slack 20%		
		IW	A	F	IW	A	F	IW	A	F
11	37	32	2	1	32	2	1	32	2	1
12	12	48	2	1	48	2	1	48	2	1
15	11	32	4	1	32	4	1	32	4	1
14	11	32	2	1	32	2	1	32	2	1
17	6	96	2	2	96	2	1	96	2	1
mesa										
PhaseID	%	Slack 5%			Slack 10%			Slack 20%		
		IW	A	F	IW	A	F	IW	A	F
2	79	96	2	1	48	4	2	48	2	1
1	11	96	2	1	64	6	2	48	2	2
21	4	64	4	2	64	2	1	32	2	1

Table 2.8 Instruction window size and number of ALUs and FPUs for the five most occurring phases of each application for *Global + Local* adaptations.

mcf										
PhaseID	%	Slack 5%			Slack 10%			Slack 20%		
		IW	A	F	IW	A	F	IW	A	F
45	26	16.00	1.74	0.00	16.00	1.74	0.00	16.00	1.44	0.00
11	18	16.00	1.43	0.00	16.00	1.43	0.00	16.00	1.43	0.00
9	11	41.99	1.51	0.00	31.78	1.54	0.00	16.00	1.44	0.00
10	10	31.83	1.48	0.00	31.85	1.43	0.00	16.00	1.42	0.00
33	9	22.04	2.18	0.00	22.04	2.18	0.00	22.04	2.18	0.00
gzip-log										
PhaseID	%	Slack 5%			Slack 10%			Slack 20%		
		IW	A	F	IW	A	F	IW	A	F
31	22	60.79	2.84	0.00	40.22	2.84	0.00	30.76	2.52	0.00
54	20	65.58	2.37	0.00	40.03	2.19	0.00	30.93	2.12	0.00
8	17	16	1.89	0.02	16.00	1.66	0.00	16.00	1.66	0.00
13	17	128	6	4	49.28	2.98	0.00	40.33	2.99	0.00
57	9	65.03	2.86	0.00	52.67	2.68	0.00	30.97	2.63	0.00
twolf										
PhaseID	%	Slack 5%			Slack 10%			Slack 20%		
		IW	A	F	IW	A	F	IW	A	F
0	57	43.74	2.53	0.73	31.10	2.57	0.73	30.95	1.81	0.67
8	43	63.53	2.43	0.33	47.81	2.57	0.71	29.86	1,99	0.47
ammp										
PhaseID	%	Slack 5%			Slack 10%			Slack 20%		
		IW	A	F	IW	A	F	IW	A	F
9	25	93.01	1.00	0.94	63.02	1.53	0.87	63.02	1.53	0.87
10	18	63.24	2.67	0.87	63.24	2.67	0.87	31.87	2.46	0.81
14	10	47.51	1.80	1.27	47.51	1.80	1.27	47.51	1.80	1.27
24	9	95.91	1.30	0.98	95.91	1.30	0.98	95.91	1.30	0.98
13	8	93.83	1.76	1.35	93.83	1.76	1.35	31.68	1.00	1.43
equake										
PhaseID	%	Slack 5%			Slack 10%			Slack 20%		
		IW	A	F	IW	A	F	IW	A	F
0	36	128.00	1.43	1.00	96.00	1.60	0.95	64.00	1.61	1.00
140	5	96.00	1.46	1.76	96.00	1.47	1.00	96.00	1.47	1.00
138	5	96.00	1.45	1.00	96.00	1.45	1.00	48.00	1.47	1.00
13	4	128.00	1.46	1.00	96.00	1.54	1.00	64.00	1.65	1.00
141	4	128.00	1.07	0.94	128.00	1.07	0.94	48.00	1.06	0.94
art-110										
PhaseID	%	Slack 5%			Slack 10%			Slack 20%		
		IW	A	F	IW	A	F	IW	A	F
11	37	96.00	1.30	0.98	96.00	1.30	0.98	96.00	1.30	0.98
12	12	16.00	1.04	0.98	16.00	1.04	0.98	16.00	1.04	0.98
15	11	32.00	1.74	0.99	32.00	1.74	0.99	32.00	1.74	0.99
14	11	48.00	1.27	0.96	48.00	1.27	0.96	48.00	1.27	0.96
17	6	47.97	1.65	0.97	47.97	1.65	0.97	47.97	1.65	0.97
mesa										
PhaseID	%	Slack 5%			Slack 10%			Slack 20%		
		IW	A	F	IW	A	F	IW	A	F
2	79	93.40	3.56	1.27	94.60	3.48	0.99	32.00	1.70	1.15
1	11	128	6	4	95.22	3.54	1.24	48.00	3.11	1.24
21	4	105.26	3.69	1.00	48.00	1.82	1.14	32.00	1.77	1.12

Table 2.9 Mean instruction window size(IW), number of FPUs(F), and ALUs(A) interval information (mean adaptation interval and standard deviation of IPC) for the five most occurring phases of each application for local adaptation.

mcf								
PhaseID	%	IPC	Interval Information			IW	A	F
			Stdev	Grads	Cycles			
45	26	0.05	140%	440	2976	16.75	2.12	0.00
11	18	0.16	174%	413	2733	63.99	1.48	0.00
9	11	0.15	154%	416	2937	56.65	1.49	0.00
10	10	0.16	160%	422	2922	57.57	1.51	0.00
33	9	0.15	52%	435	7965	18.51	2.07	0.00
gzip-log								
PhaseID	%	IPC	Interval Information			IW	A	F
			Stdev	Grads	Cycles			
31	22	1.69	43%	531	344	61.05	2.87	0.00
54	20	1.51	48%	497	360	46.54	2.92	0.00
8	17	1.60	41%	656	441	46.01	3.06	0.00
13	17	2.17	31%	580	301	47.28	3.12	0.00
57	9	1.62	44%	501	337	54.91	2.92	0.00
twolf								
PhaseID	%	IPC	Interval Information			IW	A	F
			Stdev	Grads	Cycles			
0	57	1.22	31.41%	668	588	56.27	2.78	0.70
8	43	1.45	53.63%	768	645	95.40	3.28	0.81
ammp								
PhaseID	%	IPC	Interval Information			IW	A	F
			Stdev	Grads	Cycles			
9	25	1.45	34%	1066	743	119.68	2.44	2.61
10	18	1.06	43%	1045	979	123.19	2.41	2.37
14	10	1.20	33%	882	543	117.47	2.54	2.48
24	9	1.58	57%	1575	1886	127.77	2.99	3.00
13	8	1.11	35%	957	720	119.50	2.49	2.46
equake								
PhaseID	%	IPC	Interval Information			IW	A	F
			Stdev	Grads	Cycles			
0	36	0.49	4%	9517	19072	128.00	1.64	3.88
140	5	0.55	21%	3371	6100	128.00	1.45	3.03
138	5	0.58	18%	2482	4292	128.00	1.45	2.37
13	4	0.54	24%	6701	12429	128.00	1.63	3.76
141	4	0.59	24%	2619	4441	128.00	1.67	2.16
art-110								
PhaseID	%	IPC	Interval Information			IW	A	F
			Stdev	Grads	Cycles			
11	37	0.48	72%	766	1626	125.97	1.40	1.00
12	12	0.40	79%	803	1997	126.99	1.41	0.99
15	11	0.56	66%	655	1177	127.98	1.72	0.97
14	11	0.47	69%	714	1505	127.99	1.52	0.98
17	6	0.66	66%	654	1004	126.19	1.70	0.99
mesa								
PhaseID	%	IPC	Interval Information			IW	A	F
			Stdev	Grads	Cycles			
2	79	1.73	27%	965	587	94.76	3.54	1.35
1	11	1.81	26%	971	570	107.85	3.49	1.45
21	4	1.74	23%	927	535	100.01	3.54	1.19

Table 2.10 Percentage performance degradation, relative to the nonadaptive base architecture for each algorithm, slack, and predictor type (S: simple, P: Perfect).

	Global						Global+Local						Local
Slack Predictor	5% P	5% S	10% P	10% S	20% P	20% S	5% P	5% S	10 % P	10% S	20% P	20% S	
mcf	3	5	5	7	6	8	4	4	5	5	10	7	3
gzip	3	4	6	6	16	13	3	2	9	6	15	12	6
twolf	4	4	9	9	18	18	5	5	9	9	19	19	5
ammp	3	5	7	10	13	15	4	5	7	8	16	17	2
equake	3	4	4	5	17	17	4	3	6	7	18	19	3
art	3	3	3	4	3	4	4	4	4	5	4	5	3
mesa	1	3	9	9	19	19	4	3	9	7	19	13	6

Table 2.11 Relative energy savings (%) for different processor architecture pairs and user slacks averaged over all applications.

Savings from	**Relative to**	Slack		
		5%	10%	20%
Reduced G	Base	35 [18, 59]	39 [30, 60]	45 [40, 60]
G	Reduced G	15 [0, 63]	13 [2, 60]	9 [0, 15]
Reduced G+L	Base	**44 [23, 68]**	48 [31, 68]	53 [43, 68]
G+L	Reduced G+L	**3 [0, 5]**	5 [0, 10]	4 [0, 8]

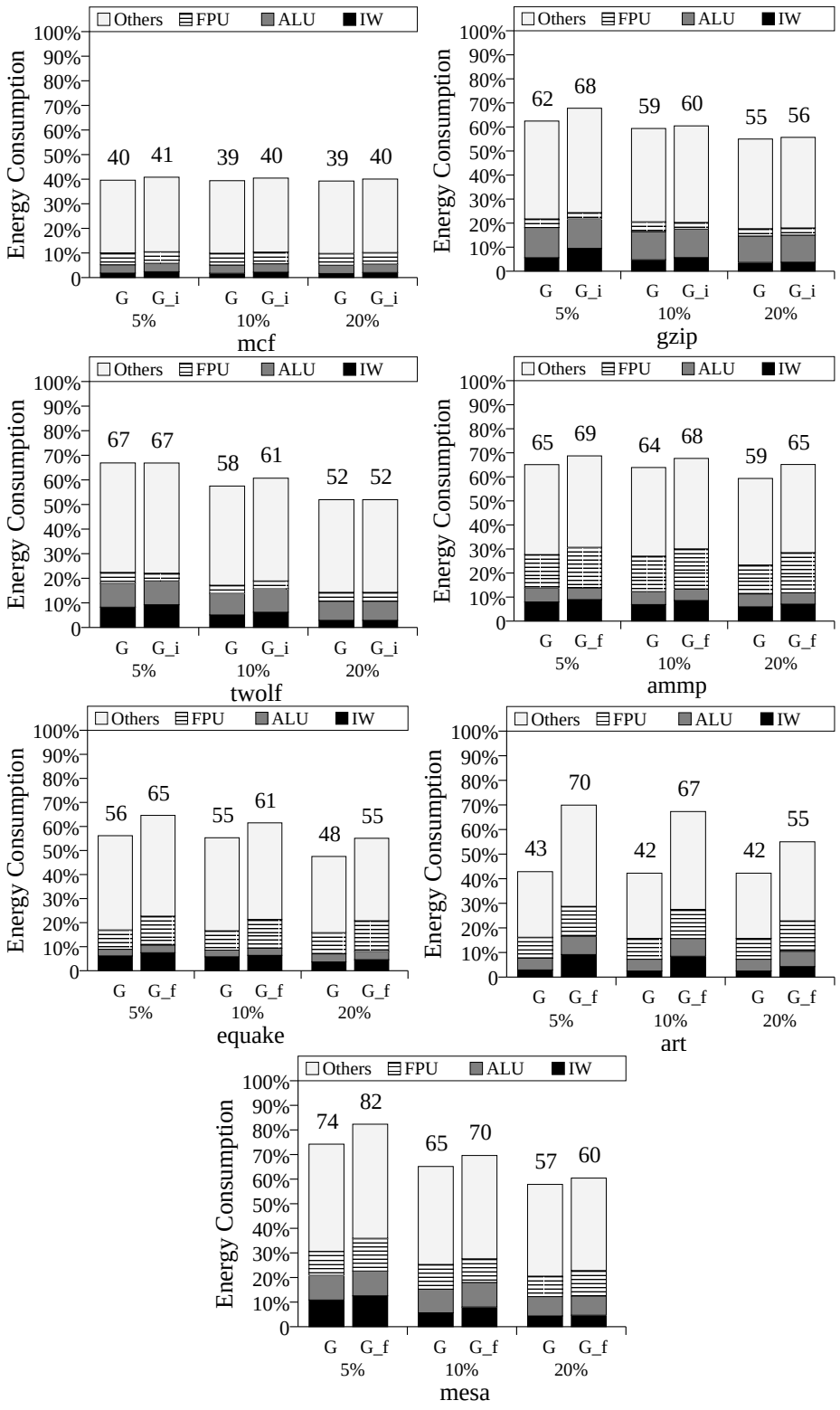

Figure 2.3 Processor energy consumption (normalized to base) for the processors capable of *Global* adaptations with G_{54} and G_i/G_f architectural configuration sets.

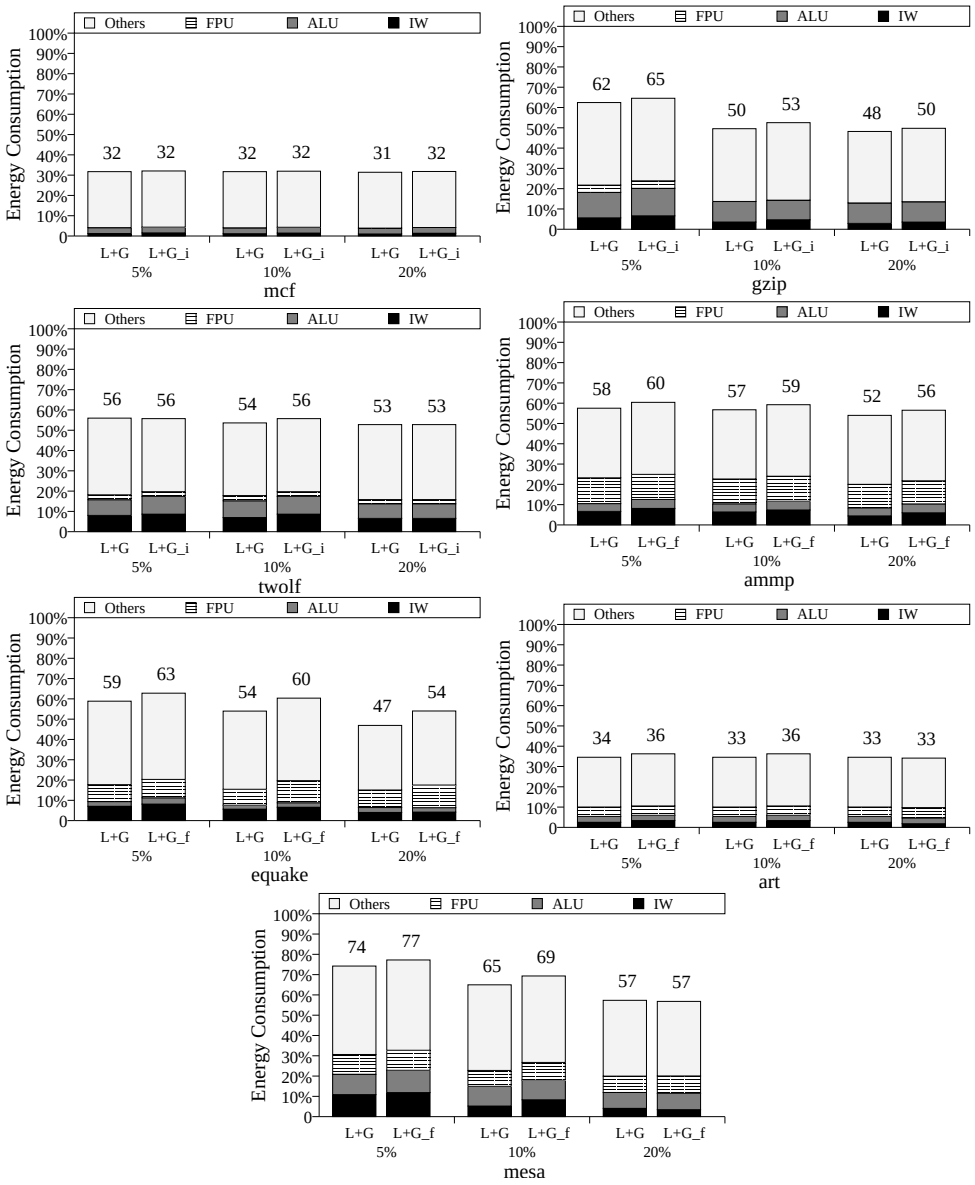

Figure 2.4 Total processor energy consumption (normalized to base) for the processors capable of *Global + Local* adaptations for architectural sets G_{54} and G_i/G_f.

Table 2.12 Instruction window size and number of ALUs and FPUs for the five most occurring phases of each application for Global adaptations with reduced profiling.

mcf										
PhaseID	%	Slack 5%			Slack 10%			Slack 20%		
		IW	A	F	IW	A	F	IW	A	F
45	26	32	2	1	32	2	1	32	2	1
11	18	32	2	1	32	2	1	32	2	1
9	11	48	2	1	48	2	1	48	2	1
10	10	48	2	1	32	2	1	32	2	1
33	9	48	2	1	48	2	1	48	2	1
gzip-log										
PhaseID	%	Slack 5%			Slack 10%			Slack 20%		
		IW	A	F	IW	A	F	IW	A	F
31	22	128	4	1	64	4	1	64	4	1
54	20	64	4	1	64	4	1	32	2	1
8	17	32	4	1	32	2	1	32	2	1
13	17	128	4	1	64	4	1	32	4	1
57	9	128	4	1	64	4	1	32	4	1
twolf										
PhaseID	%	Slack 5%			Slack 10%			Slack 20%		
		IW	A	F	IW	A	F	IW	A	F
0	57	64	4	1	64	4	1	32	2	1
8	43	128	4	1	64	4	1	32	2	1
ammp										
PhaseID	%	Slack 5%			Slack 10%			Slack 20%		
		IW	A	F	IW	A	F	IW	A	F
9	25	96	2	2	96	2	2	96	2	2
10	18	96	2	2	96	2	2	96	2	2
14	10	48	2	2	48	2	2	48	2	2
24	9	96	2	2	96	2	2	96	2	2
13	8	96	2	2	96	2	2	48	2	2
equake										
PhaseID	%	Slack 5%			Slack 10%			Slack 20%		
		IW	A	F	IW	A	F	IW	A	F
0	36	96	2	2	96	2	2	48	2	2
140	5	96	2	2	96	2	2	96	2	2
138	5	96	2	2	96	2	2	48	2	2
13	4	96	2	2	96	2	2	64	4	2
141	4	128	4	2	96	2	2	96	2	2
art-110										
PhaseID	%	Slack 5%			Slack 10%			Slack 20%		
		IW	A	F	IW	A	F	IW	A	F
11	37	48	2	2	48	2	2	16	2	2
12	12	48	2	2	48	2	2	48	2	2
15	11	128	4	2	48	2	2	48	2	2
14	11	48	2	2	48	2	2	48	2	2
17	6	96	2	2	96	2	2	96	2	2
mesa										
PhaseID	%	Slack 5%			Slack 10%			Slack 20%		
		IW	A	F	IW	A	F	IW	A	F
2	79	96	4	4	64	4	2	48	2	2
1	11	128	4	2	128	4	2	48	2	2
21	4	64	4	2	64	4	2	48	2	2

Table 2.13 Instruction window size and number of ALUs and FPUs for the five most occurring phases of each application for Global+Local adaptations with reduced profiling.

mcf										
PhaseID	%	Slack 5%			Slack 10%			Slack 20%		
		IW	A	F	IW	A	F	IW	A	F
45	26	22.22	1.73	0.00	22.22	1.73	0.00	22.22	1.73	0.00
11	18	45.91	1.67	0.00	45.91	1.67	0.00	45.91	1.67	0.00
9	11	41.99	1.51	0.00	31.78	1.54	0.00	31.78	1.54	0.00
10	10	31.85	1.51	0.00	31.85	1.43	0.00	31.85	1.51	0.00
33	9	22.22	1.73	0.00	20.35	1.71	0.00	20.35	1.71	0.00
gzip-log										
PhaseID	%	Slack 5%			Slack 10%			Slack 20%		
		IW	A	F	IW	A	F	IW	A	F
31	22	60.79	2.84	0.00	60.79	2.84	0.00	40.14	2.83	0.00
54	20	55.46	2.29	0.00	40.03	2.19	0.00	40.03	2.57	0.00
8	17	35.16	1.38	0.00	35.16	1.38	0.00	35.16	1.38	0.00
13	17	128	6	4	53.41	2.83	0.00	39.33	3.09	0.00
57	9	128	6	4	68.38	2.64	0.00	40.26	2.57	0.00
twolf										
PhaseID	%	Slack 5%			Slack 10%			Slack 20%		
		IW	A	F	IW	A	F	IW	A	F
0	53	43.72	2.38	0.73	43.72	2.58	0.73	30.95	1.81	0.67
8	47	74.72	2.14	0.53	54.72	3.14	0.53	29.86	1.99	0.47
ammp										
PhaseID	%	Slack 5%			Slack 10%			Slack 20%		
		IW	A	F	IW	A	F	IW	A	F
9	25	120.82	2.62	0.99	120.82	2.62	0.99	120.82	2.62	0.99
10	18	94.97	1.75	1.41	63.35	1.18	1.41	31.87	2.46	0.81
14	10	47.51	1.80	1.27	47.51	1.80	1.27	47.51	1.80	1.27
24	9	95.91	1.30	0.98	95.91	1.30	0.98	95.91	1.30	0.98
13	8	93.83	1.76	1.35	93.83	1.76	1.35	31.68	1.00	1.43
equake										
PhaseID	%	Slack 5%			Slack 10%			Slack 20%		
		IW	A	F	IW	A	F	IW	A	F
0	36	128.00	1.76	2.14	96.00	1.53	1.96	96.00	1.53	1.96
140	5	96.00	1.47	1.76	96.00	1.47	1.76	96.00	1.47	1.76
138	5	96.00	1.41	1.63	96.00	1.41	1.63	48.00	1.49	2.15
13	4	96.00	1.56	1.92	96.00	1.56	1.92	96.00	1.56	1.92
141	4	128.00	1.07	1.22	128.00	1.07	1.22	128.00	1.07	1.22
art-110										
PhaseID	%	Slack 5%			Slack 10%			Slack 20%		
		IW	A	F	IW	A	F	IW	A	F
11	37	95.66	1.41	1.00	95.66	1.41	1.00	16.00	1.55	0.97
12	12	32.00	1.35	0.97	32.00	1.35	0.97	32.00	1.35	0.97
15	11	32.00	1.32	0.96	32.00	1.32	0.96	32.00	1.32	0.96
14	11	44.43	1.07	1.00	44.43	1.07	1.00	44.43	1.07	1.00
17	6	47.97	1.65	0.97	47.97	1.65	0.97	47.97	1.65	0.97
mesa										
PhaseID	%	Slack 5%			Slack 10%			Slack 20%		
		IW	A	F	IW	A	F	IW	A	F
2	79	93.40	3.56	1.27	92.52	3.56	1.44	32.00	1.77	1.12
1	11	128	6	4	94.02	3.57	1.39	48.00	3.14	1.27
21	4	105.26	3.69	1.00	48.00	2.73	0.99	48.00	2.14	1.31

CHAPTER 3

JOINT PROCESSOR AND MEMORY ADAPTATION

As explained in Section 1, processor and memory energy adaptations need to be done in a cooperative way to minimize total energy consumption while still providing a performance guarantee. In this section, we describe how this cooperation can be done in a loosely coupled way, and we present the key method to successful cooperation: determining the distribution of the performance slowdown to processor and memory. We also discuss how the previous memory adaptation algorithms can be changed to better cooperate with processor adaptation.

3.1 Joint Adaptation

Processor and memory adaptations interact in two major ways. First, they can both contribute to the total execution slowdown. Therefore, to provide performance guarantee, the two components need to share the available performance slowdown specified by users. Second, the adaptation of one can affect the other. For example, the processor adaptation can affect the idle time between memory requests and thereby lead to reactions by the underlying memory algorithm that adapts at fine time granularity (e.g. the PD algorithm). Similarly, the memory adaptation can reduce the CPU utilization and result in reaction by the processor adaptation algorithm (e.g., the Local algorithm).

We propose two techniques to address the above two interactions. To handle the first interaction, we use a method to optimally distribute the user specified slack between processor and memory with a goal of minimizing the total energy consumption. To address the second interaction, we

modify processor and memory adaptation algorithms to relax the coupling between them so that each one of the components can adapt independently based on the slack allocated to it and the workload characteristics instead of the influence of the other component.

In this section, we first present the method to optimally distribute the total slack and then describe the changes to the PD memory adaptation algorithm to decouple the influence of processor adaptations on memory. The modifications in the processor adaptation algorithm to decouple it from the influence of memory adaptation have already been described in Section 2.2.2.

3.1.1 Slack distribution

As with the processor algorithm, we seek to exploit slack at the granularity of a phase interval, i.e., we would like to slow down each phase interval by the user specified slack. The goal of the *slack distribution* algorithm is to divide this total available slack between processor and memory. A simple way to perform this task is to divide the slack equally, half for memory and half for processor. This method is straightforward, but it does not minimize total energy consumption. The reason is that the energy savings per unit slack are different for processor and memory. For example, with a total 20% slack, there may be little extra energy savings giving 10% slack instead of 5% slack to processor, but the energy savings may be significant if memory can have 15% slack instead of 10% slack. As shown by our experimental results, equal distribution does not achieve minimal total energy consumption.

Our goal is to choose a slack distribution that can provide a performance guarantee and at the same time minimize the total energy consumption. To do that, we need to first convert this problem into an optimization problem with minimizing the total energy consumption as the goal and the total available slack as the constraint.

3.1.2 Problem formalization

Formally speaking, suppose the available slack is $Slack$ (e.g., 20%) specified by users; the total execution time without any processor and memory adaptation is T_{base}; and S_{cpu} and S_{mem} are, respectively, the slack distributed to processor and memory. Since both processor and memory adaptation algorithms are directed by the amount of slack allocated to them, the actual execution time with energy adaptation, T, is affected by the slack allocation and therefore can be expressed

as a function of S_{cpu} and S_{mem}. To satisfy the performance constraint, T should be smaller than $T_{base} * (1 + Slack)$. Similarly, the slack allocation can also affect the processor and memory energy consumption: a larger slack can allow the processor or the memory to transition into a lower-power configuration. Therefore, the CPU and memory energy consumption, E_{cpu} and E_{mem}, are also functions of the user allocated slacks S_{cpu} and S_{mem}.

Consequently, the slack distribution problem can be converted into the following optimization problem:

minimize $E_{cpu}(S_{cpu}, S_{mem}) + E_{mem}(S_{cpu}, S_{mem})$

subject to $T(S_{cpu}, S_{mem}) \leq (1 + Slack) * T_{base}$

3.1.3 Solving for function $T(S_{cpu}, S_{mem})$

To solve the above problem, we need to first find the function T, i.e., how slack distributions S_{cpu} and S_{mem} affect the actual execution time T.

Let us first consider a simple case where only the processor is adapting with an allocated slack S_{cpu}. The algorithm described in Section 2.2.1 chooses a configuration which consumes the least energy and guarantees the actual slowdown to be less than the allocated slack S_{cpu}. Suppose with such a configuration, the actual execution time is T'. Based on the performance constraint, T' and T_{base} have the following relationship:

$$T' \leq T_{base} * (1 + S_{cpu}) \tag{3.1}$$

Now let us consider the more complicated case where, in addition to the processor, memory is also adapting with an allocated slack S_{mem}. To memory, the adapting processor is just a low configuration new processor, so the memory adaptation would further delay the execution time from T' to T. Since the memory adaptation algorithm also provides performance guarantee, T and T' will have to satisfy the following constraint:

$$T \leq T' * (1 + S_{mem}) \tag{3.2}$$

Substituting T' using Equation(3.1), we have

$$T \leq T_{base} * (1 + S_{cpu}) * (1 + S_{mem}) \tag{3.3}$$

Therefore, the actual execution time T for a given slack distribution S_{cpu} and S_{mem} is bounded by $T_{base} * (1 + S_{cpu}) * (1 + S_{mem})$. Therefore, if the latter satisfies the overall slack constraint, so does T. In other words, as long as

$$T_{base} * (1 + S_{cpu}) * (1 + S_{mem}) \leq (1 + Slack) * T_{base} \tag{3.4}$$

we would satisfy the Performance constraint: $T < (1 + Slack) * T_{base}$.

The above deduction is based on the assumption that during the phase interval, memory adaptation does not influence the processor adaptation. This is clearly the case for purely global processor adaptation since the processor does not change its configuration during the phase interval. Section 2.2.2 described how we also modify the processor's local adaptations to enable this assumption.

3.1.4 Solve for functions $E_{cpu}(S_{cpu}, S_{mem})$ and $E_{mem}(S_{cpu}, S_{mem})$

The second challenge to solve the optimization problem is to find the relationship between the slack distribution (S_{cpu} and S_{mem}) and the processor and memory energy consumption (E_{cpu} and E_{mem}).

One method to address this challenge is to analytically estimate E_{cpu} and E_{mem} based on S_{cpu} and S_{mem}. However, both processor and memory use complex adaptation algorithms (PD for memory and Global, Local or Global+Local for processor) that are difficult to model analytically even if we consider each component in an isolation. This is because these algorithms react to workload changes at very fine time granularity. Moreover, the processor and memory interaction makes this problem even more complicated.

Our solution is to use a profiling-based method which profiles E_{mem} and E_{cpu} for different slack distributions at run time. In other words, for each phase and a given slack distribution, we can use one of the current phase occurrences to find the energy consumption by memory and processor, E_{mem} and E_{cpu}. Based on the profiling information, we can choose the best slack distribution that gives the least total energy consumption $E_{mem} + E_{cpu}$ for this phase.

3.1.5 Solving the optimization problem

To solve the optimization problem, we use a linear search in a discrete space. First we divide S_{cpu} into several steps from 0% to $Slack$. Second, we can calculate S_{mem} based on Equation (3.4). Even though we can choose many S_{mem} values based on Equation (3.4), it is better to choose the maximum S_{mem} that satisfies Equation (3.4) because the energy consumption E_{mem} is usually smaller with a larger slack (so that the memory has more opportunity to go to low power modes).

After the above two steps, there are only a limited number of slack distributions that we can choose. Therefore, for each distribution, we can use the profiling method described above to find the best possible slack distribution that minimizes total energy consumption. In our experiments, we used only 11 total distributions (i.e., 11 occurrences of each phase) to obtain the best slack distribution.

3.1.6 Joint algorithm summary

Finally, we summarize the slack distribution algorithm below.

Algorithm 1 Slack Distribution Algorithm (called at the beginning of each phase)

1: Profile different processor architectural configurations.
2: Let $STEP$ be N ($N = 11$ in our case)
3: **for** each i *in* $0,\ 1, ...,\ STEP$ **do**
4: $S_{cpu} = \frac{i \cdot Slack}{STEP}$. Choose a processor configuration based on S_{cpu}.
5: $S_{mem} = \frac{(1+Slack)}{(1+S_{cpu})} - 1$. Choose memory thresholds based on S_{mem}.
6: Profile for a phase interval and record the sum of processor and memory energy to be E_i.
7: **end for**
8: Compare E_i, for $i \in 0,\ 1, ..,\ STEP$ and record the optimal slack division between processor and memory for minimum energy.
9: Start processor and memory adaptation.

3.2 Overhead Analysis

The joint algorithm is used with processor and memory adaptation algorithms. The overhead for processor adaptation algorithms has already been discussed in Section 2.2.4. Overheads for memory adaptation algorithms have been discussed in discussed in [14]. Below we discuss the overheads of the joint algorithm.

The number of profiling intervals has now increased from number of processor architectural

configurations, i.e., N_{config} by the number $STEP$ (11 in our implementation) for each phase. During the adaptation phase, the joint algorithm is invoked at the beginning of each interval to decide the optimal slack allocation using a table lookup. The overhead of a table lookup is of $O(1)$. The joint algorithm also needs to store the optimal slack allocation information for each phaseID. The space overhead for storing this information is $O(P)$, where P is the number of phases of the application. Typically P is smaller (less than 35).

3.3 Modified PD Algorithm

For our work, we choose to adopt the PD algorithm since it is currently the best available performance guaranteed memory adaptation algorithm. Unfortunately, there are several aspects of this algorithm that are affected by the joint processor-memory adaptation. To address these problems, we modify the PD algorithm to better cooperate with the processor adaptation as follows:

- Since the processor adaptation algorithms adapt at the granularity of a phase interval (Section 2.2.1), we choose to invoke the PD algorithms also at the same granularity instead of at epoch granularity.

- In the original PD algorithm, thresholds are calculated using several heuristic based functions [14] based on the prediction on memory access behavior for the next epoch. Since now PD adjusts its threshold setting at phase granularity, its prediction on memory access behavior during the next phase is more accurate than the original algorithm. This is because the memory access (cache miss) behavior is fairly stable across different occurrences of the same phase [27].

- In the original PD algorithm, if the thresholds are too conservative or aggressive for the last epoch, PD adjusts the thresholds for the next epoch using a dynamically adjustable parameter called the $SelfAdjustFactor$. Unfortunately, this self-tuning is hard to achieve in our joint adaptation scheme because our scheme is based on a profiling-based method. Since different $SelfAdjustFactor$ values would give different energy saving, to find the best value would require profiling for each phase and each slack distribution with various values of this $SelfAdjustFactor$. Doing this would significantly increase the number of occurrences

of each phase used for profiling. Therefore, we use an empirically determined value (10) as the $SelfAdjustFactor$ and use it throughout all experiments. Our experimental results show that it works well for all cases.

3.4 Performance Guarantee

In case of a perfect predictor for next phase, the performance guarantee can be easily provided. This is because at the beginning of the interval, the joint algorithm gives the most optimal slack allocation between the processor and the memory and then the processor and memory each adapts in an isolated manner using the slack that has been allocated to each of the components. This is also shown to be true from our experimental results.

However, a phase misprediction could result in choosing a slack division that violates the performance constraint by using too much slack. To accommodate this we can keep track of the slack used in each phase interval, and if too much slack is used then we can switch to the base processor and *active* power mode for the processor and the memory, respectively. This is similar to what has been done for the memory adaptation algorithms in [14].

3.5 Results

This section presents results to show the benefits of joint processor and memory adaptation. Since the Local processor algorithm does not provide a performance guarantee, we present results with only the Global and Global+Local processor adaptations. For space reasons, we only present results for 20% slack and perfect phase prediction.

3.5.1 Overall results

Figures 3.1 and 3.2 present the main data for energy consumption for systems with only processor adaptation (G or G+L), only memory adaptation (M), and joint processor and memory adaptation, normalized to the base case with no adaptation for perfect and simple predictors, respectively. For the joint processor-memory adaptation case, we present results with the optimal distribution of slack among processor and memory (G,O or G+L,O) as described in Section 3.1.4. To understand the impact of choosing an optimal distribution, we also present results with an equal slack distribution (G,E or G+L,E). Table 3.1 summarizes this data by presenting the relative energy savings between

key pairs of adaptation algorithms (average across all applications, minimum, and maximum) for both perfect and simple predictors, respectively. Table 3.2 gives the actual percentage performance degradation seen by the joint G,O and G+L,O algorithms.

Our high-level results are as follows:

- Adapting both processor and memory provides significant energy savings over adapting either the processor alone or memory alone. Relative to processor adaptation alone, the joint algorithm with optimal slack distribution gives energy savings of 28% considering only Global processor adaptation and 30% considering Global+Local adaptation for *perfect* predictor. Relative to memory adaptation alone, G,O gives savings of 44% and G+L,O gives savings of 48%. Further, the gap between the Global+Local vs. Global processor adaptations is much smaller when memory adaptation is included, thus reducing the impact of the "hard-to-predict" Local processor adaptations.

- We also consider the effect of a simple predictor on the energy savings. With *simple* predictor, the joint algorithm with optimal slack distribution gives energy savings of about 21% considering only Global processor adaptation and 22% considering Global+Local adaptation for simple predictor. Relative to memory alone, G,O gives savings of 37% and G+L,O gives savings of 40%.

- The joint processor-memory adaptation algorithms maintain performance within the specified target in all cases. In many cases, the algorithm is able to exploit the majority of the available slack.

- Compared to equal slack distribution, we find that the optimal distribution saves 9% more energy on average with a maximum of 14% with both Global and Global+Local processor adaptations with a perfect predictor. However with simple predictor, the optimal distribution gives only about 6% more energy savings on an average with a maximum of 13% for Global processor adaption. For Global+Local adaption, these values further decrease to 3% and 10% respectively. These results indicate that our optimal slack distribution algorithm is effective but we need to improve our predictor accuracy.

  The next subsection provides more detailed data.

3.5.2 Detailed data on impact of optimal slack distribution

To further illustrate the impact of the optimal slack distribution algorithm, Figures 3.3(a) and (b) present the energy consumption for one phase occurrence of each of the applications as the slack distribution between the processor and memory is varied from 0% to 20% for G,O and G+L,O adaptations, respectively. These results are for a perfect predictor.

The figures clearly show that there is no single slack distribution that is optimal for all applications. For example, benchmarks *art* and *mcf* use a slack division so that memory gets a larger part of the slack and the processor is given about 4% and 6% slack, respectively. This follows from our previous analysis of processor-only adaptations – increasing the slack from 5% to 20% for processor adaptations did not help these applications much due to their memory intensive nature; it is better to let memory exploit most of the available slack for energy savings. On the other hand, the computationally intensive benchmarks such as *gzip* or *twolf* choose a distribution where the processor is allocated about 16% and 14% slack, and memory is allocated little slack.

These results motivate the need to make an application-specific choice of slack distribution among processor and memory. Choosing a "one size fits all" slack distribution gives up significant potential for at least some application (e.g., choosing equal slack sacrifices 14% benefit for *mcf*).

3.6 Figures and Tables

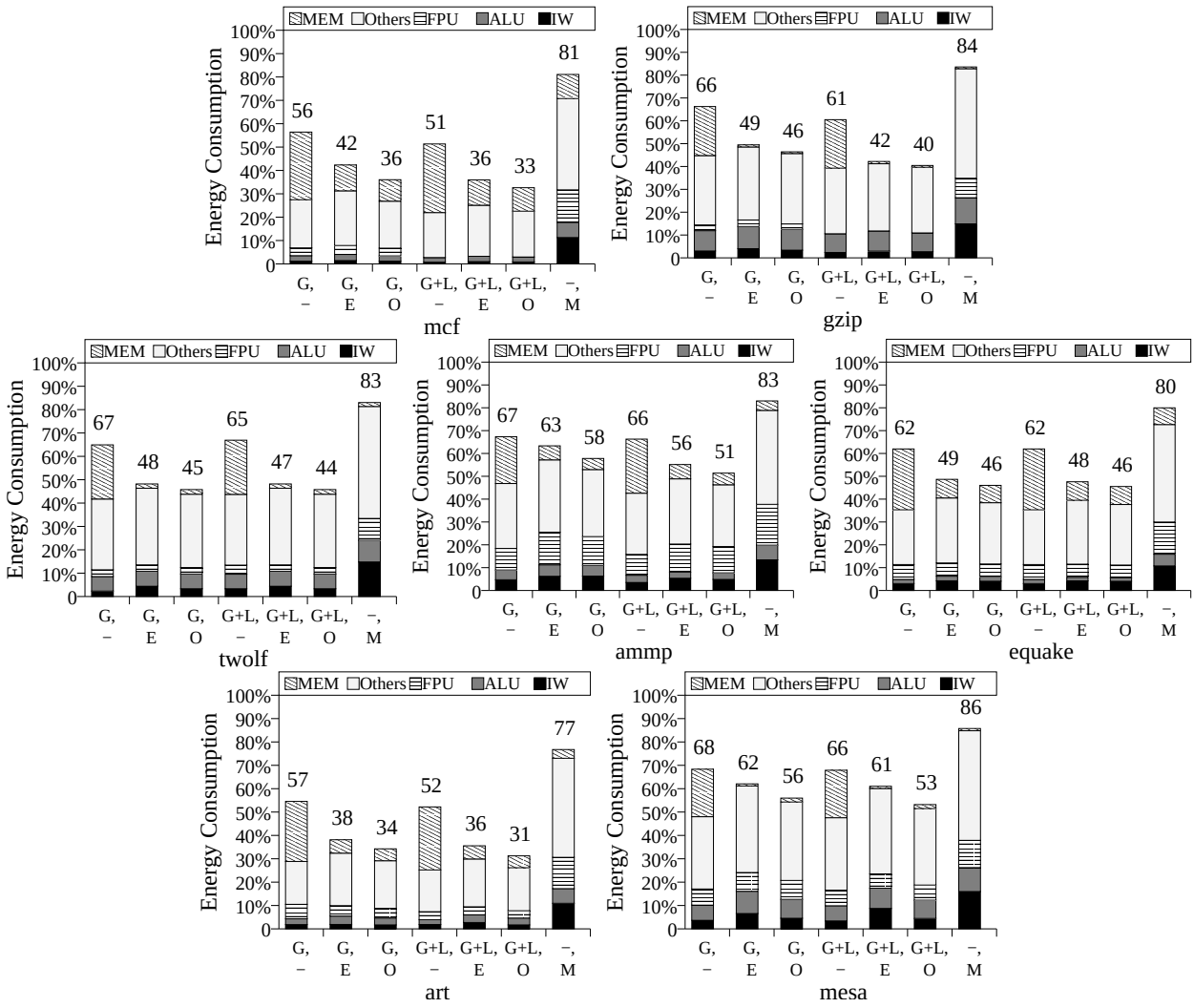

Figure 3.1 Total energy consumption (normalized to base) for different processor, memory, and joint adaptations using a **perfect** phase predictor. **G,-:** Global processor, no memory; **G,E:** Global processor and memory with equal slack division; **G,O:** Global processor and memory with optimal slack division; **G+L,-:** Global+Local processor, no memory; **G+L,E:** Global+Local processor and memory with equal slack division; **G+L,O:** Global+Local processor and memory with optimal slack division; **-,M:** No processor, only memory adaptation.

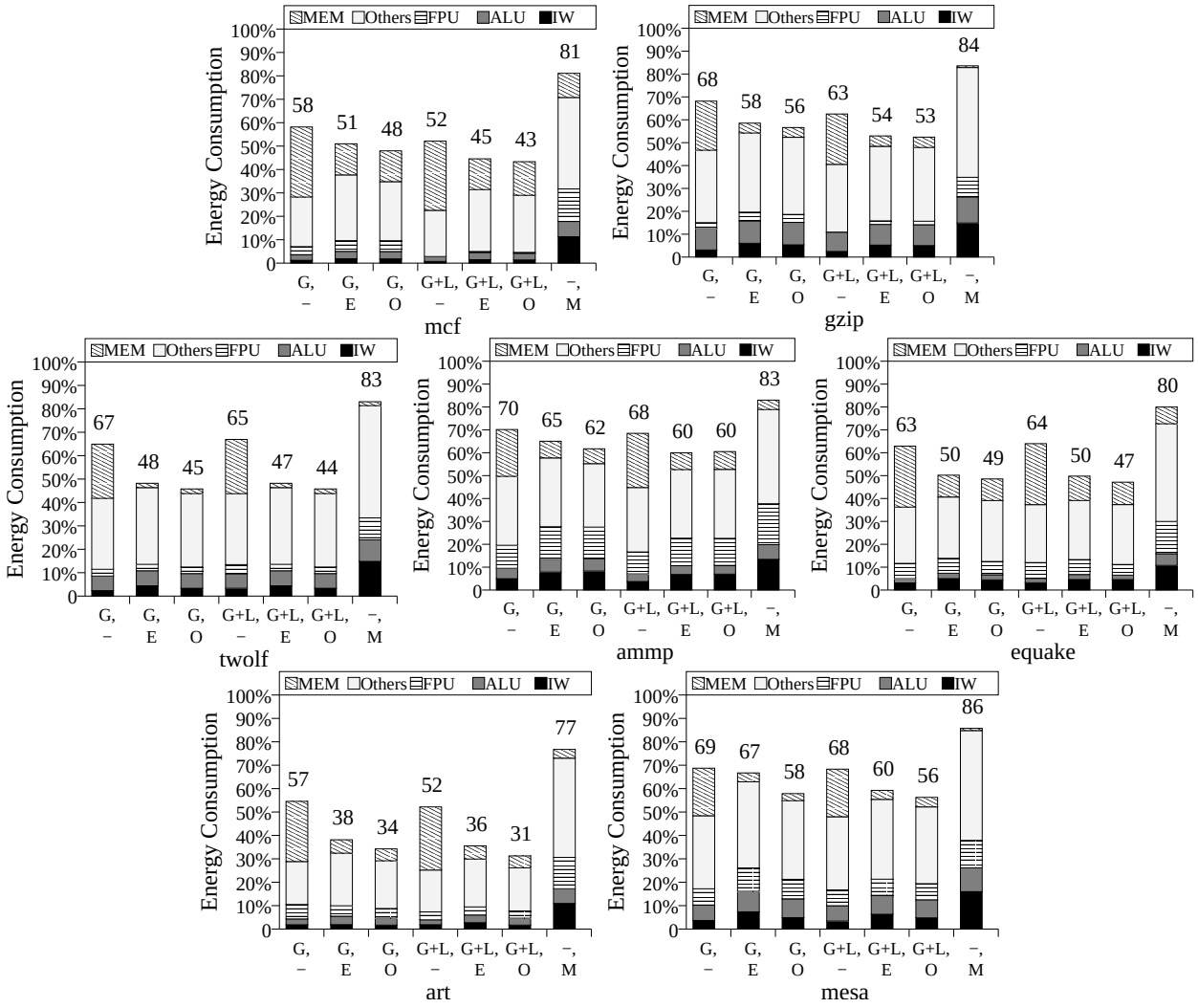

Figure 3.2 Total energy consumption (normalized to base) for different processor, memory, and joint adaptations using a **simple** phase predictor. **G,-:** Global processor, no memory; **G,E:** Global processor and memory with equal slack division; **G,O:** Global processor and memory with optimal slack division; **G**+L,-: Global+Local processor, no memory; **G+L,E:** Global+Local processor and memory with equal slack division; **G+L,O:** Global+Local processor and memory with optimal slack division; **-,M:** No processor, only memory adaptation.

Table 3.1 Relative average energy savings (in %) for different pairs of algorithms for a user slack of 20%.

Savings from **Relative to** **Predictor**	G,O vs G	G,O vs M	G,O vs G,E	G+L,O vs G+L	G+L,O vs M	G+L,O vs G+L,E	G+L,0 vs G,0
Perfect	28 [13, 40]	44 [30, 56]	9 [6, 14]	30[20, 35]	48 [38,60]	9 [4,14]	7 [0, 13]
Simple	21 [11, 34]	37 [25, 47]	6 [3, 13]	22[11, 36]	40 [28,51]	4 [2,13]	3 [-1, 10]

Table 3.2 Percentage performance degradation, relative to nonadaptive base architecture.

Application	**Predictor**	mcf	gzip	twolf	ammp	equake	art	mesa
G,O	Perfect	16	10	15	15	18	8	17
G,O	Simple	12	7	15	7	17	9	19
G+L,O	Perfect	15	11	17	18	18	10	18
G+L,O	Simple	13	9	17	19	19	10	18

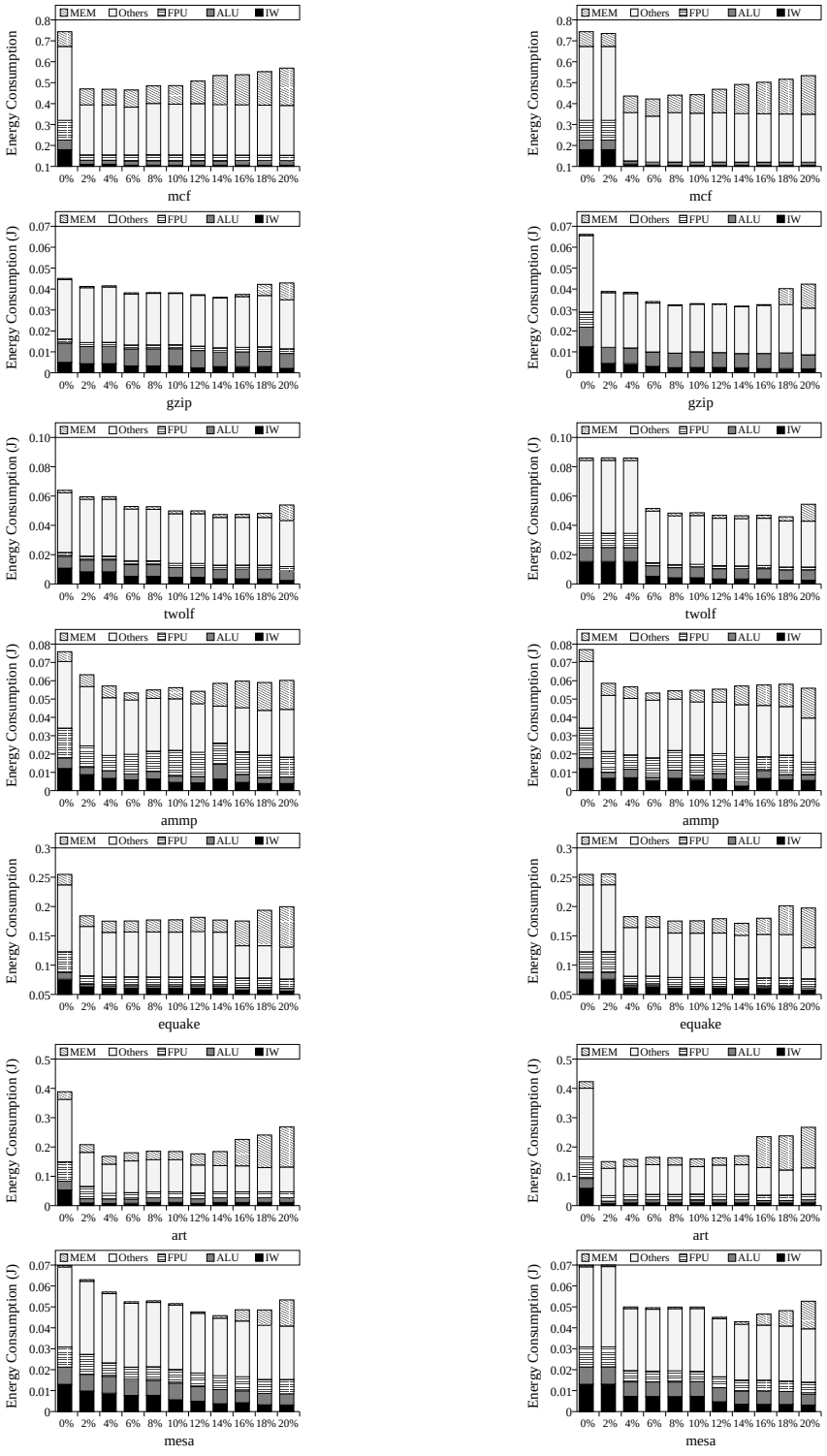

Figure 3.3 Total energy consumption for one phase (normalized to base) for joint processor and memory adaptation, with different slack distributions. A point of i% on the X-axis indicates that the processor is given i% slack and memory is given the remaining slack. Part (a) uses Global part (b) uses Global+Local adaptation for the processor.

CHAPTER 4

RELATED WORK

In this chapter we discuss related work not already discussed in Chapters 2 and 3.

4.1 Processor Adaptation

There has been a substantial work on control algorithms for processor adaptations for savings energy for general-purpose applications. Most of the work on architectural adaptation adapts either at fine granularity or at course granularity and do not provide performance guarantee. Sherwood et al. [17] and Albonesi [6] adapt caches to minimize power consumption. Sherwood et al. [17] discusses dynamically reducing the cache size and Albonesi [6] discusses disabling a subset of set associative cache during periods of modest cache activity. Balasubramonian et al. [28] propose a dynamic reconfiguration algorithm for configuring data cache hierarchy. Iyer and Marculescu [9] adapt the register update unit and the fetch rate for power optimization. It keeps track of the utilization of the resources depending on the code which is executed. The profiled information is used to choose the optimum configuration. Manne et al. [10] use a technique called pipeline gating to reduce the number of speculative issued instructions in the pipeline, minimizing the wasteful work being done and hence reducing the power consumption. Maro et al. [20] reduced the power dissipation by disabling one of the two integer pipelines and/or the floating point pipe at runtime during the execution of program. Pouwelse et al. [29] describe an energy priority scheduling heuristic which, given a set of tasks, yields a clock schedule for controlling the speed (voltage) of the processor. It orders the tasks according to the deadlines and the frequency of overlap of tasks. Weiser et al. [30] uses the number of instructions executed for a given amount of energy as a metric to reduce the cycle time of the CPU to obtain power savings.

Sherwood et al. [17] performed a brief evaluation of processor energy adaptations at the phase granularity. They focus on providing energy savings without significant performance loss, but do not provide a performance guarantee in the event of phase mispredictions. Huang et al. [15] propose DEETM, which is also a global algorithm that adapts at the granularity of several milliseconds. This algorithm does not exploit local variability and does not provide a performance guarantee. More recently, Huang et al. [16] developed an algorithm that adapts at the temporal granularity of subroutines. It uses offline profile information to select the best adaptations for the subroutines for given target slack. It would be interesting to determine how the static subroutine based global adaptation boundaries compare with dynamic phase based global boundaries used in our work, but this is outside the scope of this work. In any case, the recent work by Huang et al. also does not consider fine-grained adaptation and does not provide a performance guarantee.

4.2 Memory Adaptation

To reduce memory energy consumption, modern memory such as RDRAM [12] allows each memory chip to transition from normal *active* operating mode into several low-power operating modes, namely, *standby*, *nap* and *powerdown*. To service a request, a chip needs to transition from low power modes to *active*, which incurs extra delay and energy costs. To utilize the lower power modes, the key is to have effective control algorithms to decide when and which power mode to put each chip into. Researchers have recently proposed several memory adaptation algorithms [13, 14] and have shown that a dynamic scheme that transitions a chip into low power modes after a threshold of idle time performs better than a static scheme which places all chips in a fixed power mode except when it is necessary to service a request. Unfortunately, the dynamic scheme requires painstaking threshold tuning for each workload and can significantly degrade performance when the wrong set of thresholds are chosen.

Recently, Li et al. [14] proposed a method that can effectively provide a performance guarantee in memory adaptation. The main idea is to keep track of the slowdown in execution time introduced by the underlying memory adaptation and force all chips to active when the observed slowdown is greater than the maximum slowdown allowed by users. To improve slowdown-estimation accuracy, this method also considers certain parallelism between multiple memory requests and between

processor and memory.

Li et al. [14] also proposed two memory adaptation algorithms, *Performance-Directed Static* (PS) and *Performance-Directed Dynamic* (PD) algorithms, that remove the necessity for painstaking parameter tuning and provide performance guarantees by periodically adjusting their decisions based on the available slack and recent workload characteristics. Both algorithms outperform the corresponding original dynamic/static algorithm with the best hand-tuned parameter setting. Comparing PS and PD, PD saves more energy for memory because it also exploits workload variability at fine time granularity. At every epoch (time interval), PD changes its thresholds based on the insight that the optimal thresholds are a function of the access traffic and the acceptable slowdown that can be incurred. It predicts the latter two quantities at epoch granularity. To overcome any errors in its predictions and heuristics, it self-corrects its adaptation for the next epoch based on its performance in the last epoch.

Delaluz et al. have also studied compiler-directed techniques [31,32] as well as operating-system-based approaches [33, 34] to reduce the energy consumed by the memory subsystem. Recently, Huang et al. [35] has proposed power-aware virtual memory implementation in OS to reduce memory energy consumption. But all of these works focus only on memory and do not provide any kind of performance guarantee.

4.3 Joint Adaptation

To our knowledge, there is no previous work that considers the joint adaptation of processor and memory while providing a performance guarantee. The only related work we are aware of is by Fan et al. [36]. This work studies a system with processor DVS and memory adaptation and shows that there is a positive synergistic effect between DVS and memory adaptation. But they study only multimedia workload and their work does not consider any architectural adaptation and is not performance-aware.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

This thesis proposes the first algorithm that can cooperatively adapt both processor and memory to save energy. We show that joint adaptation of both components provides significantly more energy savings than adapting either component alone. Further, our results illustrate the importance of cooperative adaptation, where the distribution of targeted slack among the processor and memory is determined in an intelligent and application specific way. Our algorithms also advance the state of the art in processor adaptation for energy. Most previous processor algorithms focused on saving energy without "much" performance loss, but without bounding the loss. We show how to trade off a targeted amount of performance to save energy – our algorithms not only save more energy but also do so by providing a performance guarantee. Finally, our processor algorithms adapt at multiple time scales, exploiting both short-term and long-term variability in application behavior.

There are several avenues for future work. First, our fine-grain or local adaptations are heuristic based, like previous work. These algorithms are difficult to tune. We would like to adapt the formal approach used in recent work [37] for multimedia applications to general-purpose applications as well. Second, currently we assume a user defined performance slack and seek to minimize energy while staying within this slack. In the future, we would like to explore explicit performance-energy tradeoffs, and the involvement of the operating system in determining application-specific tradeoffs that can maximize the utility of the entire system. Finally, we would also like to integrate dynamic voltage scaling techniques with the architecture adaptations studied here.

REFERENCES

[1] K. Govil, E. Chan, and H. Wasserman, "Comparing algorithm for dynamic speed-setting of a low-power CPU," in *Proceedings of the 1st Annual International Conference on Mobile Computing and Networking*, 1995, pp. 13–25.

[2] M. Neufeld, D. Grunwald, P. Levis, C. Morrey, and K. Farkas, "Policies for dynamic clock scheduling," in *Proceedings of the Fourth Symposium on Operating System Design and Implementation OSDI'2000*, October 2000, pp. 27–30.

[3] Y.-H. Lee and C. M. Krishna, "Voltage-clock scaling for low energy consumption in real-time embedded systems," in *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, 1999, p. 272.

[4] P. Pillai and K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, 2001, pp. 89–102.

[5] T. Pering, T. Burd, and R. Brodersen, "Voltage scheduling in the IpARM microprocessor system," in *Proceedings of the 2000 International Symposium on Low Power Electronics and Design*, 2000, pp. 96–101.

[6] D. H. Albonesi, "Selective cache ways: On-demand cache resource allocation," in *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, 1999, pp. 248–259.

[7] D. Folegnani and A. Gonzlez, "Energy-effective issue logic," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, 2001, pp. 230–239.

[8] S. Ghiasi, J. Casmira, and D. Grunwald, "Using IPC variation in workloads with externally specified rates to reduce power consumption," in *Proceedings of Workshop on Complexity-Effective Design*, June 2000, pp. 127–135.

[9] A. Iyer and D. Marculescu, "Power aware microarchitecture resource scaling," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2001, pp. 190–196.

[10] S. Manne, A. Klauser, and D. Grunwald, "Pipeline gating: Speculation control for energy reduction," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998, pp. 132–141.

[11] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller, "Energy management for commercial servers," *IEEE Computer*, vol. 36, no. 12, pp. 39–48, December 2003.

[12] Rambus, "Rdram," http://www.rambus.com, 1999.

[13] A. R. Lebeck, X. Fan, H. Zeng, and C. S. Ellis, "Power aware page allocation," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000, pp. 105–116.

[14] X. Li, Z. Li, F. David, P. Zhou, Y. Zhou, S. Adve, and S. Kumar, "Performance directed energy management for main memory and disk," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[15] M. Huang, J. Renau, S.-M. Yoo, and J. Torrellas, "A framework for dynamic energy efficiency and temperature management," in *Proceedings of the 33rd annual ACM/IEEE International Symposium on Microarchitecture*, 2000, pp. 202–213.

[16] M. C. Huang, J. Renau, and J. Torrellas, "Positional processor adaptation: Application to energy reduction," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003, pp. 157–168.

[17] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003, pp. 336–349.

[18] C. J. Hughes, P. Kaul, S. Adve, R. Jain, C. Park, and J. Srinivasan, "Variability in the execution of multimedia applications and implications for architecture," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, 2001, pp. 254–265.

[19] C. J. Hughes, J. Srinivasan, and S. V. Adve, "Saving energy with architectural and frequency adaptations for multimedia applications," in *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, 2001, pp. 250–261.

[20] R. Maro, Y. Bai, and R. I. Bahar, "Dynamically reconfiguring processor resources to reduce power consumption in high-performance processors," in *Proceedings of Workshop on Power-Aware Computer Systems*, 2000, pp. 97–111.

[21] R. Sasanka, C. J. Hughes, and S. V. Adve, "Joint local and global hardware adaptations for energy," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002, pp. 144–155.

[22] A. S. Dhodapkar and J. E. Smith, "Managing multi-configuration hardware via dynamic working set analysis," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 2002, pp. 233–244.

[23] W. Liu and M. C. Huang, "EXPERT: Expedited simulation exploiting program behavior repetition," in *Proceedings of the 18th Annual International Conference on Supercomputing*, 2004, pp. 126–135.

[24] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002, pp. 45–57.

[25] V. S. Pai, P. Ranganathan, and S. V. Adve, *RSIM Reference Manual, Version 1.0*, Rice University, 1997.

[26] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000, pp. 83–94.

[27] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," in *Proceedings of the 30th International Symposium on Computer Architecture*, 2003, pp. 336–349.

[28] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures," in *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, 2000, pp. 245–257.

[29] J. Pouwelse, K. Langendoen, and H. Sips, "Energy priority scheduling for variable voltage processors," in *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, 2001, pp. 28–33.

[30] M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for reduced CPU energy," in *Proceedings of 1st USENIX Symposium on Operating Systems Design and Implementation*, November 1994, pp. 13–23.

[31] V. Delaluz, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Energy-oriented compiler optimizations for partitioned memory architectures," in *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, November 2000, pp. 138–147.

[32] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramniam, and M. J. Irwin, "Hardware and software techniques for controlling dram power modes," *IEEE Transactions on Computers*, vol. 50, pp. 1154–1173, November 2001.

[33] V. Delaluz, M. Kandemir, and I. Kolcu, "Automatic data migration for reducing energy consumption in multi-bank memory systems," in *The 39th Design Automation Conference*, June 2002, pp. 213–218.

[34] V. Delaluz, A. Sivasubramaniam, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Scheduler-based dram energy management," in *Proceedings of the 39th Conference on Design Automation*, 2002, pp. 697–702.

[35] H. Huang, P. Pillai, and K. G. Shin, "Design and implementation of power-aware virtual memory," in *Proceedings of the 2003 USENIX Annual Technical Conference*, June 2003, pp. 57–70.

[36] X. Fan, C. S. Ellis, and A. R. Lebeck, "The synergy between power-aware memory systems and processor voltage scaling," in *Proceedings of the Workshop on Power-Aware Computer Systems PACS'03*, December 2003.

[37] C. Hughes and S. Adve, "Spreading slack for optimal energy-performance tradeoffs for multimedia applications," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2004.