# POWER MANAGEMENT TECHNIQUES FOR CONSERVING ENERGY IN MULTIPLE SYSTEM COMPONENTS

by

**Neven Abou Gazala**

B.E., Arab Academy for Science and Technology, 1996

M.E., Arab Academy for Science and Technology, 2000

M.S., University of Pittsburgh, 2003

Submitted to the Graduate Faculty of
the Department of Computer Science in partial fulfillment
of the requirements for the degree of

**Doctor of Philosophy**

University of Pittsburgh

2008

UNIVERSITY OF PITTSBURGH

DEPARTMENT OF COMPUTER SCIENCE

This dissertation was presented

by

Neven Abou Gazala

It was defended on

February 7th 2008

and approved by

Rami Melhem, Department of Computer Science, University of Pittsburgh

Daniel Mossé, Department of Computer Science, University of Pittsburgh

Bruce Childers, Department of Computer Science, University of Pittsburgh

Mary Jane Irwin, Department of Computer Science and Engineering, Penn State University

Dissertation Advisors: Rami Melhem, Department of Computer Science, University of Pittsburgh,

Daniel Mossé, Department of Computer Science, University of Pittsburgh

# ABSTRACT

# POWER MANAGEMENT TECHNIQUES FOR CONSERVING ENERGY IN MULTIPLE SYSTEM COMPONENTS

Neven Abou Gazala, PhD

University of Pittsburgh, 2008

Energy consumption is a limiting constraint for both embedded and high performance systems. CPU-core, caches and memory contribute a large fraction of energy consumption in most computing systems. As a result, reducing the energy consumption in these components can significantly reduce the system's overall energy consumption. However, applying multiple independent power management policies in the system (one for each component) may interfere with each other and in some occasions increase the combined energy consumption.

In my thesis, I present three power management techniques that target more than a single component in a system. The focus is on reducing the total energy consumption in processors, caches and memory combinations. First, I present a memory-aware processor power management using collaboration between the OS and the compiler. The technique objectives are: (1) finish the application execution before its deadline and (2) minimize the combined energy consumption in processor and memory. Second, I present an integrated-DVS technique for processor and on-chip cache power management in multi-voltage domain systems. Integrated-DVS co-ordinates power management decisions across voltage domains rather that being applied in isolation in each domain. Third, I present a Power-Aware Cached DRAM (PA-CDRAM) memory organization for reducing the energy consumption in DRAM memory and off-chip caches. PA-CDRAM exploits the high internal memory bandwidth by bringing the off-chip caches "closer" to the memory, which improves both the overall performance and energy consumption.

The techniques in my dissertation highlight the importance of designing power management schemes that consider multiple components and their interactions (in terms of power and performance) in the system rather than applying multiple isolated power management polices. This study should lay the foundation for further research in the domain of integrated power management, where a single power manager controls many system components.

## TABLE OF CONTENTS

# LIST OF TABLES

## LIST OF FIGURES

# 1.0 INTRODUCTION

Energy consumption is a limiting constraint for both embedded and high performance systems. In embedded systems, the operational lifetime of a device is limited by the rate of energy dissipation from its battery. On the other hand, energy consumption in high-performance systems increases thermal dissipation, which requires more cooling resources, and thus higher system management overhead. Among the major energy consumers in computing systems are the CPU and memory subsystems. In high performance systems such as servers, CPU and memory consume up to 50% [3] and 41% [4], respectively; while in portable devices they consume 26% and 23%, respectively [5].

The continuous demand for more computing power motivates the design of more sophisticated processors. With the increase in system's computing capabilities, power dissipation is expected to rise proportionally in these systems [6]. Moreover, increasing the computing capability would potentially increase workload on the memory system, thus increasing its power dissipation as well. Accordingly, this increase in power dissipation motivates the need for more efficient system-wide power management schemes.

Reducing the energy consumption in computing systems has become a prime design criterion–motivated by the limited lifetime of battery operated systems, and the high electrical and cooling power costs in server farms. Processor and memory subsystems contribute a large fraction of the overall system power dissipation. The fraction of the processor energy compared to the memory energy varies based on the system parameters (memory size, processor core complexity) and on the application behavior (memory versus CPU intensive). As a result, to achieve overall low system energy, choosing an appropriate power management technique should be based on the fraction of the power consumed by each.

Applying power management techniques often involves a trade-off between performance and energy consumption. Thus, a power management scheme should consider this trade-off in addition to existing application and system performance constraints. Application performance constraints can be a result of a limitation in the available time for application execution, as in *real-time systems*. System constraints are the result of performance bottleneck(s) in one or more of the system components (as in memory and I/O subsystems).

A major constraint imposed on computing systems is the increasing speed gap between processor and memory. This speed difference limits the overall system performance. The main contributors to this problem are the large memory access time and the limited transfer bandwidth between the memory and the CPU [7]. Memory architectures aim at decreasing the access time by using faster clocks and increasing the level of concurrency in data accesses. However, the bus transmission speed is becoming a significant bottleneck limiting the memory system performance [8]. Thus, systems —especially with large memory traffic— face the problem of minimizing this speed gap to achieve high performance while consuming less energy.

Independently managing power in one system component while neglecting other component's power may not necessarily achieve lower combined energy consumption for the entire system. For example, slowing down the processor to save its energy increases the execution time and causes memory to retain data for longer periods, thus consuming more memory's energy. Hence, an effective power management scheme should consider multiple system components for optimizing the system's total energy consumption.

## 1.1 PROBLEM STATEMENT

Power dissipation in CPU, caches and memory accounts for the majority of total power consumption in most computing systems. Components that consume large percentage of the overall system's energy consumption are good targets for power management.

The objective of applying power management policies is to ideally optimize the overall system energy consumption rather than reduce energy consumption of one component. The majority of power management policies proposed in the literature address the energy problem for one system component. While these policies can locally minimize energy consumption for a component, being oblivious to the overall system power may adversely affect other system components by increasing their energy consumption. As a result, the system's overall energy savings is lower. For example, dynamic voltage scaling in processors slowdown execution of a process. Accordingly, this increases energy consumption in the memory system, and the I/O devices associated with a process due to the extension of their active periods. Thus, to achieve higher energy savings for the entire system, the power management technique should take into account the resulting effects on the other components in the system or even target more than one system component simultaneously.

We address the problem of achieving efficient power management in more than one system component simultaneously. The main focus of our study are CPU, caches and main memory energy consumption. We propose three power management techniques with the objective of reducing the combined energy consumption in multiple components with minimal impact on performance.

## 1.2 CONTRIBUTIONS

In this dissertation, we present three power management techniques. The common feature among the techniques is that they reduce the combined energy consumption of at least two system components. Each of these techniques addresses a particular class of systems.

For systems with dominant processor power consumption, we propose a scheme that primarily targets the CPU energy consumption without consuming excessive memory energy. We approach this problem in the context of real-time systems where timing constrains are critical. For systems with comparable processor and cache energy consumption, we propose a second scheme that manages both the CPU and the on-chip caches simultaneously. This technique is feasible in modern chips with multiple clock and voltage domains, where the clock and voltage of each domain is controlled independently. The third scheme targets the memory and off-chip cache energy consumption while improving the overall system performance. This technique is intended for systems where memory energy dominates the total system energy. Figure 1.1 summarizes the focus and domain of each of the three proposed techniques. A brief overview of each scheme is presented next.

### 1.2.1 Memory-aware processor power management

Dynamic Voltage Scaling (DVS) is an effective technique for reducing power dissipation in processors. DVS slows down the processor's speed to lower its energy consumption. Because some applications must finish by a certain deadline, the objective of a power management scheme employing this technique is to efficiently use the available slack (time when processor is idle) to reduce

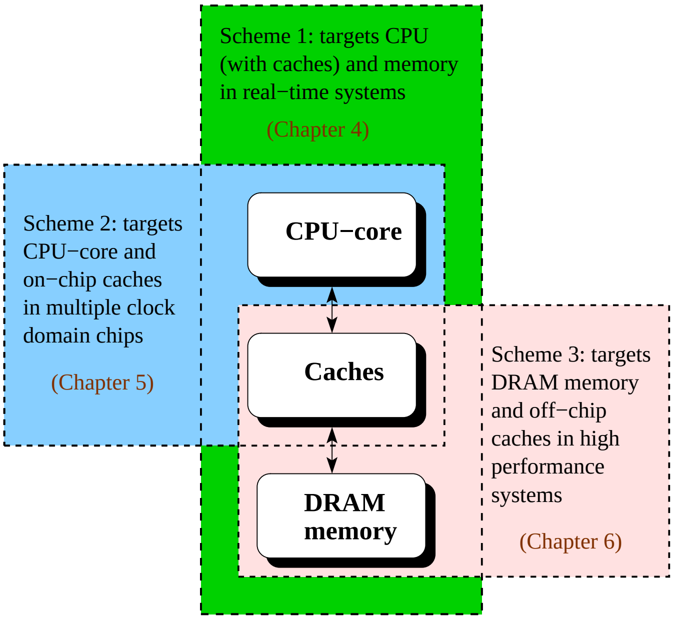

Scheme 1: targets CPU (with caches) and memory in real−time systems
(Chapter 4)
Scheme 2: targets CPU−core and on−chip caches in multiple clock domain chips
(Chapter 5)
CPU−core
Caches
DRAM memory
Scheme 3: targets DRAM memory and off−chip caches in high performance systems
(Chapter 6)

Figure 1.1: Proposed power management techniques targeting more than one system component.

the processor's frequency without violating an application's timing constraints. More slack utilization implies more slowdown and thus more processor energy savings. On the other hand, excessive slowdown in processor speed increases the time where the memory stays active, which increases the system's overall energy. Thus, finding an efficient operating frequency that balances these trade-offs is essential to reduce the system's overall energy.

To maximize the amount of detected slack in the system, we take advantage of dynamic slack, which is the slack resulting from applications running for shorter times than their worst case execution times. This is different than static slack which is known offline (the difference between the worst case execution time and an application's deadline). To detect and utilize dynamic slack, we use the compiler knowledge of the state of execution of an application. The compiler inserts special hints in the application code. These hints are capable of computing–at run-time– the worst case remaining cycles for the application to finish execution at each hint location. At run time, information from the hints is passed to the operating system to periodically schedule the proper speed based on the combined available static and dynamic slacks. To avoid excessive slowdown of the application's execution, and hence increasing the memory's energy, we compute a cut-off frequency below which the processor should not operate. This cutoff frequency represents the break-even point where further slowdown causes the energy savings in the processor to be less than the additional energy consumed in memory. We model the processor and memory energies to compute this cut-off frequency based on processor and memory parameters.

The main contributions of this work are: (1) describe an approach where the OS and compiler collaborate to extract information useful for building more efficient power management policies, and (2) determine the cut-off frequency beyond which the overall system energy increases despite decrease in the CPU energy. More details on this work in presented in Chapter 4.

### 1.2.2 Integrated DVS policies for CPU and cache power management

When the fractions of power consumed by the processor and memory are comparable, we should target both components simultaneously, focusing on the interplay of power management in both subsystems. In this scheme, we are especially concerned with the processor and the on-chip caches.

In multiple clock and voltage domain chips, we propose an integrated technique for controlling the voltage/frequency of each of the domains. The integrated technique is more efficient than its local counterpart because it considers interactions among domains. In our proposed policies, we focus mainly on the CPU-core and L2 cache domains. We first propose a generic online heuristic-based integrated DVS policy that suits most application classes. We then proposed a machine learning approach that generates DVS policies optimized for each class of applications.

This work shows that integrated DVS policies are more energy efficient than local DVS policies that target different domains in isolation. This efficiency is primarily due to the global knowledge of the workload and power dissipation in different chip domains. That global knowledge allows the policy to make better informed decisions with respect to selecting the best speeds for each domain. More details on this work in presented in Chapter 5.

### 1.2.3 Power-aware Cached DRAM

Dynamic power management in memory can indirectly influence the processor's energy consumption when applications experience performance degradation due to increased average memory access latency. This increase is a result of extra delay experienced from transitioning the memory to low power states. Longer execution time implies that the processor resources have to be active for longer periods, thus consuming more energy.

On the other hand, memory has a huge internal bandwidth compared to its external bus bandwidth. To exploit the wide internal bus, cached DRAM (CDRAM) adds an SRAM cache to the DRAM array on the memory chip. Such a near-memory cache acts as an extra memory hierarchy level, whose fast latency improves the average memory access time and potentially improves system performance. However, proposed CDRAM organizations do not necessarily optimize the system energy due to the extra energy consumed in the near-memory caches.

In our work, we investigate the effect of placing a subset of the total cache capacity closer to the memory rather than closer to the CPU. We are especially interested in overall performance and energy consumption. We optimize the CDRAM organization for energy efficiency. We integrate a moderately sized cache within the chip boundary of a power-aware multi-banked memory. We call this organization power-aware cached DRAM (PA-CDRAM). In addition to improving performance, PA-CDRAM significantly reduces energy consumption in caches and in main memory. This is due to (1) using small caches distributed to the memory chips reduces the cache access energy compared to using a large non-distributed cache, (2) near-memory caches allow the access of relatively large blocks from memory, which is not affordable with near-processor caches, and (3) memory energy consumption is reduced by having longer memory idle periods during which DRAM banks can be powered off. PA-CDRAM improves the original CDRAM by tackling the interplay of the cache and memory organizations to optimize the memory's performance and energy consumption. The contribution of this part of the work is twofold. First, we propose near-memory caches for energy reduction of the overall system. Second, we describe an implementation of PA-CDRAM that integrates a near-memory cache in a Rambus chip (RDRAM). We also describe how our implementation can maintain backward compatibility with existing Rambus memories. More details on this work in presented in Chapter 6.

### 1.2.4 Summary of contributions

To summarize, the contributions of this dissertation work are:

- We present a collaboration scheme between operating system and compiler for a memory-aware CPU power management technique for real-time systems. This technique is more efficient than a compiler-only and OS-only schemes because it can extracts more information about the system and use it more efficiently.
- We propose a class of integrated DVS techniques for reducing energy consumption in both CPU-

core and L2 caches for multiple clock domains. The integrated DVS technique is more energy efficient than local DVS techniques due to its global knowledge of both power and activity of the target components.

- We devise an architectural optimization for improving the energy consumption in DRAM memory and off-chip caches, called PA-CDRAM, in high performance systems. PA-CDRAM improves both overall performance and energy consumption over traditional memory hierarchy.

## 1.3 THESIS ROADMAP

The remainder of this dissertation is organized as follows. Chapter 2 reviews background and some related work to power management techniques that target CPU, caches and memory energy consumption. Chapter 3 introduces the main models used throughout this work. Our proposed policy for memory-aware CPU power management is presented in Chapter 4, followed by techniques for integrated CPU and cache power management in Chapter 5. Our technique for improving performance and reducing energy consumption in DRAM memory and off-chip caches is presented in Chapter 6. Chapter 7 summarizes and concludes our thesis.

# 2.0 BACKGROUND AND RELATED WORK

## 2.1 CPU POWER MANAGEMENT USING DVS

An efficient power saving technique for processors dynamically changes the CPU supply voltage according to current workload. This technique is called *dynamic voltage scaling (DVS)*. Reducing voltage in CMOS circuits reduces the processor's power consumption quadratically. Because the processor clock frequency is dependent on the supply voltage, reducing the voltage causes a program to run slower. However, this slowdown is linear with the supply voltage. Since, the dynamic energy consumed by an application is the product of the CPU power and time spent running an application, running a program with reduced voltage and frequency leads to significant energy savings.

DVS techniques can be classified into schemes that use information about tasks' deadlines and schemes that maintain an acceptable performance degradation to achieve significant energy savings. Orthogonally, speed scheduling in DVS algorithms can be controlled by either the OS or the compiler/application. Table 2.1 lists some power management techniques based on this classification.

Table 2.1: Examples on CPU power management schemes

| | application w/o deadlines | applications w deadlines |
|---|---|---|
| OS directed | Childers et al. [9], ECOSystem [10], Vertigo [11, 12] | Mossé et al. [13], Gruian [14], PACE [15], Pillai et al. [16], Aydin et al. [17], Pering et al. [18] |
| Compiler directed | Hsu el al. [19, 20] | Azevedo et al. [21], Shin et al. [22], Saputra et al. [23] |

For systems with no strict time constraints, the operating system periodically collects information about the system and adjusts the speed accordingly. Childers et al. [9], periodically invokes an interrupt service routine that adjusts the speed to maintain a desired performance goal. The OS keeps track of the accumulated application's instruction level parallelism throughout the application execution time. The ECOSystem framework [10] allocates energy budget per resource to limit the battery energy dissipation per period. Flautner et al. [11, 12] classified execution intervals as interactive, periodic producer, and periodic consumer to derive potential deadlines. The derived deadlines guide the speed setting in each scheduling interval.

Power-aware compilation performs specialized code optimizations to assign lower speeds to selected code segments. Hsu el al. [19, 20], identifies program regions where the CPU can be slowed down. The compiler sets the speed in each region based on the expected time the CPU would wait for a memory access. However, this work does not exploit dynamic slack time generated in computation-dominated applications.

Time restrictions in real-time applications mandate the processor to finish the application's execution before its deadline. On the OS level, CPU speeds are set using knowledge about a task execution time profile. Mossé et al. [13] present a number of dynamic speed-setting schemes that reclaim both static and dynamic slack. Gruian [14] determines a voltage schedule that changes the speed within the same task/application based on task's statistical behavior. Similarly, in PACE [15], a task's speed increases as it's execution progresses over time based on probability distribution of its execution times. Pillai et al. [16] presented power management heuristics that modify EDF and RMS scheduling in RT-Linux to incorporate voltage scaling. Aydin et al. [17] determine the optimal static speed for periodic real-time tasks with different power characteristics. Pering et al.[18] presented one of the early DVS implementation on a CPU scheduler to target real-time system.

Compiler controlled DVS techniques in real-time systems analyze an application and insert code offline that controls the CPU speed at run-time. Azevedo et al. [21] insert checkpoints at the start of each branch, loop, function call, and normal segment. Information about the checkpoints along with profile information are used to estimate the remaining cycle count and hence compute a new frequency. Shin et al. [22] augment each basic block in a CFG with its worst case execution cycles along with the remaining worst case cycles till the end of the program. Speed-change code is inserted in selected branches. The branch is selected if the remaining cycles at this branch is smaller than the worst case remaining cycles at a sibling branch. Saputra et al. [23] select the best supply voltage for each loop nest based on the loop's estimated energy. The voltage levels are set at compile time for each region using an integer linear programming DVS strategy. Kim et al. [24] presented a comparison of Shin's and Gruian's algorithms on the same simulation environment. They showed that the performance of these two schemes are quite different depending on the available slack times.

My proposed CPU power management scheme falls in to the class of schemes that target applications with deadlines. OS-directed schemes predict the remaining workload based on overall worst-case and average-case execution time estimates. However, the collaborative scheme estimates the remaining execution time based on knowledge about the current execution instance of an application. This knowledge provides more accurate estimates that can be used in scheduling the CPU speed.

Compared to the aforementioned compiler-directed schemes, the key advantage of the collaborative scheme is the use OS's runtime knowledge in computing the CPU speed while accounting for overheads. In Azevedo et al. [21], run-time overhead of updating data structures and setting the new voltages is relatively high especially on the nodes granularity assumed (almost every basic block). The collaborative scheme uses a much larger granularity than a basic block to schedule the CPU speed with minimal overheads. In Shin et al. [22], profiled timing information and speed change decisions are hard coded in the selected branches. However, part of the dynamic slack is wasted due to the deviation of the actual execution time from the profiled information used to set the speed. In contrast, the collaborative scheme can account for this slack by calculating the speed at run-time. Since Saputra et al. [23] assign lower voltages to loops only, their scheme is best suited for media applications where the loop dominates the execution of an application, while my proposed scheme fits general purpose applications.

## 2.2 CACHE POWER MANAGEMENT

The current increase in the number of cores per chip requires a similar increase in on-chip cache capacities. Larger caches provide faster access to the data coming from multiple running threads on these cores. As a result, on-chip caches occupy large chip area and consume a large percentage of the total chip power. Efficient cache management and organization improve the cache access latency and power consumption. Examples of techniques that can optimize cache performance and energy are cache partitioning, turning off unused sections in the cache, and tag-array optimizations.

Cache partitioning reduces both the per-access energy and latency due to activating and accessing small portions of the cache rather than the entire cache. Accessing a portion of the cache involves activation of shorter bit and word lines. Kim et al. examine ways of splitting the cache into smaller units, each of which is called a sub-cache [25]. They selectively activate a target cache upon a memory access. In CMP processors, a cache partitioning manager can allocate cache sizes according to the demands of the running threads [26, 27, 28]. Ravindran et al. use the compiler to analyze applications' cache access characteristics, and accordingly determine the cache partitioning strategy [26]. The compiler inserts hints to control cache lookup and data placement of each partition. Data is accessed from partitions as indicated by a bit-vector associated with each Load/Store instructions.

In conjunction with partitioning, parts of the cache can be set at a lower power state or even turned off. Putting unused cache portions into lower power states is beneficial in reducing the cache's leakage energy with insignificant impact on performance. Patel et al. exploits the spatial/temporal properties of data caches to store high locale data in a separate small memory while turning off the cache lines belonging to this data [29]. Kadayif et al study the relationship between prefetching and the turnoff of cache lines to save leakage energy [30, 31].

Since the cache's tag array is accessed at every cache access, reducing per-access energy in the tag array causes significant decrease in the cache energy consumption. Zhou et. al. propose a cache architecture which eliminates the majority of references of TLB lookups by storing both virtual and physical tags [32]. For I-cache, Petrov et al. proposes a software-directed technique that reduces the number of tag bits lookup for instructions in the most frequently executed loops [33]. Loghi et al. propose reducing the number of tag bits by moving a large number of tag bits to an external register [34].

## 2.3 POWER MANAGEMENT IN MULTIPLE CLOCK DOMAINS CHIPS

With the increase in number of transistors and reduced feature size, higher chip densities create a problem for clock synchronization among chip computational units. With a single master clock for the entire chip, it is harder to design a clock distribution network that limits clock skew in the different chip components. Several solutions have been proposed to this problem using globally-asynchronous locally synchronous (GALS) designs. In a GALS design, a chip is divided into multiple clock domains (MCD). Each domain operates synchronously with its own clock, and communicates

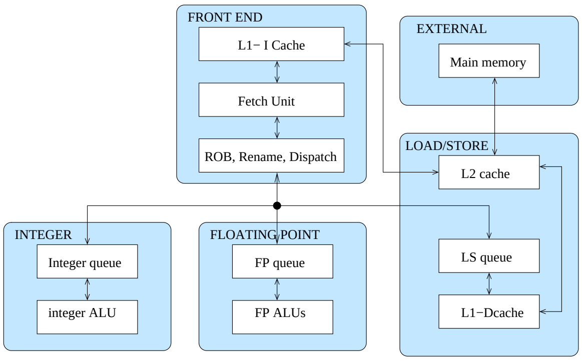


Figure 2.1: Example of domain partitions in MCD processors [1, 2]

with other domains asynchronously through FIFO queues.

MCD design increases the opportunity for better power management with dynamic voltage and frequency scaling (DVS). Since each domain maintains its own clock and voltage independently of other domains, DVS can be applied in each domain for an extra level of power management (rather than applying DVS at the chip level). Power and energy benefits come from dynamically adjusting an individual domain's clock and voltage according to its activity.

Several power management policies have been proposed to incorporate DVS into MCD chips. For example, an online power management policy that monitors queue occupancy of each domain and adapts the domain voltage accordingly has been proposed [1, 2]. Figure 2.1 shows the domains of the proposed MCD chip. For each domain, the policy computes the change in the average queue length in consecutive intervals. Increasing queue length triggers higher voltage and clock settings for a domain and vice versa. Results show a significant power and energy improvement. In general, policies in the literature focus on each domain in *isolation* without considering possible inter-domain effects when varying clock/voltage.

MCD design has the advantages of alleviating some clock synchronization bottlenecks and reducing the power consumed by the global clock network. Semeraro et al. explored the benefit of voltage scaling in MCD versus globally synchronous designs [35]. They find a potential 20% average improvement in the energy-delay product. Similarly, Iyer at al. analyzed the power and performance benefit of MCD with DVS [36]. They find that DVS provides up to 20% power savings over an MCD core with single voltage.

In industrial semiconductor manufacturing, National Semiconductor in collaboration with ARM developed PowerWise technology that uses Adaptive Voltage Scaling and threshold scaling to automatically control the voltage of multiple domains on chip [37]. The PowerWise technology can support up to 4 voltage domains [38]. Their current technology also provides power management interface for dual-core processors.

Another technique by Magklis et al. is a profile-based approach that identifies program regions that justify reconfiguration [39]. This approach involves extra overhead of profiling and analyzing phases for each application. Zhu et al presented architectural optimizations for improving power and reducing complexity [40]. However, these policies do not take into account the cascading effect of changing a domain voltage on the other domains.

Wu et al. present a formal solution by modeling each domain as a queuing system [41]. However, they study each domain in isolation and incorporating domain interactions increases the complexity of the queuing model. Varying the DVS power management interval is another way to save energy. Wu et al. adaptively vary the controlling interval to react to changes in workload in each domain [42]. They do not take into account the effect induced by voltage changes in one domain on the other domains.

MCD design can be applied to multicore and simultaneous multithreading processors such [43, 44, 45]. In [43, 44], each core has its own clock network, and the DVS policy independently controls each core's voltage. Lopez et al. studies the trade-off between adapting the L2 cache capacity and speed based on the number of active threads in the core domain [45].

## 2.4 DRAM POWER MANAGEMENT

Current memory organizations, such as SDRAM and Rambus, allow managing energy consumption by setting the DRAM chips to one of two or more power states [46]. DRAM chips are set at a low power (sleep) state during idle periods and transitioned to a high power state (active) to service memory requests. During a read/write transaction, *all* chip components (row decoder, clock, etc.) are powered up to service requests. Powering down some of these components creates power-saving states. Delay penalties result from transitioning between power states before servicing a memory request. A common technique of scheduling a chip's power state transitions is through a *time-out* policy. A power management algorithm defines a time-out threshold such that, when a memory idle period exceeds this threshold, the chip is transitioned to a lower power state.

At the micro-architecture level, the use of special purpose memory like *Streaming Memory*, *Scratch-Pad Memory*, or *Energy-Saver Buffers* can reduce the memory's energy by accessing the most frequent data from these special low power memories [47, 48]. Fan et al. [49, 50] analyzed different memory controller policies and their effect on the DRAM power. They found that immediate shutdown of the memory chips yields better results than sophisticated adaptive techniques. Memory and disk controller policies can apply similar power management strategies [51, 52].

In the OS layer, Lebeck et al. [53] proposed the use of a power aware allocation policy where data is allocated sequentially in each memory bank to increase the bank idle periods. An implementation of the memory power manager in the Linux operating system [54] allocates memory pages to banks grouped based on the allocating process. Li et al. [51] presented a technique that provides performance guarantees on the memory system by periodically adjusting the time-out threshold based on the available slack and the current workload. Cai et al. [52] uses an interval based scheme that adapt the memory page size and the disk time-out threshold based on the system workload.

Other research direction targets compiler techniques for memory power management. Delaluz et al.[55, 56] group the allocation of the requested pages in memory based on application's order of data accesses. Ozturk et al. [57] modify the data access pattern of a performance-oriented scheme to reduce leakage and dynamic energy in the memory hierarchy. Grun et al. [58] clustered variables into memory banks based on their spatial and temporal locality. Kandemir et al. [59] presented a code optimization technique to manage the flow of data to/from a scratchpad memory.

My proposed memory power management scheme compares to schemes implemented in the architecture and the OS levels. While the use of extra on-chip memory (as with energy-saving buffers and streaming memories) reduces the activation of the DRAM chips, the aggressive prefetching creates a memory bus performance bottleneck and increases the bus's energy consumption. These factors were not accounted for in the above related work. In comparison, my scheme uses near-memory caching (within the DRAM chips) to avoid such a bottleneck. With respect to adaptive OS techniques that vary the time-out threshold, they mainly optimize the energy-delay product without providing bounds on the delay experienced during this adaptation (except in [51]). However, we propose to maintain the memory delays within specified bounds. In contrast to [51], we look for adapting more reconfigurable system parameters, such as the cache block size, in conjunction with the time-out threshold.

## 2.5 EMBEDDED DRAM

Integrating DRAM and logic cells on the same chip is an attractive solution to achieve both high performance (from logic cells) and high memory density (from DRAM cells). This integration avoids the high latency of going off-chip by doing computation (or even caching) at the memory itself. Currently, manufactured chips with embedded DRAM and logic are mainly used in applications like computer graphics, networking, and handheld devices [60]. Based on the fabrication technology (either DRAM-based or logic-based), some degradation to the speed (density) of the logic (DRAM) cells may occur. For example, in early DRAM-based chips, logic cells were reportedly slower by 20% to 35% [60]. However, emerging fabrication technologies aim at overcoming these penalties. For example, NEC's embedded DRAM chips offer DRAM-like density with SRAM-like performance [61], and IBM's third generation embedded DRAM chips support two embedded DRAM families for high density and high performance to serve both purposes with no degradation [62].

# 3.0 MODELS

## 3.1 REAL-TIME APPLICATIONS

Real-time applications are characterized by deadlines. A *deadline* is the amount of time available for an application to finish execution. The deadline should be larger than or equal to the application's worst case execution time. Since the execution time for an application may vary in a system with multiple operating frequencies, the execution time is expressed in cycles rather than time units. Thus, the application's duration is represented by its *worst case execution cycles* (WCC). We define an application *worst case execution time* (WCET) as the time spent executing the WCC at the maximum frequency. We consider two types of real-time applications: applications subject to hard deadlines and others subject to soft deadlines. It is *critical* for applications with hard deadlines to meet these deadlines. However, it is *desirable* for an application with a soft deadline to finish execution within the allowed deadlines. That is, in contrast to hard deadlines, it is acceptable for applications with soft deadlines to endure performance degradation in excess of their deadlines in order to achieve higher energy savings.

When running at the maximum frequency, if $deadline > WCET$ (i.e., the allotted time exceeds the WCET), the time difference is known as *static slack*. During actual execution, an application runs for its *actual execution time* (ACET), which is smaller than or equal to its WCET. The difference between WCET and ACET is called the *dynamic slack*, which usually comes from data dependencies that cause program instances to execute different paths. Thus, it is safe to represent application duration by its *worst case cycles* (WCC). Similarly, ACC is used to represent the actual number of cycles spent executing an application.

We consider the general form of real-time applications that may consist of sequential code, branches, loops and procedures. A program is represented by a *control flow graph* (CFG). CFGs are used as an intermediate representation in compilers. CFG consists of *basic blocks* (code segment with one entry point, exit point and no jump instructions contained within it).

## 3.2 CPU ENERGY MODEL

In CMOS circuits, energy dissipated is the sum of three main components: *dynamic*, *static* and *leakage* power. Dynamic energy is a function of the processor's switching activity, operating frequency and voltage. Dynamic power is directly proportional to the square of the input voltage: $P_{dyn} \; \alpha \; CV_{dd}^{\beta}f$, where $C$ is the switched capacitance, $V_{dd}$ is the supply voltage, $f$ is the operating frequency and $\beta$ is a processor dependent constant that is typically greater than or equal to two. Static energy is the energy consumed due to powering up some of the core's structures like the register file and the load/store queue. This energy is consumed independent of any activity in the core. Leakage energy is the energy consumed during processor's idleness as a result of the leakage current flow in the transistors. Leakage energy is a function of both the gate supply and threshold voltages.

Most commercially available processors support Dynamic Voltage Scaling (DVS) capabilities to control the trade-offs between performance and energy consumption. Two examples are the Transmeta Crusoe TM5400 [63] and the Intel XScale [64]. In this work, we consider realistic processors with discrete voltage and frequency levels. We refer to changing the voltage/frequency and speed changes interchangeably.

When computing and changing the CPU speed, two main sources of overhead may be encountered depending on the CPU architecture: (1) computing a new speed, and (2) setting the speed through a voltage transition in the processor's DC-DC regulator (resulting in a processor frequency change) and the clock generator. The speed change overhead takes between $25\mu$sec to $150\mu$sec and consumes energy in the range of micro Joules (for example $4\mu$J in *lparm*) [65, 18]. We assume that the overhead of changing the voltage is constant for all the power level transitions in a given processor, whereas the overhead of computing the speed depends on the CPU operating frequency during each computation.

## 3.3 CACHE ENERGY MODEL

Access energy and latency for each cache access is a function of the cache configuration. For a given technology, the key parameters for cache configuration are cache size, associativity, and cache block size. Varying these parameters affects the length of the word and bit lines that need to be activated at each cache access. Longer lines causes longer delay and higher power consumption per access. We use the Cacti 3.0 [66] simulator to obtain the access energy and latency for the different configurations of caches used in our work. In Cacti, the per-access energy and latency is divided into portions consumed in the tag-array and the data-array (including sense amplifiers and output latches). In n-way set associative caches, tag and data arrays are accessed concurrently to reduce the total access time. In fully associative caches, the tag array is replaced by a fully associative decoder, after which the data array is accessed. Tag comparison takes place in the decoder. Then, the decoder drives the wordline associated with the cache entry. The serial access of tag and data arrays reduces the cache per-access energy; however, it increases the per-access latency. Using Cacti, we obtain access latencies and energy assuming that the cache is operating at voltage $V_{dd}$ = 1.3 V.

When integrating SRAM cache and DRAM memory on the same chip as discussed in Section 2.4, we add a delay penalty ranging from 10% to 35% for accessing logic cells in the memory chip [60]. We also account for delay penalties for accessing off-chip caches.

## 3.4 MEMORY ENERGY MODEL

External DRAM memory usually consists of multiple memory chips. A chip contains one or more memory banks. Each chip can be transitioned independently to/from a low power memory state. A chip consists of control logic (for example, row and column decoders) and data array (memory banks). Energy in the DRAM-core is consumed during read/write activity ($E_{dyn}$) and during idle periods ($E_{static}$). The memory's dynamic energy is consumed in address decoding, $E_{addr\_decode}$, and data transfer to/from DRAM-core. At each miss in the lowest level cache, the memory controller decodes the row and column addresses then precharges a row in the desired bank containing the requested data. Data access energy is proportional to the number of bytes transferred during read or write transactions. During inactivity (idle periods), the memory's idle energy is a function of the idle time spent at each power state. We assume five DRAM power states: active, standby, nap, sleep, and off [67]. Transition energy, $E_{trans}$, is consumed when a chip transitions to/from the active state. Thus, the energy consumed in the DRAM-core $E_{tot}$ equals:

$$E_{tot} = d\ E_{addr\_decode} + c\ P_{access}\ t_{access}\ +\ \sum_{s} P_s\ t_s\ +\ r\ E_{trans}$$

where $d$ is the number of data requests; $c$ is the number of transferred bytes; $P_{access}$ is the power dissipated during read/write transactions. $P_s$ is the power consumed when the DRAM is in the $s^{th}$ state. $t_{access}$ and $t_s$ are the time for transferring a single byte to/from DRAM-core and the time spent in the $s^{th}$ state, respectively; and $r$ is the number of transitions to and from the active state. $E_{addr\_decode}$, $P_{access}$, $t_{access}$, $P_s$, and $E_{trans}$ are memory specific parameters while $d$, $c$, $t_s$, and $r$ are factors of the application's memory access pattern and the memory hierarchy configuration.

# 4.0 COMPILER AND OS COLLABORATION FOR CPU AND MEMORY POWER MANAGEMENT

Dynamic Voltage Scaling (DVS) slows down the processor speed to reduce its power consumption. The objective of a power management scheme employing DVS is to efficiently use the available slack (time where the processor is idle) to reduce the processor's speed without violating applications time constraints. Better slack utilization implies a decrease in processor energy consumption but an increase in other system components.

Dynamically changing the speed in real-time applications (tasks) is classified into two categories: *inter-task* and *intra-task* voltage scaling [24]. Inter-task DVS schemes schedule speed changes when a task starts or finishes execution, while intra-task schemes schedule speed changes during task execution. In this chapter, we describe an intra-task DVS scheme, where voltage scheduling points (also called *power management points, PMP*) are invoked throughout an application's execution to change the CPU speed.

Considering the general form of a real-time application code, automatically deciding on the proper location to invoke PMPs using intra-task DVS scheduling is not trivial. One problem is how frequently the power manger should change the speed. Ideally, the more voltage scaling invocations, the more fine-grain control the application has for exploiting dynamic slack and reducing energy. However, in practice, the energy and time overhead associated with each speed adjustment can overshadow the DVS energy savings.

To design an efficient intra-task DVS scheme, a PMP should have enough information about the application to select the most energy-efficient speed level. With knowledge about the application execution, a PMP estimates the amount of work remaining to execute versus the time available before a deadline. The proper estimation by PMPs of the remaining workload saves energy by selecting the slowest speed that guarantees meeting the application's deadline.

The contribution of this chapter is threefold. First, we present a novel technique that involves the collaboration between the OS and the compiler to collect both run-time information and path-dependent information. Second, we show the effect of reducing the CPU frequency on increasing the memory energy consumption. We present speed scheduling schemes that compute the proper CPU frequency such that the sum of the CPU energy and the memory energy consumption is minimized. Third, we evaluate our technique on time-sensitive applications running on two commercially available processors with dynamic voltage scaling. We show that our technique can achieve significant energy reduction over no power management, OS-directed and compiler-directed power management techniques.

Next, we present a brief description of the power management approach that deals with these issues. We start with a discussion of the idea behind our OS-compiler collaborative approach in Section 4.1 and highlight its advantages over OS-only or compiler-only approaches in Section 4.2. A brief overview of the technique is presented in Section 4.3. We then describe the different algorithm phases and the role of both compiler and the OS in Section 4.4 and Section 4.5. We present a detailed evaluation on different real-time applications in Section 4.6 followed by a chapter conclusion in Section 4.7.

## 4.1 COMPILER AND OS COLLABORATIVE APPROACH

To control the number of speed changes during the execution of an application, we present a collaborative compiler-OS technique [68]. Initially, some timing information is collected about a program's execution behavior. This information is used offline to compute how frequently PMPs are invoked. The number of cycles between executions of two PMPs is called the *PMP-interval*. The length of the PMP-interval controls the number of PMPs executed in an application, and therefore the overhead. Next, the compiler inserts instrumentation code to compute some timing information and make it available to the operating systems. We call such instrumentation: power management hints, *PMH*. PMHs compute the *worst-case remaining* cycles, $wcr_i$, starting from the $i^{th}$ PMH location to the end of the application. The value of $wcr_i$ at these locations may vary dynamically based on the executed path for each run. For example, the remaining cycles at a PMH inside a procedure body is dependent on the path from which this procedure is invoked. During run-time, a PMH computes and passes dynamic timing information (i.e., $wcr_i$) to the OS in a predetermined memory location named $R_{WCR}$, which holds the most recent value of the estimated worst case remaining cycles. Periodically, a timer interrupt invokes the operating system to execute an *interrupt service routine* (ISR) that does the PMP job; that is, the ISR adjusts the processor speed based on the latest worst case remaining cycles, $WCR$, value and the remaining time to the deadline.

Figure 4.1 shows an example CFG and how PMHs and PMPs work together. A PMP is periodically invoked (the large bars) to adjust the processor's speed based on $R_{WCR}$. In the compiler, a PMH placement algorithm first divides an application code into *segments*, such that the worst case execution cycles of each segment, $wcc$, does not exceed the PMP-interval length (in cycles). The compiler then inserts one PMH (the short bars) after each segment. Each $PMH_i$ computes the worst case remaining cycles ($wcr_i$) and stores it in $R_{WCR}$. A $PMH_i$ is guaranteed to execute at some point before a PMP to update the $R_{WCR}$ with the value of $wcr_i$ based on the path of execution. Because the actual execution cycles of each segment is less than or equal $wcc$, more than one PMH can be executed in a single PMP-interval, improving the estimation of $WCR$.

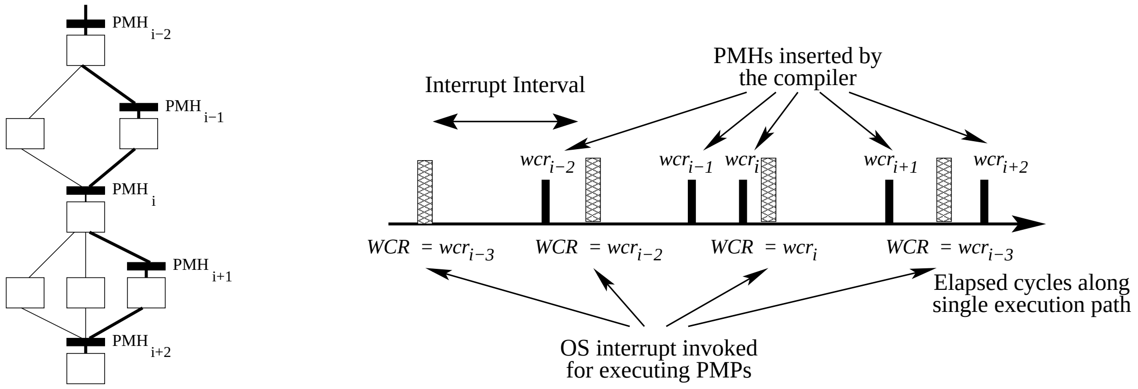


Figure 4.1: (a) Sample CFG. (b) Invocations of PMHs & PMPs for executing the bold path in (a).

The collaborative scheme uses knowledge shared by both the OS and the compiler to control the speed, in contrast to OS-only or compiler-only DVS scheme. The collaborative scheme exploits the compiler's ability to extract run-time information about the application execution progress indicated by $wcr_i$. The scheme also takes advantage of the OS's ability to schedule DVS events independent of which path is executed in the application. In a solely OS-directed scheme, the OS is unaware of an application's execution progress and the number of remaining cycles, while a solely compiler-directed scheme can not control how often PMPs will be called (due to the non-determinism of the path followed during program execution), and thus, there is no upper bound on the overhead.

## 4.2 PMP VERSUS PMH PLACEMENT STRATEGY

For a single application, a PMH inserted in the application code while PMPs are invoked by the OS is more beneficial with respect to slack utilization than inserting PMPs directly in the code. Since actual execution paths are only known at run-time, placement of PMPs directly in the applications code based on information available offline may lead to inefficiencies. There are two reasons for the inefficiency. First, an offline decision may not account for all the run-time slack generated by an application, and second, there is no control on the frequency of executing PMPs due to the uncertainty related to which paths will be taken at run-time.

Scheduling the speed offline based on profile information can not use all the generated slack. The actual execution time of an application is likely to differ from its offline estimates using profiling or static analysis. This difference contributes to the dynamic slack generated in the system. A compiler-only intra-task DVS scheme makes speed change decisions based solely on offline estimates. For this, such schemes can not exploit all the dynamic slack generated at run-time. On the other hand, PMHs can convey run-time knowledge about the application execution. Thus, during run-time a more informed decision can be made based on better estimates of the actual slack available at the time of the speed transitions. Further analysis of the slack utilization efficiency compared to other offline schemes is discussed in Section 4.6.

The uncertainty in finding the execution path at compile time can cause some offline placement strategies to increase energy consumption rather than achieve energy savings for some combinations of code structures. This can be due to the overhead associated with executing too many PMPs or due to inefficient slack utilization due to too few PMP invocations. Figure 4.2 and Table 4.1 show different PMP placement strategies and some combinations of code structures that show the shortcomings of those strategies for specific cases. Although some slack is exploited, the given examples show that there is no guarantee that the overhead of changing the speed does not offset the total energy savings. We evaluate this issue later in this chapter.

The strength of the collaborative scheme lies in three properties. First, a separate PMH placement algorithm can be devised to supply the OS with the necessary timing information about an application at a rate proportional to the PMP invocation. Second, the actual speed-change is done seldom enough by the OS at pre-computed time intervals (every PMP-interval) to keep the overhead low. The PMP-interval is tailored to an application's execution behavior in a way that minimizes the energy consumption while meeting deadlines. Finally, by giving the OS control to

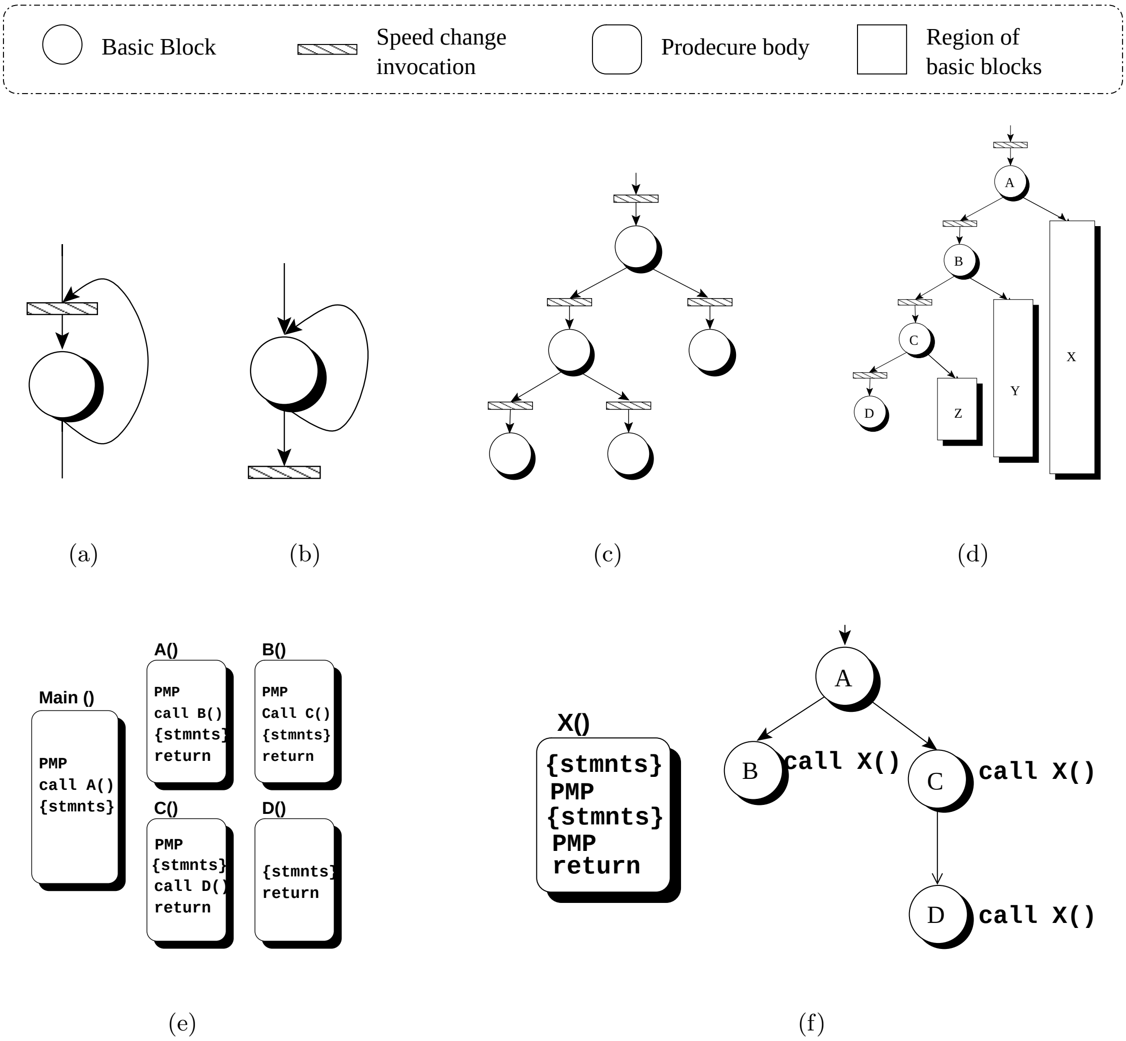


Figure 4.2: Some PMP placement strategies and their adverse cases of code structure (See Table 4.1 for description).

Table 4.1: Adverse combinations of code structure for some PMP placement strategies.

| **Case** | **Placement Strategy** | **Adverse case** | **Side effect** |
|---|---|---|---|
| **(a)** | PMP placed at each loop iteration | Loop body (single iteration) consists of one or few small sized basic blocks, i.e., loop body size is in order of few to hundreds of cycles) | Too frequent speed-change invocations relative to the slack that may be generated in the system. |
| **(b)** | A PMP placed before the start or after the termination of a loop execution | An application consisting of single main loop | Dynamic slack generated from the loop can not be exploited. |
| **(c)** | PMP placed at each branch | Each branch is a basic block before it branches again | Too frequent speed-change invocations. |
| **(d)** | Place PMP in branches with non worst case (cycles) path | Path ABCD is a sequence of basic blocks interleaved with PMPs. Regions X,Y and Z are worst case paths after blocks A,B, and C, respectively. | Too frequent speed-change invocations. |
| **(e)** | PMP placed at each procedure call | Frequent call to procedures of small sizes | Too frequent speed-change invocations. |
| **(f)** | PMPs inserted in procedures' bodies | Procedure X contains PMPs and is called from different call-sites in the application | Need a way to estimate the worst case remaining cycles inside procedure X to compute a new speed |

change the CPU speed, the scheme controls the number of executed PMPs independent of any execution path.

The advantage of using a collaborative scheme that includes both PMHs and PMPs over a scheme that uses only PMPs is that the application's execution progress can be known without actually invoking a speed-change and incurring its high overhead. For example, consider the case of a loop body that contains mainly two exclusive branches (e.g., `if-then-else` statement), such that the *wcc* of the first branch (`then` part) is much larger then the second one (`else` part). Consider a scheme that utilizes PMPs alone (the locations of PMPs in the code are statically determined), and places a PMP inside the loop body before the branches. This scheme would hurt energy consumption if the smaller branch is executed more often than the large branch. This is because, with each loop iteration, the scheme would pay the overhead of the speed-change independent of the branch/path visited. However, using the proposed scheme, the overhead in each iteration is reduced by replacing the PMP in the loop with a PMH. Also, the actual speed-change (PMP) is scheduled after a "reasonable" amount of work, such that the potential energy savings from DVS can justify the energy overhead of the PMP. Similar scenarios occur in all program constructs that have variable dynamic execution times.

## 4.3 COLLABORATIVE POWER MANAGEMENT ALGORITHM OVERVIEW

The collaborative approach is divided into offline and run-time phases. Figure 4.3 shows the tasks required at each phase. Below, we describe the role of the compiler and of the OS in each of these tasks.

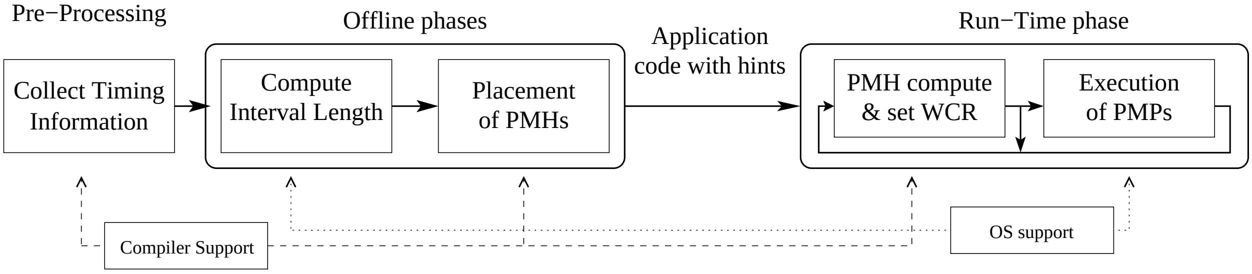


Figure 4.3: Phases of the collaborative power management scheme.

The compiler and the OS interplay roles in our scheme. The compiler analyzes the application's code through a timing profile phase. Then, the OS uses this profiling data to compute the interrupt interval length. Based on the interval length and the timing information collected in pre-processing, the compiler places the PMHs in the application source code. At runtime, as the application executes, hints are executed and evaluated based on the code path followed. Hints estimate the worst case remaining cycles, and accordingly the OS computes the desired speed. Next, we describe the detailed roles of both the compiler and the OS.

## 4.4 COMPILER SUPPORT FOR THE COLLABORATIVE SCHEME

### 4.4.1 Timing Extraction

Before deciding on the locations of the PMHs in the code, the compiler collects timing knowledge about an application. We assume that the WCET (the worst-case execution time at maximum speed) of an application is known or can be obtained (for example, through profiling or software timing analysis[1] [69, 70]).

To collect timing information, we divide a program's control flow graph (CFG) into *regions*. Each region contains one or more adjacent basic blocks ($BB$). Since the typical size of a basic block is small (tens of cycles) compared to the typical size of a PMP-interval (hundreds of thousands of cycles), blocks can be aggregated into a region to be profiled. Region formation has the key advantage of reducing the number of traversed objects and the complexity of traversing the CFG in later stages of our power management scheme. Region formation also avoids aggregating too many basic blocks into a region such that the region execution at run-time exceeds the PMP-interval. Since some procedures called within a BB may execute for relatively long periods, we avoid overgrowing a region by isolating each BB that contains a procedure call into its own region. We aggregate the blocks into a region while maintaining the structure of the program constructs (for example, loops and control flow statements). Regions are determined by a region construction algorithm described in [71].

Figure 4.4 shows examples of region formation and how it reduces the number of regions and preserves the program structure. We show the cases where branches and loops contain procedure calls. For Figure 4.4-a (a branch construct like an `if-then-else` statement), if none of the blocks

[1] The usual trade-off applies: profiling cannot guarantee the deadlines of tasks, since it does not derive the **worst-case** execution, while software analysis allows for that at the expense of lower slack utilization.

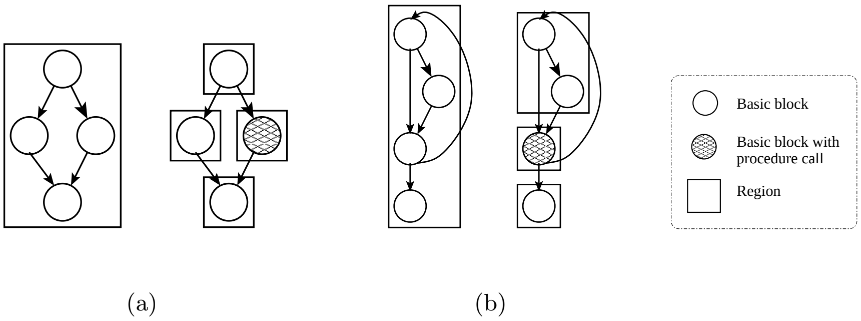


Figure 4.4: Examples on region segmentation for (a) branches and (b) loops, with and without procedure calls.

forming the construct contains any procedure call then all the blocks are merged into one region. Also, if followed by a block without calls, then that block is merged with the larger region. However, if any of the BBs that belong to the branch construct contains a call, then this block forms a separate region not to be merged with other blocks. With respect to loops (Figure 4.4-b), if the entire loop body does not include any calls then it forms a single region that can be merged with other BB outside the loop. Similar to the case of branches with procedure calls, if any of the BBs has a call then the body is divided into more than a single region, and these regions cannot be merged with any region outside the loop.

After forming regions, profiling is used to extract timing information about the constructed regions. For loops, we collect the maximum cycle count of loop segments along with the maximum trip count of that loop, where *loop_segment* includes the total execution cycles of the loop iterations and, the *trip count* is the number of iterations in the loop. We use loop_segment and the maximum trip count to estimate the *loop body* size, which is the number of cycles spent in a single iteration of the loop. For each procedure, we collect the time spent executing that procedure. High level information (WCC and average cycle count) about the entire application is used to compute the PMP-interval size.

Based on the timing information collected by the compiler, the OS computes the best PMP-interval that avoids excessive execution of PMPs (as described in Section 4.5). PMP-interval length is dependent on the speed scheduling scheme used by the OS. PMP-interval length is key information that the compiler use for the PMH placement as described in the next section.

### 4.4.2 Placement of PMHs

The goal of the PMH-placement algorithm is to ensure that, at least, a PMH is executed before the invocation of the next PMP. The placement algorithm traverses the CFG to insert PMHs no further apart than the size of the PMP-interval (computed by the OS). Ideally, a PMH should execute right before the PMP is invoked, so that the PMP has the most accurate information. Since we do not control the application's dynamic behavior, we allow more than a single PMH to be executed in each execution_segment to improve the probability of an accurate speed computation.

In the PMH-placement algorithm [72], while traversing the CFG of a procedure, a cycle accumulators, *ac*, is incremented by the value of the elapsed worst-case cycles of each traversed region. A PMH is inserted in the code just before this counter exceeds the PMP-interval and the counter is reset. PMH locations are selected according to the different types of code structures in an application, namely sequential code, branches, loops or procedure calls. Next we describe the criteria of placement in each of these cases.

**4.4.2.1 Sequential code** We define a *sequential segment* as a series of adjacent regions in the CFG that are not separated by branches, loops, joins or back edges (although the segment may include a procedure call). Sequential placement inserts a PMH just before *ac* exceeds the PMP-interval. It is non trivial to insert a PMH in a region containing a procedure call, since the procedure's cycles are accounted for in the enclosing region's execution cycles. If the called procedure is too large (i.e., larger than the PMP-interval), then inserting PMHs only at the region boundary is insufficient to update the $WCR$ before the next PMP invocation. For this reason, we need to further investigate possible locations inside a region related to the locations of procedure calls. For regions in a sequential segment, PMHs are inserted according to the following rules:

- When the cumulative counter *ac* exceeds the PMP-interval cycles and there are no procedure calls in the region then a PMH is placed before the current region.
- If a region contains a procedure call and the body of a called procedure exceeds the PMP-interval, a PMH is placed before the procedure call and another PMH after the procedure call. The called procedure is marked for later placement. The PMH before the call indicates the worst case remaining cycles at the start of this procedure's execution. The PMHs before and after the called procedure – that may contain PMHs– simplify the PMH placement scheme by avoiding the need to track inter-procedural information about *ac* (i.e., *ac* does not have to be

remembered from one procedure to the next).

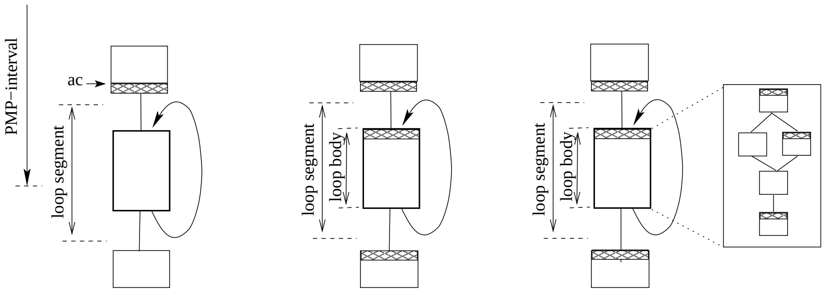
**4.4.2.2 Branches** For any branch structure, each path leaving a branch is treated as a sequential segment or recursively as a branch structure. At any branch, the value of $ac$ is propagated to all the branch's paths. In a join, $ac$ is set as the maximum value of all the propagated counters just before the join.

**4.4.2.3 Loops** The decision of placing PMHs in a loop is based on the loop_segment and loop body sizes (in cycles). The different cases for inserting PMHs in loops are as follows:

- If the sum of $ac$ and loop_segment exceeds PMP-interval but the loop_segment is smaller than PMP-interval then a PMH is placed before the loop (see Figure 4.5-a).
- If a loop_segment exceeds PMP-interval but the loop body is smaller than PMP-interval, then a PMH is placed at the beginning of the loop body in addition to the PMH placed before the loop. Another PMH is inserted after the loop exit (see Figure 4.5-b).
- If the size of the loop body exceeds PMP-interval, a PMH is placed at the start of the loop body and the loop is treated separately as either sequential code or code with branches. Another PMH is inserted after the loop exit (see Figure 4.5-c).

The reason for placing a PMH after the loop in the last two cases is to adjust any over-estimation of $WCR$ done by the loop's PMHs. Nevertheless, over-estimation of $WCR$ is possible in the case where the actual number of loop iterations is unknown during loop execution.

**4.4.2.4 Procedure calls** As described above, in the processing of sequential segments, procedure calls are detected and accordingly some procedures are selected to undergo PMH placement in their bodies. The procedure selection is subject to satisfying the need for updating the $WCR$ (through PMHs) at least once during each PMP_interval. For each procedure selected for placement, a PMH is placed before a procedure call to compute $wcr_i$ of each instance of this procedure, and to store $wcr_i$ in the procedure's activation frame. Instrumentation code is inserted in the procedure prologue to retrieve the value of $wcr_i$ computed dynamically by the PMH located before the call. Each PMH located inside this procedure uses this value in its calculations of the remaining cycles. This instrumentation is necessary because a procedure may be called from different program paths and each with different $wcr_i$ value even for the same called procedure (but different



(a) (b) (c)

Figure 4.5: PMH placement in loops when (a) ac+loop_segment > PMP-interval, (b) loop_body < PMP-interval < loop_segment, and (c) loop_body > PMP-interval.

instance). This is especially beneficial in case of recursive calls, where each call instance has its own $wcr_i$ estimate.

### 4.4.3 PMH computation of $wcr_i$

The PMH placement algorithm can insert two different types of PMHs, *static* or *index-controlled*, based on the program structure. The two types compute the worst-case remaining cycles based on whether the PMH is located inside a loop or not.

**4.4.3.1 Static PMH** This type of PMH is placed in any location in the code outside a loop body. Generally, a PMH is located inside a procedure. A PMH computes $wcr_i$ based on the remaining cycles at the start of the procedure instance, $p\_wcr$. Since the path is only known at run-time, during execution, a procedure retrieves its $p\_wcr$ stored in its stack space. A static PMH computes $wcr_i$ by subtracting the displacement of the PMH location in the procedure from $p\_wcr$. For example, for a PMH inserted 100 cycles from the beginning of the procedure, the remaining number of cycles in the program is computed as: $wcr_i = p\_wcr - 100$.

**4.4.3.2 Index-controlled PMH** PMHs of this type are inserted inside the body of a loop where $wcr_i$ varies according to the current loop iteration counter. The PMH computes the worst case remaining cycles based on equations from [70]. Using the worst case cycles of a loop segment, *wc_loop_segment*, the term *wc_loop_segment* / *maximum_trip_count* (maximum number of iterations) estimates the size of a loop body, $loop\_body$. Then, $wcr_i$ is computed as $wcr\_before\_loop - (iteration\_count * loop\_body) - ldisp$, where $wcr\_before\_loop$ is the remaining cycles determined at run-time from the PMH that precedes the loop, and $ldisp$, is number of cycles elapsed since the start of the loop body in a single iteration. The same technique can also be applied in case of nested loops.

Estimating the loop's execution time using *wc_loop_segment* rather than using *wc_loop_body* * *maximum_trip_count* (this term denoted by *wc_lbody*) creates a trade-off between energy savings and timeliness. Although accounting for the *wc_lbody* overestimates the loop execution time and thus increases the probability that a task will not miss its deadline[2]. This overestimation delays the slack availability until the loop finishes execution. On the other hand, using *wc_loop_segment* lets the slack to be used earlier in the execution, resulting in more energy savings. Even if the actual cycles of a loop iteration exceeds *loop_body*, the application is less likely to miss a deadline in some cases: there is no large variation in the actual cycles of each iteration, or the iteration with the duration of *wc_lbody* happens early in the execution that the remaining code would generate enough slack to avoid a deadline miss. Our experiments confirm that these cases are very common, and thus no deadlines were missed. We conclude that, if the application is very sensitive to deadline misses, then index-controlled PMH computes the $wcr_i$ as a function of the *wc_lbody*; otherwise, PMH computes it based on *wc_loop_segment*.

### 4.4.4 Example on the PMH placement and execution

We present an example that shows how the placement algorithm works for the simple CFG shown in Figure 4.6. The timing information for the CFG is listed in Tables 4.2 and 4.3. Assume that the PMP-interval is 1000 cycles. Below, we first give the details on how the algorithm selects the locations to insert PMHs and then the details of how each inserted PMH computes $WCR$.

- **PMH$_1$:** A PMH is placed at the beginning of the CFG, indicating the $WCR$ (= 16,500 cycles) at the start of this procedure.
- **PMH$_2$ and PMH$_3$:** Since R2 starts a loop with a segment size of 16,000 cycles that is larger than PMP-interval, $PMH_2$ is placed at the end of R1 (before the loop). Because the body of

[2] Software analysis (in Section 4.4.1) does guarantee deadlines

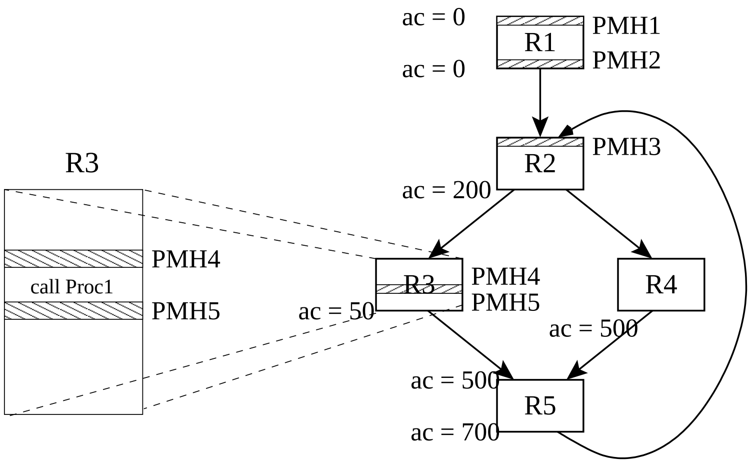


Figure 4.6: Example of the PMH placement in a simple CFG.

Table 4.2: Profiled *wcc* for each region in CFG shown in Figure 4.6

| Region | WCC (cycle) |
|---|---|
| 1 | 500 |
| 2 | 200 |
| 3 | 1200 |
| 4 | 300 |
| 5 | 200 |

Table 4.3: Other profiled information for CFG shown in Figure 4.6

| | |
|---|---|
| Loop body | 1600 cycles |
| Max TripCounts | 10 iterations |
| Procedure size | 1100 cycles |
| disp of Proc1 | 50 cycles |

Table 4.4: The inserted PMHs details.

| PMH | Type | Computed as: |
|---|---|---|
| 1 | static | p_wcr - 0 (= 16,500 for this procedure instance) |
| 2 | static | p_wcr - 500 (= 16,000 ) |
| 3 | index-controlled | 16,000 - (iteration count x 1600) |
| 4 | index-controlled | 16,000 - (iteration count x 1600) - 250 |
| 5 | index-controlled | 16,000 - (iteration count x 1600) - 1350 |

the loop exceeds PMP-interval, $PMH_3$ is placed at the start of R2 (inside the loop) and, the loop body is traversed for further PMH placement (similar to case (c) in Figure 4.5).

- **PMH$_4$ and PMH$_5$:** Because the sum of *ac* and R3 cycles is larger than PMP-interval, the algorithm looks for the first called procedure (i.e., `Proc1`) in region R3 which is located at 50 cycles from the beginning of R3. Since the procedure body is larger than PMP-interval, $PMH_4$ and $PMH_5$ are placed before and after the procedure call, respectively. Procedure `Proc1` is marked for later PMH placement. The PMHs placed inside procedure `Proc1` are not shown in this example.

Assuming that the presented CFG is the application's `main` procedure, *p_wcr* equals the application's worst case remaining cycles (= 16,500). Table 4.4 lists the details of the computation done by each inserted PMH. Note that $PMH_2$ is useful in evaluating $wcr\_before\_loop$.

## 4.5 OS SUPPORT FOR THE COLLABORATIVE SCHEME

To support our collaborative scheme on the OS side, the OS uses information provided by the compiler and periodically schedule the speed change. We first explain our speed scheduling schemes in Section 4.5.1, then show how we compute the best PMP-interval length used with each scheme in Section 4.5.2. Implementation details to support our scheme in the OS are described in Section 4.5.3.

### 4.5.1 Dynamic Speed Setting

From the CPU perspective, we can achieve the least CPU dynamic energy consumption by lowering the operating frequency. However, from the system perspective, lowering frequency increases execution time and increases the power consumption of other system components, memory in particular. This increase in memory energy can overshadow CPU energy savings. Hence, two constraints have to be considered in setting the CPU frequency. First, the algorithm should lower the CPU frequency such that it is just high enough to meet the application deadline. Second, the algorithm should prevent the CPU frequency from being too low, to the point of increasing the overall system energy. We present ways of computing the CPU speed for efficient energy savings in Section 4.5.1.1, then show how to make the CPU speed setting aware of the memory energy consumption in Section 4.5.1.2.

**4.5.1.1 CPU speed computation** We use two schemes from [13] as examples to demonstrate ways of computing CPU speed in the OS-ISR. We selected the Proportional and Greedy dynamic schemes. We further incorporated the overhead in the speed computation in these schemes. To guarantee meeting the deadline, the total time overhead of computing and changing the speed is deducted from the remaining time before deadline. This time overhead is composed of changing the speed once in the current ISR interval $(T_{comp}(f_{curr}) + T_{set})$ and once for potentially changing the speed after the next interval to the maximum speed $(T_{comp}(f_{next}) + T_{set})$ if the schedule requires so. The total time overhead is expressed as $T_{total}(f_{curr}, f_{next}) = T_{comp}(f_{curr}) + T_{comp}(f_{next}) + 2T_{set}$.

**The Proportional scheme**

In this scheme, reclaimed slack is uniformly distributed to all remaining intervals proportional to their worst-case execution times. Our main concern here is to compute the exact time left for the application to execute before the deadline. The processor's speed at the start of an interval, $f_{next}$, is computed as follows: the execution times for the remaining tasks are stretched out based on the remaining time to the deadline $(d - ct)$ minus the possible time spend switching the speed, $T_{total}$, where *ct* is current time at the beginning of the interval. While taking into consideration the overhead of changing the speed, this scheme computes a new speed as:

$$f_{next} = \frac{R_{WCR}}{d - ct - T_{total}(f_{curr}, f_{next})} \tag{4.1}$$

**The Greedy scheme**

In this scheme all the available slack is allocated for the next interval. As a result, the scheme tends to schedule the first few intervals at a low speed, and gradually increases it after consuming slack. The frequency for segment $i$ is computed as:

$$f_{next} = \frac{wcc}{d - ct - \frac{R_{WCR} - wcc}{f_{max}} - T_{total}(f_{curr}, f_{next})} \tag{4.2}$$

where $f_{max}$ is the maximum frequency the processor can operate on and $wcc$ equals $WCC/$number of intervals. Similar to the proportional scheme, the overhead components should be subtracted from the remaining time after accounting for the execution time of the remaining intervals (excluding the current one) running at $f_{max}$.

**4.5.1.2 Memory-aware speed setting** To avoid operating at an excessively low frequency, which increases the overall energy consumption, we have to account for the power of other components while computing the speed. We only focus on memory in this work as the other major power consumer in the system; however, we can easily extend the model to other components. From the memory perspective, increasing execution time increases the idle time during which data needs to be refreshed in memory. hence, reducing the CPU speed increases the memory energy consumption. To reach a compromise between reducing the CPU energy and the memory energy, we limit the CPU frequency at a point where further reduction in operating frequency will force memory to consume energy more than the CPU energy savings. We call this frequency, the break-even frequency, $f_{be}$. Figure 4.7 illustrates the intuition behind the computation of $f_{be}$.

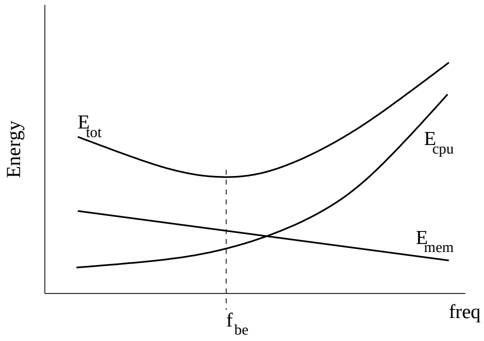


Figure 4.7: Lowest total energy consumption at break-even frequency.

Given that supply voltage, $V_{dd}$ is proportional to frequency, we simplify the CPU energy model to $E = K_c f^{\beta}$ where $K_c$ is a constant that includes the factor of proportionality as well as some technology specific parameters. $\beta$ typically equals 3. For memory, we consider the case where the memory controller can turn off memory banks that belong to an application immediately after the application finishes execution. We define $P^m_{static}$ and $P^m_{dyn}$ as the average power consumed in memory during idle period and data access, respectively. Let $E_{tot}$ be the sum of the CPU and memory energy consumption.

$$E_{tot} = WCET \; \frac{f_{max}}{f_{next}} \; K_c f^{\beta}_{next} \; + \; P^m_{static} \; WCET \; \frac{f_{max}}{f_{next}} \; + \; P^m_{dyn} \; N_m$$

Where $N_m$ is the number of memory accesses. To get the frequency that yields minimum energy consumption, we differentiate $E_{tot}$ with respect to frequency and equate to zero. Hence,

$$\frac{dE_{tot}}{df} = WCET \; f_{max} \; (\beta - 1) \; K_c \; f^{\beta-2}_{next} \; - \; P^m_{static} \; WCET \; \frac{f_{max}}{f^2_{next}}$$

After equating with zero, we find that break-even frequency ($f_{be} = f_{next}$) that results in the minimum energy is $f_{be} = \sqrt[\beta]{\frac{P^m_{static}}{(\beta-1)K_c}}$. Hence, the OS selects a frequency that is the minimum of $f_{next}$ and $f_{be}$.

The computation of $f_{be}$ can be applied to account for other components in the system, such as disk and motherboard power. It also can account for the aggregate power of multiple components by replacing $P^m_{static}$ by the sum of the static power of all components under consideration. The same concept can also be applied to account for leakage power assuming that leakage power is constant throughout execution.

### 4.5.2 Setting the PMP-interval

Determining the PMP-interval for invoking PMPs is based on the average and worst case execution times of a program and the overhead of PMP executions. To compute PMP-interval length, we derive a closed form formula from Equation 4.1 or 4.2 to model the relationship between the computed frequency and the number of execution segments in a program, $n$. Then, we find the value of $n$ that achieves minimum energy consumption when executing a program. From $n$, the optimal number of PMPs, we compute the best PMP-interval length as shown below.

We incorporate the overhead of a invoking PMP in the speed computation. This overhead includes the time and energy consumed in computing and setting the speed. We assume that (1) each execution_segment has a perfect execution behavior (that is, the actual execution times of all the execution_segments are equal to the average execution time of each execution_segment), and (2) the overhead, $T_{total}$, of computing and setting the speed at each PMP is constant. Below are the closed form equations for Proportional and Greedy speed setting schemes. Appendices A.1 and A.2 list the derivations of these formulas starting from Equations 4.1 and 4.2, respectively.

$$\textbf{Proportional scheme}: \ \phi_i = \frac{1}{n-i+1}(\frac{n}{load} - \frac{T_{total}}{wcc})\Pi_{k=1}^{i-1}[1 - \frac{\alpha}{n-k+1}] \tag{4.3}$$

$$\textbf{Greedy scheme}: \ \phi_i = \frac{1-(1-\alpha)^i}{\alpha} + (\frac{n}{load} - n - \frac{T_{total}}{wcc})(1-\alpha)^{i-1} \tag{4.4}$$

where $\phi_i$ is the ratio $f_{max}/f_i$ ($f_i$ is the operating frequency for execution_segment $i$), $n$ is the number of execution_segments in the longest execution path, $load$ equals $WCET/d$, and $\alpha$ is the ratio of average to worst case cycles in each execution_segment. To obtain $n$ that corresponds to the minimum energy consumption in the average case, we substitute the formula of $\phi_i$ in the energy formula from Section 3.2 and solve iteratively for $n$. Hence, we get the PMP-interval size by dividing the average number of cycles for executing an application by $n$.

### 4.5.3 OS extensions

We introduce an ISR for setting the speed [73] and a system call to set the application's power management parameters in the OS. Additionally, more information is stored in the process's context to support multi-process preemption. Details about each of the needed OS extensions are given below.

- **Power-management parameters initialization system call:** This system call, inserted by the compiler, sets the application's power management parameters in the OS. The parameters include the length of PMP-interval (in cycles), and the memory location of the WCR. This system call is called once at the start of each application. Although the PMP-interval is constant (in cycles) throughout the application execution, each ISR instance computes its equivalent time (to set the interrupt timer) based on the frequency set by this PMP instance. If there are multiple applications running, the context switching mechanism adjusts these timers (see below).
- **Interrupt service routine:** According to the dynamic speed setting scheme for computing a new operating frequency, the ISR selects an appropriate speed/power level to operate on. When working with discrete power levels, for a real-time application to meet its deadline, there are two possible ways to set the operating speed. One is to operate on the smallest power level larger than or equal to the desired frequency obtained from the speed setting scheme. The

```
$ct$ = read current time
$f_{curr}$ = read current frequency
$f_{new}$ = compute_speed (algo, $ct, d, wcc, WCR$ )
if ($f_{new} \neq f_{curr}$) then
    if ($f_{new} \leq f_{min}$ ) then $f_{new} = f_{min}$
    if ($f_{new} \geq f_{max}$ ) then $f_{new} = f_{max}$
    if ($f_{new} < f_{be}$) then $f_{new} = f_{be}$
    set_speed ( discretize_speed ($f_{new}$) )
    next_PMP = ct + (PMP-interval / $f_{new}$)
    set_timer (next_PMP)
rti // return from interrupt
```

Figure 4.8: ISR pseudocode for PMPs.

second is to use the dual-speed-per-task scheme [74]. The second scheme may be more efficient, depending on the overhead of changing the speed twice. However, in this work we choose to select the closest speed larger than the desired speed for simplicity. The pseudocode of the ISR is listed in Figure 4.8. compute_speed is responsible for computing the next speed according to a selected speed scheduling scheme, algo. discretize_speed returns the smallest available frequency that is bigger than the input frequency, $f_{new}$.

- **Preemption support:** In case there is more than one application running in a preemptive system, the OS should keep track of how long each application was running with respect to the current PMP-interval. At each context switch, the elapsed time in the current PMP-interval is stored as part of the state of the departing process (application) and replaced by the corresponding value of the newly dispatched process. The elapsed time in the current PMP-interval and the operating frequency become part of the application context.

  In case of multiple applications, every time the application is preempted, the timer value is stopped and the time value is reset when the application is resumed. For this, the PCB must contain a variable, *saved_PMP*, that saves these values: at preemption time, $saved\_PMP = next\_PMP - ct$ and at resume time, $next\_PMP = ct + saved\_PMP$, where $ct$ is the current time.

Table 4.5: Simplescalar configuration

| | |
|---|---|
| fetch width | 4 instruction/cycle |
| decode width | 4 instruction/cycle |
| issue width | 4 out of order |
| commit width | 4 instruction/cycle |
| RUU size | 64 instruction |
| LSQ size | 16 instruction |
| FUs | 4 int, 1 int mult/divide, 4 fp, 1 fp mult/divide |
| branch pred. | bimododal, 2048 table size |
| L1 D-cache | 512 sets, 32 byte block, 4 byte/block, 1 cycle, 4-way |
| L1 I-cache | 512 sets, 32 byte block, 4 byte/block, 1 cycle, 4-way |
| L2 cache | 1024 sets, 64 byte block, 4 byte/block, 1 cycle, 4-way |
| memory | 40 cycle hit, 8 bytes bus width |
| TLBs | instruction:16 sets, 4096 byte page - data: 32 sets, 4096byte page |

When the system allows for multitasking, our scheme will extract timing information (i.e., slack) from the task execution through the PMHs, and the operating system will decide how to use such slack in inter-task DVS [13, 16].

## 4.6 EVALUATION

We evaluate the efficacy of our scheme in reducing the energy consumption of DVS processors using the SimpleScalar micro-architecture toolkit [75] with configurations shown in Table 4.5. Also, a dual-issue processor with similar configurations was tested and had the same energy savings trends. Because these results are similar, we present only the results for the 4-way issue machine.

We extended the *sim-outorder* simulator with a module to compute the CPU and memory energy consumption of running an application as $E_{tot} = E_{cpu} + E_{mem}$. We mainly focus on optimizing the CPU dynamic energy given that there are efficient architectural solution that efficiently reduce leakage energy [76] [77] [78]. CPU energy, $E_{cpu}$, is a function of the number of cycles, $C$, spent at

Table 4.6: The power levels in the Transmeta processors model.

| Frequency(MHz) | 700 | 666 | 633 | 600 | 566 | 533 | 500 | 466 |
|---|---|---|---|---|---|---|---|---|
| Voltage (V) | 1.65 | 1.65 | 1.60 | 1.60 | 1.55 | 1.55 | 1.50 | 1.50 |
| Frequency(MHz) | 433 | 400 | 366 | 333 | 300 | 266 | 233 | 200 |
| Voltage (V) | 1.45 | 1.40 | 1.35 | 1.30 | 1.25 | 1.20 | 1.15 | 1.10 |

Table 4.7: The power levels in the Intel XScale processor model.

| Frequency(MHz) | 1000 | 800 | 600 | 400 | 150 |
|---|---|---|---|---|---|
| Voltage (V) | 1.8 | 1.6 | 1.3 | 1.0 | 0.75 |

each voltage level, $V$ ($E_{cpu} = Pt = CV^2$, since $f\alpha\frac{1}{t}$ and $E_{cpu} = kCV^2$, where $k$ is a constant–in this chapter, we consider only $k = 1$). $C$ includes the cycles for executing the application as well as the cycles for computing the speed in PMPs and $wcr_i$ in PMHs. Since energy consumed during memory data access is not a function of the CPU operating frequency, we focus only on the memory idle power. Hence, the memory energy, $E_{mem}$, is the integration of the idle power dissipated in memory over time. When memory is idle (not servicing any requests), we assume an average memory idle power equals 20% of the maximum CPU power, and 0 W while shut-down [79].

We also consider the overhead of setting the speed by adding a constant energy overhead for each speed-change to the total energy consumption. In the experiments below, we used the Transmeta Crusoe and the Intel XScale CPU cores. The Transmeta Crusoe has 16 speed levels and the Intel XScale has five levels. Tables 4.6 and 4.7 show the different speeds and the corresponding voltages for both the Transmeta and Intel models. Energy consumed in other subsystems is beyond the scope of this evaluation.

We emulate the effect and overhead of the ISR in SimpleScalar by flushing the pipeline at the entry and exit of each ISR. The PMP (i.e., the ISR) computes a new frequency every PMP-interval and then sets the corresponding voltage. The no-power-management (NPM) scheme always executes at the maximum speed, and static power management (SPM) slows down the CPU based on static slack [80]. SPM scales down the frequency according to the ratio of deadline to WCET. It is proved to be a static optimal [80]. In addition, we evaluate our Greedy and Proportional power management described in Section 4.5.1.

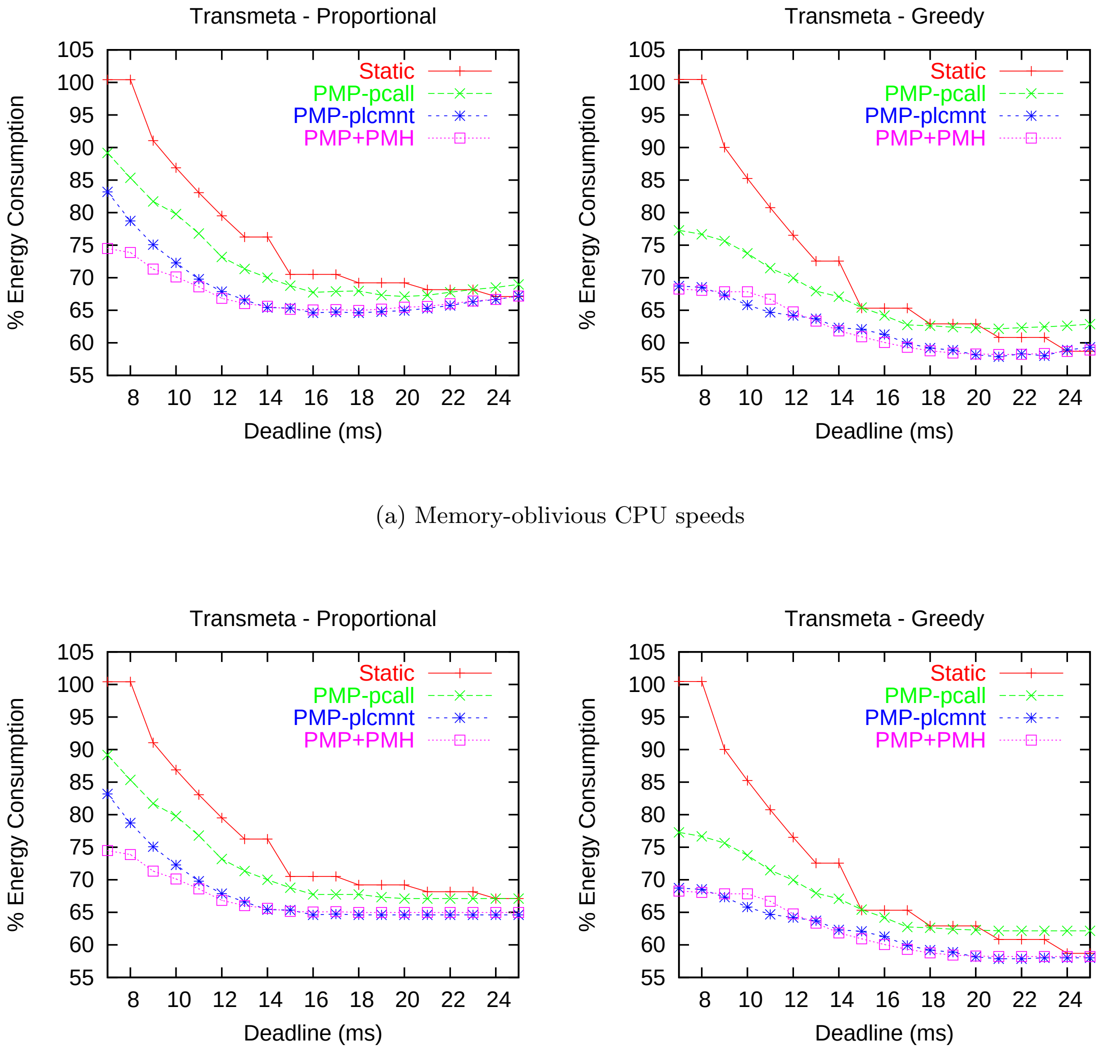
We show the no power management (NPM) scheme that always executes at the maximum speed and the static power management (SPM) scheme that slows down the CPU based on static slack [80]. To demonstrate the potential benefit of using PMHs, we compare the collaborative technique (PMP+PMH) with two schemes that use only PMPs: the first (*PMP-pcall*) places a PMP before each procedure call, and the second (*PMP-plcmnt*) inserts PMPs according to the PMH-placement algorithm.

In our benchmarks, the most frequently called procedures have simple code structures and are less likely to generate slack due to their small sizes. To be fair to the PMP-call scheme (i.e., to reduce its overhead), we do not place PMPs before small procedures that do not have any call to other procedures. We show experimental results for three time-sensitive real-life applications: automatic target recognition (ATR), an MPEG2 decoder, and a sub-band tuner. In the simulation, all the application deadlines were met for the tested data sets.

**4.6.0.1 Impact on energy and performance** Our results show that, as the time allotted (or the deadline) for an application to execute is increased, less dynamic energy is consumed due to the introduction of more slack that can be used to reduce the speed/voltage. However, as we extend the deadline, more memory idle energy is consumed. Hence, the total energy is reduced due to savings in dynamic energy, but it can be increased due to the increase in the memory energy. This is especially obvious when extending the deadline does not result in enough slack to lower the speed. When an application runs at the same frequency as with a tighter deadline, it consumes the same CPU dynamic energy, but higher memory idle energy. This is clear with the ATR application in the Intel case when the deadline exceeds 12 ms; the application consumes constant dynamic energy, but the memory energy consumption increases as the deadline increases.

**Automatic target recognition.** The ATR application[3] does pattern matching of targets in input image frames. We experimented with 190 frames, the number of target detections in each frame varies from zero to eight detections, and the measured frame processing time is proportional to the number of detections within a frame. Figures 4.9 and 4.10 show the energy consumption of the Transmeta and Intel models when using memory-oblivious (top row) and memory-aware (bottom row) speed computations. When we compare the top rows of Figure 4.9 and Figure 4.10, the static scheme for Intel is flat for several consecutive deadline values; e.g., for deadlines larger

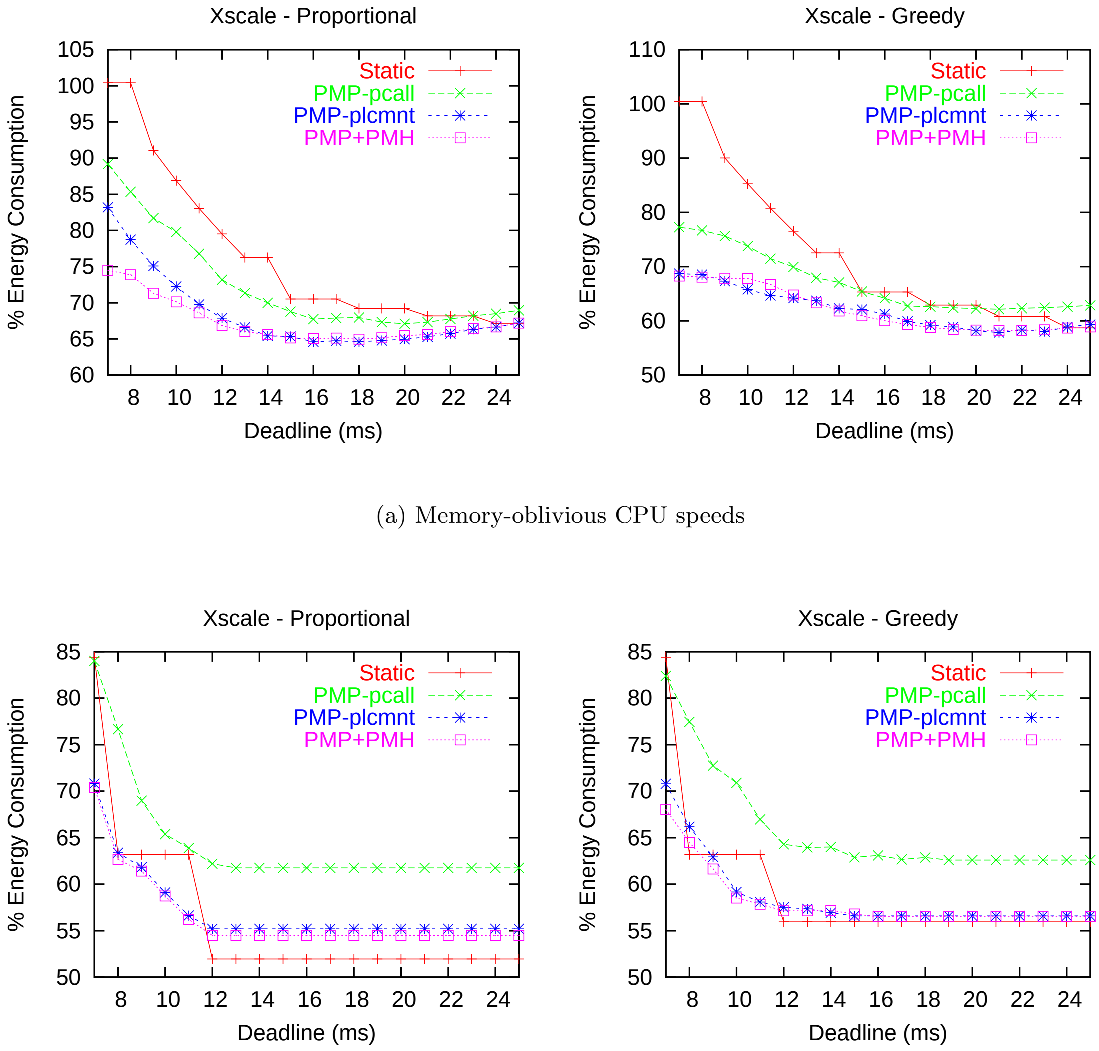
[3] The original code and data for this application were provided by our industrial research partners.



(a) Memory-oblivious CPU speeds

(b) Memory-aware CPU speeds

Figure 4.9: ATR: Total (CPU+memory) energy consumption normalized to no power management on Transmeta Crusoe.



(a) Memory-oblivious CPU speeds

(b) Memory-aware CPU speeds

Figure 4.10: ATR: Total (CPU+memory) energy consumption normalized to no power management on Intel XScale.

than 12 ms, the operating frequency is the same. This is because Transmeta has more power levels than Intel. All schemes experience increase in total energy consumption at relaxed deadlines when memory energy consumption exceeds CPU energy savings. At tight deadlines (up to 8 ms), the static scheme for the Transmeta processor executes the application at the highest speed, $f_{max}$; this yields a higher-than-NPM energy consumption due to the overhead of the PMP executed at each data frame processing. This is not true for our Proportional and Greedy schemes because of the dynamic slack reclamation. On the other hand, the same workload constitutes only about 80% of the Intel processor load due to a higher $f_{max}$ than Transmeta. Next, we discuss the results for the memory-aware speed computations, since this is the novelty of this work.

When using the PMP-plcmnt and PMP+PMH schemes in the Transmeta model, Greedy and Proportional consume less energy than the static scheme because of dynamic slack reclamation. However, when using the Intel model, static power management may consume less energy for deadlines beyond 12 ms. At those deadline values, the dynamic schemes operate most of the time on the same frequency as static power management. These schemes can not operate at a lower performance level; however, the overhead of code instrumentation and voltage scaling increase the energy consumption of the dynamic schemes with respect to the static scheme.

The energy savings of the Greedy scheme relative to the Proportional scheme vary based on the deadline. When deadlines are short, Greedy consumes less energy than Proportional. Because Greedy is more aggressive in using slack than Proportional, it is more likely to select a lower speed earlier in the execution than Proportional. Once Greedy transitions to a low speed, the dynamic slack reclaimed from future intervals helps Greedy stay at a relatively low speed later in the execution. On the other hand, Proportional selects a medium range of speed levels throughout the execution. When increasing the deadline (more than 10 ms), Greedy exploits most of the static slack early in the execution to reach low speeds. However, the amount of dynamic slack generated later in the execution is not large enough to maintain these low speeds. Thus, the power manager needs to increase the speed causing Greedy to consume more energy than Proportional. In the case of the Intel processor, Greedy's energy consumption is lower than Proportional at more relaxed deadlines ($\geq$15 ms). This is due to the minimum speed level that prevents Greedy from exploiting all the slack at the beginning of execution.

In the Intel case, the static scheme outperforms the Greedy scheme for 12-16 ms deadlines. This is because most of the speeds computed by Greedy are slightly higher than 150MHz but can only operate on 400 MHz, which is the same operating frequency for static power management in this deadline range. Hence, both schemes run with almost the same power but because of the overhead incorporated with the dynamic PMP-only schemes, Greedy consumes more energy than the static scheme.

The energy consumption of the PMP-plcmnt scheme is close to our PMP+PMH scheme for ATR. There are two reasons for this result: high overhead because of the large number of called procedures and the higher accuracy of the finer-granularity PMP invocation. It is also the case that most of the executed PMPs do not lead to a speed change, and thus there is no energy savings achieved from executing these PMPs.

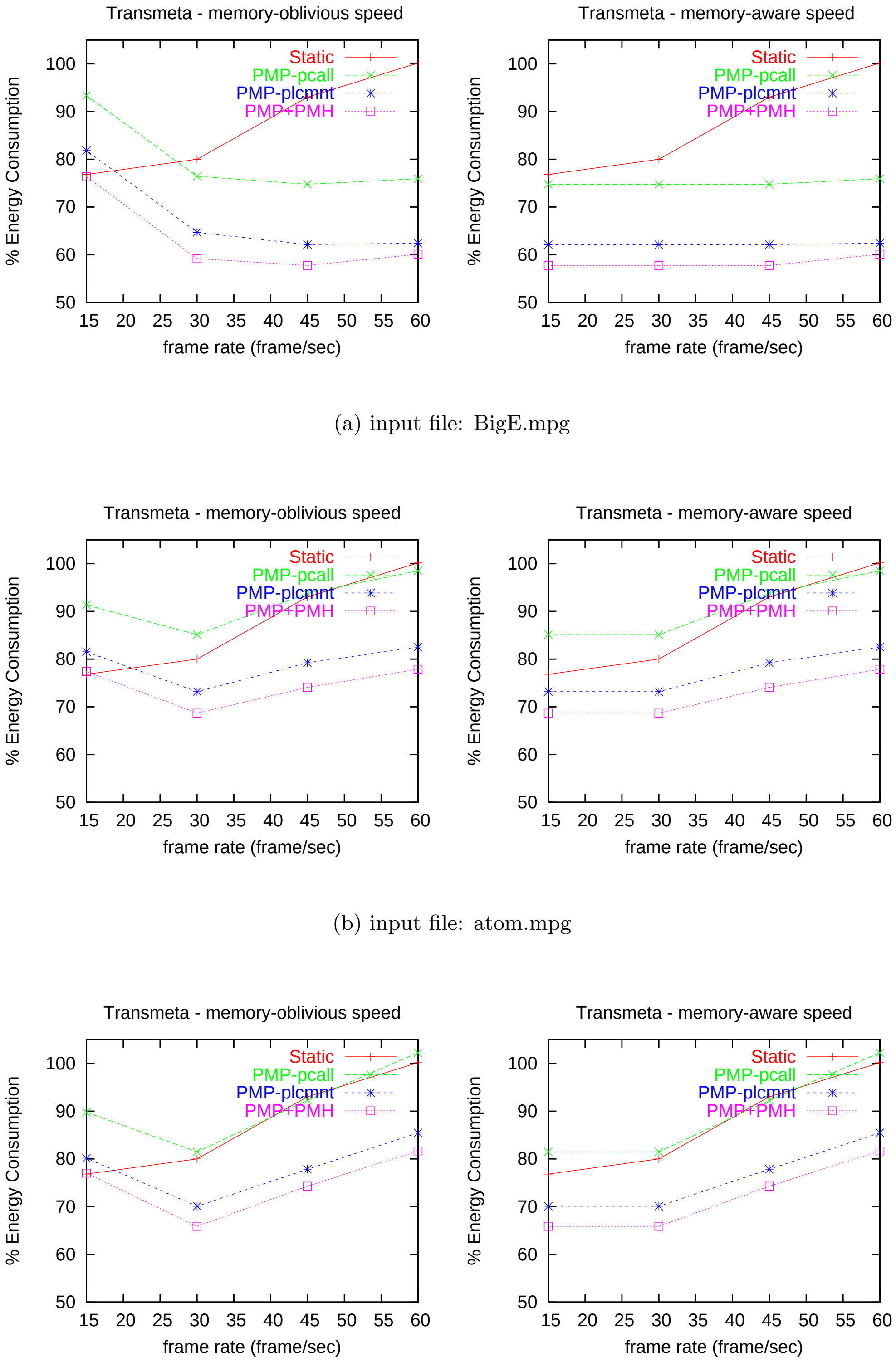
**MPEG video decoder.** We collected timing information about the MPEG2 decoder [81] using a training data set of six movies and tested it on 20 different movies. Our scheme inserts PMHs in the decoder's code based on profile information about frames of the training set movies. We run experiments for four different frame rates: 15, 30, 45 and 60 frame/sec that correspond to deadlines 66, 33, 22, and 16 ms per frame.

Figures 4.11 and 4.12 show the normalized energy consumption for three of the test movies for Transmeta and Intel models. For presentation simplicity, we only show the energy consumption for the Greedy scheme because Greedy outperforms the Proportional scheme. Similar to the ATR results, our proposed PMP+PMH scheme consumes less energy than the other schemes. Unlike the ATR results in the memory-oblivious speed computation case, the energy trade-off is less obvious because most of the time the computed speed is higher than the break-even frequency.

The MPEG decoder code includes mainly a set of nested loops, most of which have large variation in their execution times. As a result, in our scheme, the number of executed PMHs is much larger than the invoked PMPs. Hence, the $WCR$ estimation is improved over having only one PMH for each PMP invocation. In comparison with the PMP-plcmnt technique (replacing the PMHs with PMPs), PMP execution's energy overhead is higher, which overshadows the energy saving from the few extra speed adjustments. This is especially true for the Transmeta model. Because of the large number of speed levels that are close in range, this creates the chance for large number of speed changes in PMP-plcmnt. Each speed change results in small energy savings compared to the PMP energy overhead. The PMP-pcall scheme has the highest energy consumption due to the large number of procedure calls within loops in this application.

**Sub-band Tuner.** A sub-band tuner[4] is a signal processing application that accepts signals as input events. It uses time and frequency domain analysis to extract and process signals of interest,

[4] The original code and data trace files for this application were also provided by our industrial research partners.

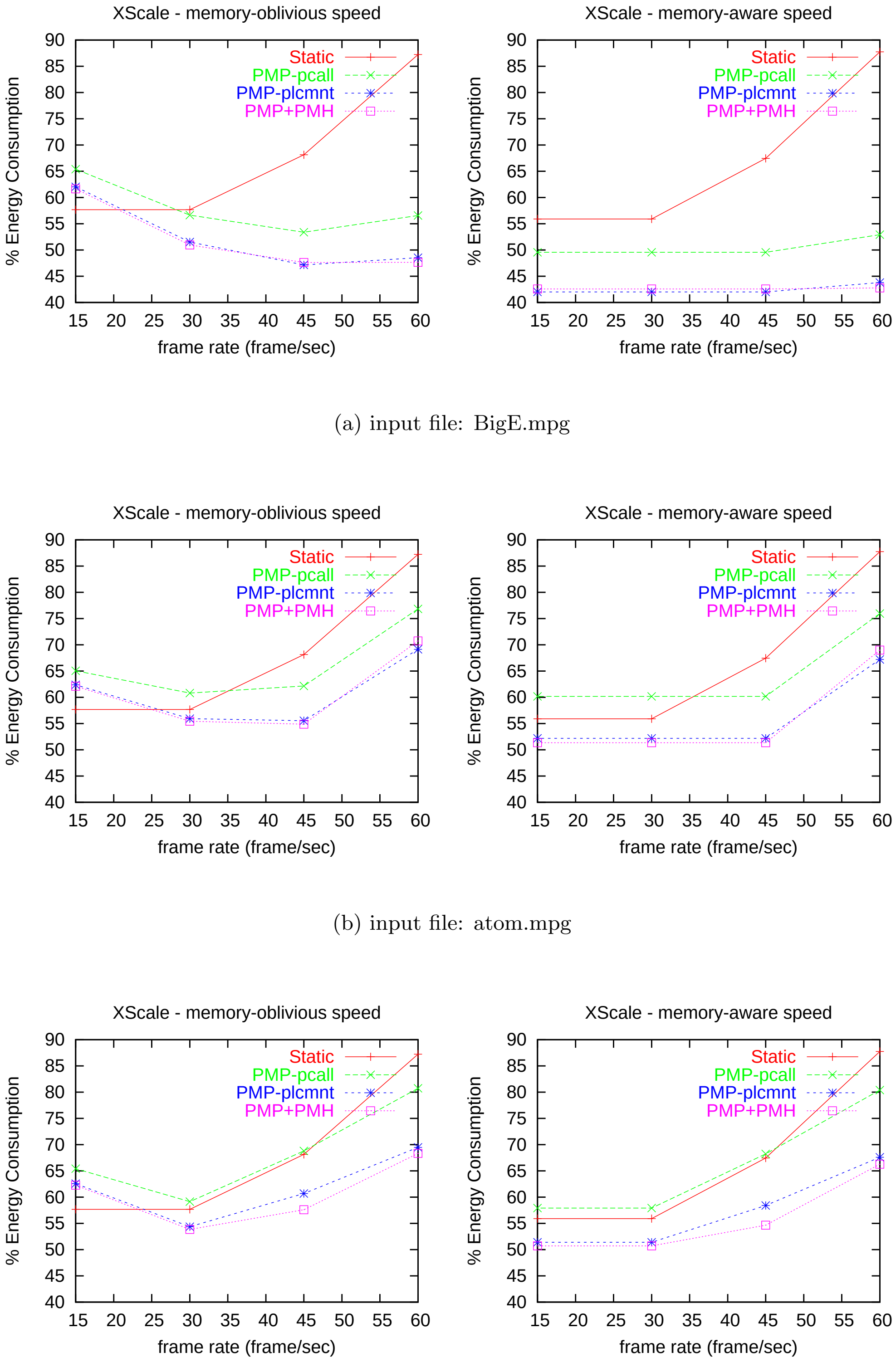


(a) input file: BigE.mpg

(b) input file: atom.mpg

(c) input file: robot.mpg

Figure 4.11: MPEG2 decoder: Total energy consumption normalized to no power management scheme on Transmeta Crusoe.



(a) input file: BigE.mpg

(b) input file: atom.mpg

(c) input file: robot.mpg

Figure 4.12: MPEG2 decoder: Total energy consumption normalized to no power management scheme on Intel XScale.

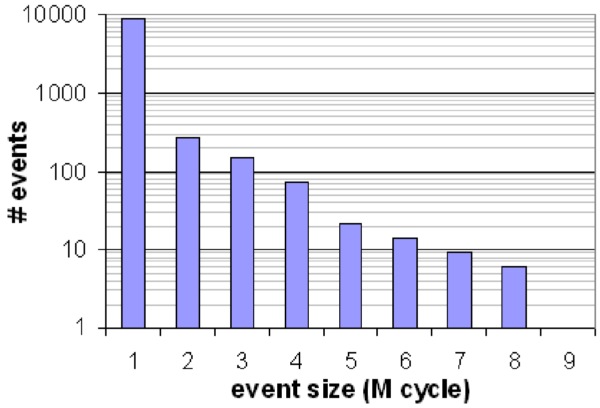


Figure 4.13: Size distribution (in million cycles) for the sub-band filter events.

called peaks and ridges, from those events. The number of cycles per event processed varies based on the number of peaks and ridges detected in each event but the deadline for each event is fixed. We experimented with a trace of 9,853 events. From the collected timing information about each event processing time, we noticed a large variation between the average and worst case execution times. An event's average execution time is 200 times smaller than its worst case execution time. The size distribution is shown in Figure 4.13 (note the logarithmic scale for the Y-axis). To make good use of the dynamic slack for the average event, the scheme tailors the frequency of invoking a speed-change (PMP-interval) based on the event's average execution time, and PMHs are placed accordingly. As a result, events larger than the average size would experience more PMH executions and speed-change invocations. This effect is prominent in case of very long events.

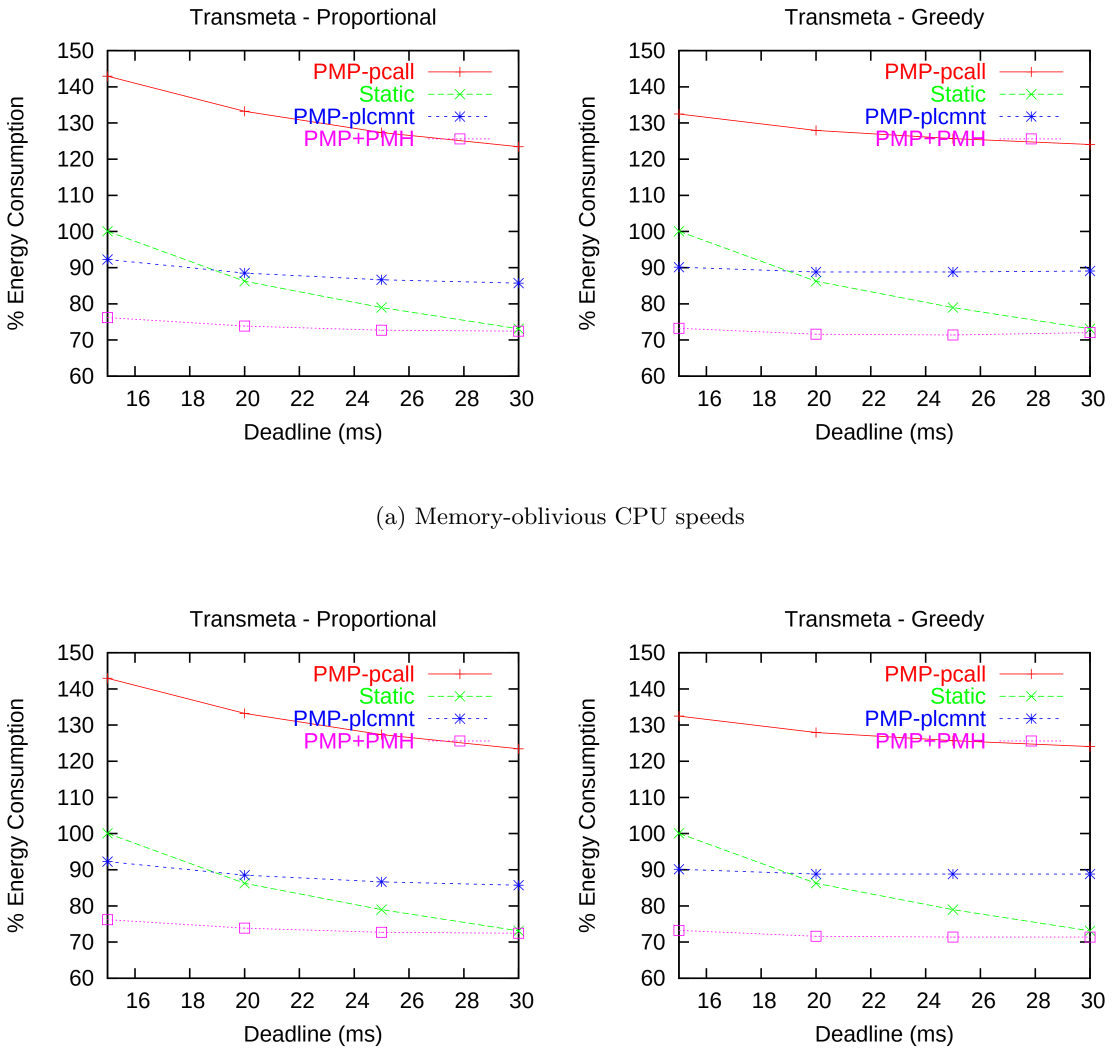
Figures 4.14 and 4.15 show the energy consumption for the tested schemes. Among the dynamic schemes, PMP+PMH performs best. When comparing PMP+PMH with static, PMP+PMH performs well in high system loads (small deadlines). As the system load decreases, the difference in energy consumed by static and PMP+PMH decreases. For the Intel model, the static scheme performs better than PMP+PMH for very small loads, due to the large difference between the lowest two speed levels (400 MHz and 150 MHz). When the static operates at 400 MHz, the dynamic schemes may have dynamic slacks to operate on less than this frequency but not enough slack to operate at 150MHz; therefore, the 400MHz frequency is selected to avoid missing deadlines. The difference in energy consumption is due to the extra instrumentation added by the dynamic schemes; this is particularly true when there is a large variation in the execution times among different events as with the sub-band tuner. In the Transmeta model, this does not happen since there are several frequencies between 200MHz and 400MHz, allowing for the dynamic schemes to actually execute at lower frequencies.

Although PMP-plcmnt performs almost as well as PMP+PMH for the ATR and MPEG benchmarks, PMP-plcmnt performs much worse for this application. This is due to the large variation in execution times described earlier that required inserting relatively large number of PMHs. The overhead effect is significant in PMP-plcmnt, because the amount of dynamic slack between 2 consecutive PMPs is too small to compensate this overhead. Table 4.8 compares the two techniques in terms of the number of executed PMHs, PMPs and actual speed change that takes place. We conclude that the work done by most of the executed PMPs (in PMP-plcmnt) was not useful. In contrast, the overhead of placed PMHs in our scheme is minimal (few cycles) compared to PMP-plcmnt. Thus, the total energy consumption using PMHs is much less than inserting PMPs directly in the code.

**4.6.0.2 Run-time Overhead** Our scheme introduces overhead due to PMPs and PMHs. In this section, we further investigate the impact of each on the overall performance (number of cycles and number of instructions).

**Number of instructions.** Increasing the number of instructions in the application by inserting PMPs and PMHs could have a negative effect on (1) the cache miss rate and (2) the total number of executed instructions. Both these factors could degrade performance. We measured their effects through an implementation in SimpleScalar. In our scheme, the measured effect of increasing the code size on the instruction cache miss rate is minimal since the inserted PMH code is very small. We did not measure a significant increase in cache misses (instruction and data) for our benchmarks, hence we do not expect a negative effect on the memory energy consumption due to our scheme. A static PMH takes 36 instructions to execute. Index-controlled PMHs are reduced to static PMHs inserted inside the loop with the addition of a division operation executed only once for each loop to compute the size of a loop body. The value of $wcr_i$ is decremented in each iteration by the size of the loop body. In comparison, our measurements indicated that a PMP takes from 74 to 463 instructions to compute and select a speed (excluding the actual voltage change).

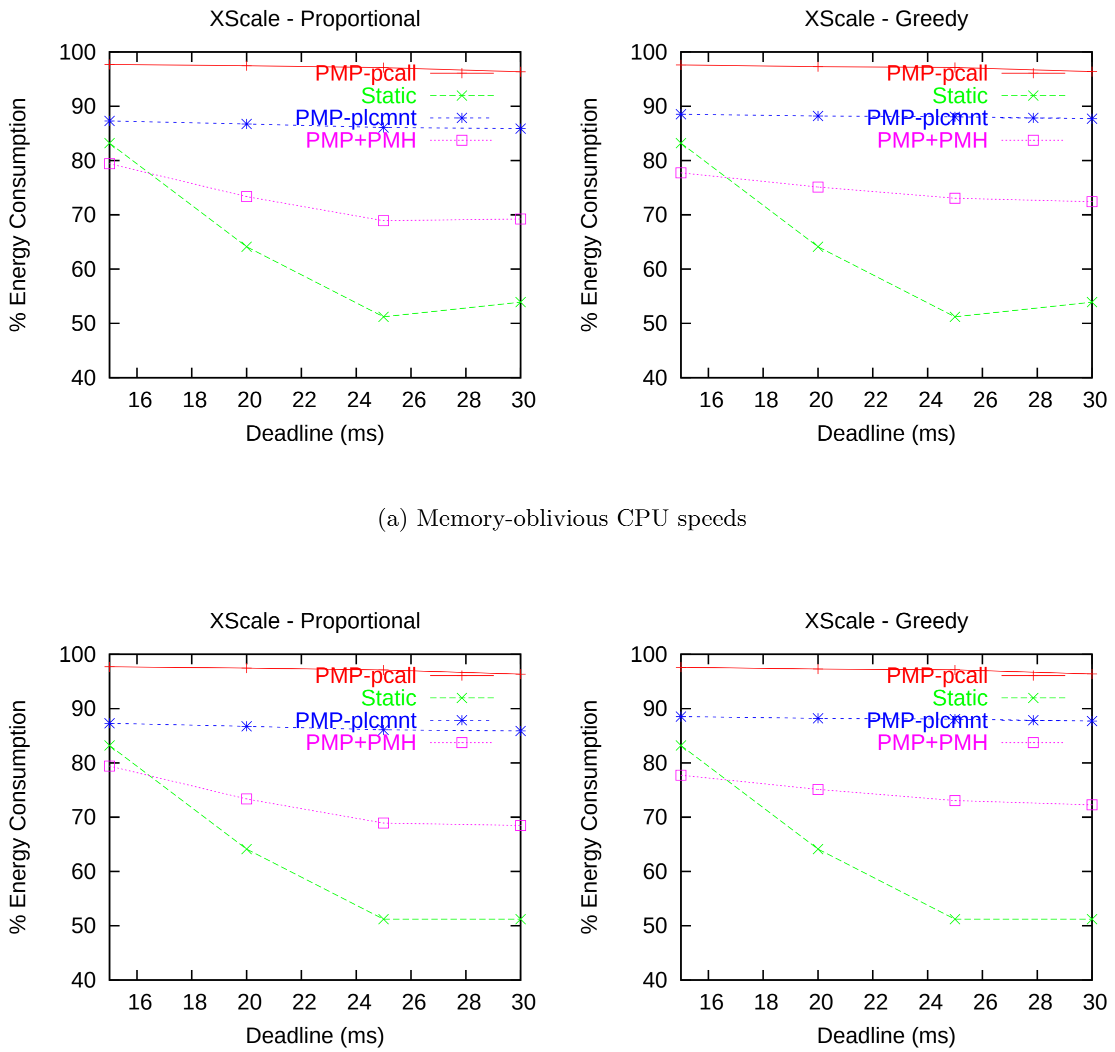
The number of executed instructions is kept minimal by (1) invoking the PMP at relatively large PMP-intervals (it ranges from 50 K to 580 K cycles in the presented applications) and (2) avoiding



(a) Memory-oblivious CPU speeds

(b) Memory-aware CPU speeds

Figure 4.14: Sub-band Tuner: Total energy consumption normalized to no power management scheme on Transmeta Crusoe.



(a) Memory-oblivious CPU speeds

(b) Memory-aware CPU speeds

Figure 4.15: Sub-band Tuner: Total energy consumption normalized to no power management scheme on Intel XScale model.

Table 4.8: The number of executed PMPs, PMH and speed changes for the sub-band tuner.

| | no. PMHs executed | no. PMPs executed | no. speed changes |
|---|---|---|---|
| PMP-plcmnt | – | $\approx 19 \times 10^6$ | $\approx 13.5 \times 10^3$ |
| PMP+PMH | $\approx 19 \times 10^6$ | $\approx 77.5 \times 10^3$ | $\approx 11.8 \times 10^3$ |

the excessive insertion of PMHs. For example, in the ATR application, for each PMP-interval, only 505 extra instructions are executed on average (including all executed PMHs and a PMP in this PMP-interval[5]). The total increase in the number of executed instructions due to the overhead is 0.05% for ATR, 0.25% for MPEG, and 3.7% for sub-band tuner. With respect to other intra-task DVS schemes, like PMP-plcmnt, the average increase in the number of executed instructions is <1% in ATR, 1.4% in MPEG, and 47% in sub-band tuner. The large increase in the number of executed instructions in sub-band tuner when using PMP-plcmnt is due to the excessive execution of PMPs (as described in Section 4.6.0.1).

**Number of cycles.** The extra inserted code (PMHs and PMPs) increases the number of cycles taken to execute an application. The average execution of each PMH takes about 13 cycles and PMP takes from 22 to 229 cycles to execute[6], excluding the actual voltage change). The total increase in the number of cycles ranges between 0.19% to 0.4% for ATR, 0.4% to 1.7% for MPEG, and 4.6% to 4.8% for sub-band tuner (the range corresponds to the low to high deadlines selected). In comparison to PMP-plcmnt, the average increase in the number of executed instructions is <1% in ATR, 1.5% in MPEG, and 64% in sub-band tuner. When using PMP-plcmnt, the execution of a large number of PMPs in sub-band tuner causes a relatively larger increase in the number of cycles.

We also note that the total overhead, in terms of number of cycles, decreases when the processor frequency is decreased. This is because the memory latency becomes relatively small when compared to the CPU frequency; that is, the CPU stalls for fewer cycles (but still stalls for the same amount of time) waiting for data from memory.

## 4.7 CONCLUSION

In this chapter, we present a memory-aware collaborative compiler-operating system intra-task DVS scheme for reducing the energy consumption of time-sensitive embedded applications. The operating system periodically invokes Power Management Points (PMPs) to adapt processor performance based on the dynamic behavior of an application and memory energy consumption. Information about run-time temporal behavior of the application is gathered by very low cost power management hints (PMHs) inserted by the compiler in the application. We describe our collaborative

[5] An inserted PMH can be executed more than once in a single PMP-interval, for example if placed inside a loop.
[6] Note this is a 4-way architecture, executing about 2 instructions per cycle.

scheme and its advantages over a purely compiler-based approach. We also present an algorithm for inserting PMHs in the application code to collect accurate timing information for the operating system. Considering the relatively expensive overhead of a voltage/speed-change, our scheme reduces the energy consumption by controlling the number of speed-changes based on application behavior. The algorithm computes the break-even frequency, which acts as a limitation on the CPU operating frequency in order to avoid extra energy consumption in other subsystems. Finally, we present results that show the effectiveness of our scheme. Our results show that our scheme is up to 58% better than no power management and up to 45% better than static power management on several embedded applications.

# 5.0 INTEGRATED DVS POLICIES FOR CPU AND CACHE POWER MANAGEMENT

## 5.1 INTRODUCTION

Embedded systems are evolving to accommodate a new and diverse set of applications. Such applications require increased processor computational power, larger storage capacity, and longer operation time. The advancement in processor technology creates the opportunity for embedded processors to approach the performance of general purpose processors by adopting performance solutions like large caches, superscalar, and multiple cores. However, adopting most of these solutions leads to more power consumption. Unfortunately, with the plethora of embedded system designs and their applications, it is hard to construct a power management policy that can be directly applied to a variety of embedded systems.

Multiple Clock Domain chip design has been proposed to allow fine grain power management of each domain, especially when using dynamic voltage and frequency scaling (DVS) [82]. Since each domain has its own clock and voltage (i.e., independent of the other domains), DVS can be applied in each domain for an extra level of power management (rather than applying DVS at the chip level). Power and energy can be reduced with minimal impact on performance by dynamically reducing the clock speed and voltage in domains with low activity.

In this chapter, we introduce the idea of managing the power consumption of more than one component using integrated DVS scheme, *IDVS*. We focus on the CPU core and cache power management. We first present a general heuristic based online-IDVS and show its advantage over a DVS policy that locally manages the CPU and cache frequency independent of each other. We then propose a machine learning approach, *ML-IDVS*, which can automatically synthesize IDVS policies that efficiently manage power in multiple domains. Our ML-IDVS tailors the generated polices to the behavior of a class of applications running on the system under consideration.

The contribution of this part of the work is fourfold. First, we identify a significant inefficiency in current online DVS policies, and show the sources and implications of this inefficiency. Second, we propose a heuristic-based online integrated DVS policy that adapts the core and L2 cache speeds in a way that avoids these inefficiencies, taking into account domain interactions. Third, we present a more flexible machine learning based integrated DVS approach. The approach automatically generates integrated DVS policies customized for a given class of applications running on a given system. Fourth, we experimentally evaluate the two integrated DVS techniques against a local-DVS policy, which forms its decisions based on local information collected about each domain. Both integrated DVS techniques show positive gains with respect energy-delay product with minimal impact on performance.

The chapter is organized as follows. We first introduce the idea and motivation behind IDVS in Section 5.2. We then present our online-IDVS in Section 5.3. Its counterpart, the ML-IDVS approach is presented in Section 5.4. Detailed evaluation of both techniques is presented in Section 5.5.

## 5.2 INTEGRATED DVS (*IDVS*) POLICY

### 5.2.1 Motivation

A typical application goes through phases throughout its execution. An application has varying cache/memory access patterns and CPU stall patterns. In general, application phases correspond to loops, and a new phase is entered when control branches to a different code section. We are especially interested in managing the power of the CPU-core and the on-chip L2 cache, as they consume a large fraction of the total power in current processors. For this, we characterize each code segment in a program using performance monitors that relate to the activity in each of these domains. Figure 5.1 shows the variations in three performance counters as examples of monitors that can be used to represent a program behavior. The figure shows: cycles per instruction (CPI), number of L2 accesses per instruction (L2PI), and memory accesses per instruction (MPI). CPI and L2PI are selected as indications of the amount of workload in the CPU-core and L2 cache, respectively. On the other hand, L2PI and MPI can be used to indicate the idleness in the CPU core and the L2 cache domains, respectively.

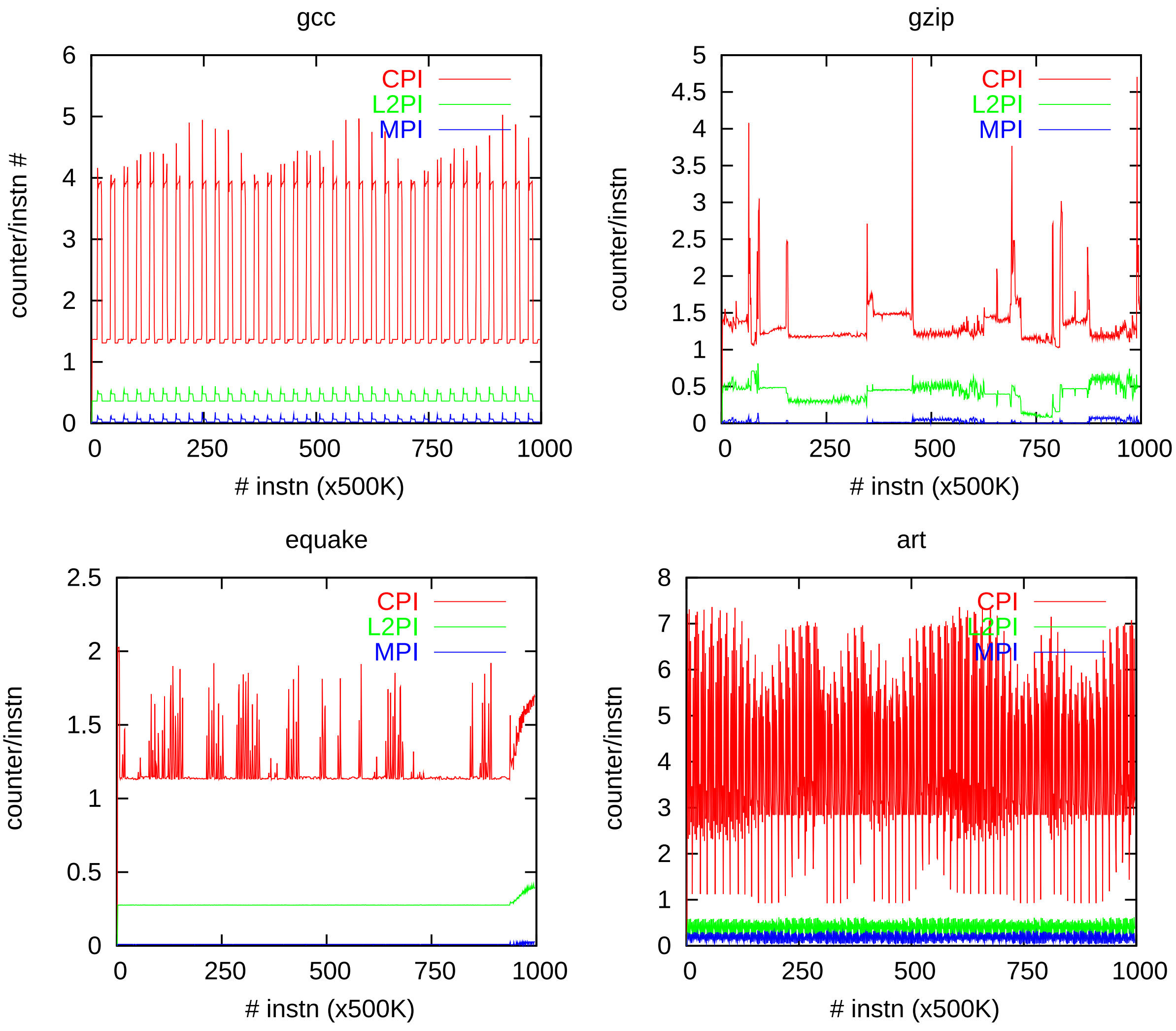


Figure 5.1: Variations in application phases throughout execution.

Intuitively, each program phase has a different requirement and preference toward a certain "configuration" of the CPU-core and L2 cache frequencies. For example, if a section of code is CPU bound, it will benefit from running at high CPU frequencies, and may be insensitive to L2 cache latency (as with most phases for *equake* in Figure 5.1). On the other hand, a memory bound phase benefits the most from reducing the gap between the core and cache performance (as with most phases for *art* in Figure 5.1). Typical applications alternate between CPU and memory bound phases (as shown for *gzip* in Figure 5.1). This is precisely the intuition behind our approach. Our goal is to construct an integrated CPU-core and L2 cache DVS, *IDVS*, policy that identifies application phases and selects appropriate frequencies for the CPU and L2 cache domains for each code section.

The "best frequencies" to use for the CPU-core and L2 cache domains are defined in terms of some optimization metric. There are three natural metrics: energy, performance, and energy-delay product. When the metric is energy, it would seem that the most energy-efficient frequencies are the minimum ones, due to the well-known quadratic relationship between frequency and power [83]. However, when looking at the system as a whole, this is no longer true [84]. Reducing the frequency of one component (e.g., the CPU-core) increases execution time, which increases the energy consumption of other components (due to static power dissipation for longer periods). Thus, the problem of identifying the optimal frequencies that minimize the system energy is far from trivial. When performance is the main requirement, we are interested in minimizing energy while maintaining execution time within a specified percentage of full performance (which corresponds to the highest frequencies available for both CPU and L2 cache). When energy and performance are equally important, the optimization metric is defined as the energy-delay product.

### 5.2.2 Integrated DVS architecture

Figure 5.2 shows a schematic diagram of an example processor with two domains and a microcontroller that manages the voltages of the domains according to our IDVS policies. During runtime, the microcontroller periodically monitors the activity in each domain by recording a set of performance counters. The microcontroller executes the policy to determine the best frequency combination based on the values of the latest performance counters read.

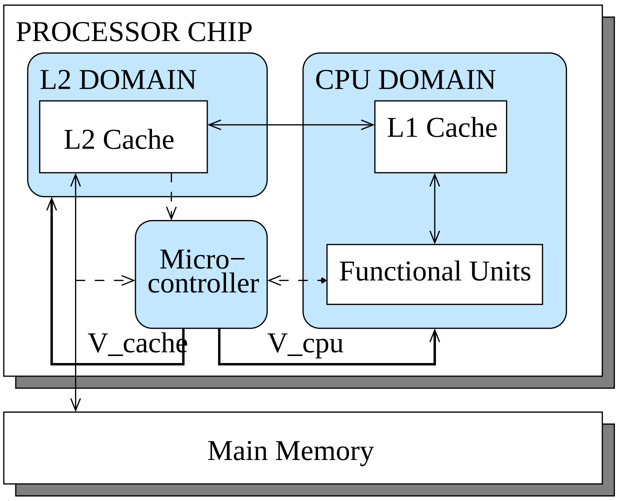


Figure 5.2: Example of an MCD processor design with integrated DVS control.

### 5.2.3 MCD inter-domain interactions

Applying DVS independently in an MCD processor creates domain interactions that may negatively affect the performance and/or energy of other domains. We present an example to illustrate the cause of these effects and their undesired implications; the reader is encouraged to follow the numbered steps in Figure 5.3. (1) Assume an MCD processor is running an application that experiences some pipeline stalls (e.g., due to branch misprediction). The increased number of stalls results in reduced IPC. (2) The local DVS policy triggers a lower speed setting in the core domain. Slowing down the core will reduce the rate of issuing instructions, including L2 accesses. (3) Fewer L2 accesses per interval causes the local policy to lower the speed in the L2 cache domain. (4) This, in turn, increases the cache access latency, which (5) causes more stalls in the core-domain. Hence, this interaction starts a vicious cycle, which spirals downward.

The duration of this positive feedback[1] depends on the application behavior. For benchmarks with low activity/load variations per domain, this feedback scenario results in low speeds for both domains. While these low speeds reduce power, they clearly hurt performance and do not necessarily reduce the total energy-delay product. Analogously, positive feedback may cause increased speeds in both domains, which potentially improves delay at the expense of increasing energy consumption.

[1] A positive feedback control is where the response of a system is to change a variable in the same direction of its original change.

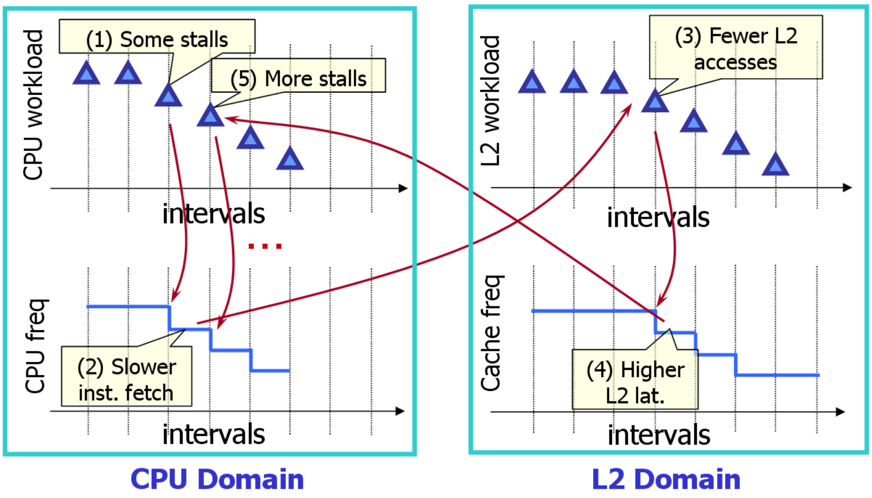


Figure 5.3: Example of positive feedback in local power management in each domain

These two scenarios illustrate that local DVS policies may not properly react to a domain's true workload.

These undesired positive feedback scenarios arise from the fact that the local DVS policy monitors only the local performance of a given domain to set its speed. This local information does not identify whether the source of the load variability is local to a domain or induced by other domains. As a result, the policy cannot take the correct action. In our example, the variation in IPC can be induced by local effects such as executing a large number of floating point instructions or suffering many branch mispredictions. Alternatively, effects from other domains such as higher memory and L2 access latency can induce variations in IPC. Although the effect on IPC is similar, the DVS policy should behave differently in these two cases.

Table 5.1: Percentage of time intervals that experience positive feedback scenarios in some MiBench and SPEC2000 benchmarks.

| adpcm_dec | adpcm_enc | basicmath | crc32 | gsm_toast | gsm_untoast | lame | rsynth |
|---|---|---|---|---|---|---|---|
| 12.28% | 7.13% | 2.40 % | 10.8% | 27.7% | 20.09% | 22.56% | 27.56% |
| bzip | equake | gcc | gzip | parser | twolf | vortex | vpr |
| 8.18% | 0.02% | 20.97 % | 4.59% | 9.06% | 25.79% | 6.5% | 16.12% |

To find how often applications experience such undesired positive feedback, we analyzed applications under Semeraro et. al's local DVS policy [35]. Table 5.1 illustrates the percentage of time intervals where positive feedback occurs in some MiBench and SPEC2000 benchmarks. The data is collected over a window of 500M instructions (after fast-forwarding simulations for 500M instructions). We divide the execution into 100K instruction intervals then count the percentage of consecutive intervals that experience positive feedback in both the CPU and L2 domains simultaneously. The table shows that some applications experience high rates of positive feedback, while others are largely unaffected. In the former (e.g., *gsm, lame, rsynth, bzip, parser,* and *vpr*), we expect that the local policy will result in relatively high delays or high energy because it reacts with inappropriate speed setting for more than 20% of the time.

To have a better indication of the core and L2 cache workloads, the policy has to be aware of the status of both domains, because each domain may indirectly influence the workload in the other domain. This motivates the need for run-time policies that take into account the core and the L2 cache interactions to appropriately set the speeds for both domains in a way that minimizes total energy, delay or energy-delay product.

## 5.3 ONLINE IDVS POLICY

In our first proposed *IDVS* policy, we monitor the IPC and the number of L2 accesses with performance counters [85]. The speeds are driven by the change in the *combined* status of IPC and the number of L2 accesses in a given execution interval. The rate of increase or decrease in speed is based on the rate of increase or decrease in the monitored counters subject to exceeding a threshold as proposed by Zhu et al. [40]. We introduce a new set of rules (listed in Table 5.2) to be executed by the DVS policy for controlling the speeds. The symbols $\Uparrow$, $\Downarrow$, and $-$ depict an increase, decrease, and no-change in the corresponding metric. Columns 2 and 3 in the table show the change in the monitored counters while columns 4 and 5 show the action taken by the local policy. Columns 6 and 7 show the action taken by our online-IDVS.

Given the evidence from Table 5.1, we decided to focus on the positive feedback cases described in Section 5.2.3. These cases only cause a change in Rules 1 and 5 in Table 5.2, and maintain the other rules exactly the same. The policy only changes the rules when there is a simultaneous increase or decrease in IPC and L2 cache accesses. As a result, our policy requires minimal changes

Table 5.2: Rules for adjusting core and L2 cache speeds in local DVS policy and proposed policy.

| rule | Event to monitor | | Action by local-DVS policy | | Action by our integrated policy | |
|---|---|---|---|---|---|---|
| # | IPC | L2access | $V_c$ | $V_\$$ | $V_c$ | $V_\$$ |
| 1 | ⇑ | ⇑ | ⇑ | ⇑ | ⇓ | ⇑ |
| 2 | ⇑ | ⇓ | ⇑ | ⇓ | ⇑ | ⇓ |
| 3 | ⇑ | – | ⇑ | – | ⇑ | – |
| 4 | ⇓ | ⇑ | ⇓ | ⇑ | ⇓ | ⇑ |
| 5 | ⇓ | ⇓ | ⇓ | ⇓ | – | ⇓ |
| 6 | ⇓ | – | ⇓ | – | ⇓ | – |
| 7 | – | ⇑ | – | ⇑ | – | ⇑ |
| 8 | – | ⇓ | – | ⇓ | – | ⇓ |
| 9 | – | – | – | – | – | – |

to existing policies (i.e., it can be readily supported without any additional cost), yet it achieves better energy savings. Contrary to the local policy, which seems intuitive, our IDVS policy does **not** increase (or decrease) the speed if both counters show an increase/decrease during a given interval. Instead, the policy changes speeds as shown in the table. Next, we describe both cases and the reasons behind the counter-intuitive actions of our policy.

Rule 1 reacts to the first positive feedback case by reducing the core speed rather than increasing it, as in the local policy. This decision is based on the observation that the increase in IPC was accompanied by an increase in the number of L2 cache accesses. This increase may indicate a start of a program phase with high memory traffic. Hence, we preemptively reduce the core speed to avoid overloading the L2 cache domain with excess traffic. In contrast, increasing the core speed would exacerbate the load in both domains. We choose to decrease the core speed rather than keeping it unchanged to save core energy, especially with the likelihood of longer core stalls due to the expected higher L2 cache traffic.

In Rule 5, we target the second undesired positive feedback scenario where the local policy decreases both core and cache speeds. From observing the cache workload, we deduce that the decrease in IPC is not due to higher L2 traffic. Thus, longer core stalls are a result of local core activity such as branch misprediction. Hence, increasing or decreasing the core speed may not eliminate the source of these stalls. By doing so, we risk unnecessarily increasing in the application's execution time or energy consumption. Hence, we choose to maintain the core speed without any change in this case, to break the positive feedback scenario without hurting delay or energy.

### 5.3.1 Limitation of the online IDVS policies

There are two main shortcomings in an online heuristic IDVS policy. First, with a larger number of domains, it is harder to discover and model the interactions among domains. Some domain interactions are intuitive such as correlation between CPI and L2PI; however, others are very hard to model such as the interaction between the FPU and the branch prediction unit. Second, a policy is not customizable for a certain class of applications or systems. Optimizing the policy based the knowledge of the target system and the expected class of applications that run on this system is often more effective than using a general policy. The optimized policy can take into account the behavior of applications on the target system under consideration. To solve these problems, we present next a machine learning approach that generates IDVS policies that are customized for a given class of applications running on a target system.

## 5.4 MACHINE LEARNING BASED IDVS APPROACH

To overcome the shortcoming of online IDVS policies, we use a learning-based approach first introduced by Rusu et al. in [86]. We further developed the technique to apply for a realistic architectural model with multiple clock domains. We call the technique *ML-IDVS*. ML-IDVS uses supervised machine learning to automatically generate a custom power management policy for each set of applications. ML-IDVS provides a novel methodology to automatically derive an IDVS policy for CPU-core and L2 cache power management. The derived policy dynamically adapts the domains' voltages and frequencies to current workload in an MCD processor. Our approach identifies application phases at runtime and takes corresponding actions (i.e., setting the voltage and frequency of both the processor and the L2 cache).

In ML-IDVS, the automated generation of power management policies relies on given system settings. Our ML-IDVS technique takes as an input a state description of the system, which includes the architectural and application behaviors, and an optimization criterion. Based on this input, it generates a custom policy for this particular system. ML-IDVS uses a supervised learning process on a set of representative training workloads to derive the DVS policy [87].

One of the advantages of using the ML-IDVS approach is its ability to automatically construct a power-management policy for different architectures (embedded and general-purpose) and different classes of applications. This technique also can be used to optimize for different metrics (such as energy and energy-delay product) that can be set by the user. This is useful with systems that use different optimization metric for different operational modes. For each mode, our approach derives a policy that can be loaded at the time of the mode switch to optimize the system for a given optimization criteria. All policies are derived using the same methodology.

Next, we present an overview of ML-IDVS in Section 5.4.1. We describe the essential phases of our approach for obtaining a policy using supervised learning technique in Section 5.4.2, followed by a discussion of some practical design issues in Section 5.4.3.

### 5.4.1 Overview of ML-IDVS approach

Our power management approach consists of two main stages. First, an offline stage where our ML-IDVS approach constructs a power management policy. Second, a run-time stage where an embedded microcontroller monitors the system (including the application behavior) and accordingly reacts by setting domain voltages as defined by the policy.

Figure 5.4.1 shows the information flow in ML-IDVS during the offline and online stages. In the offline stage, ML-IDVS learns the power management policy using sample applications. For this, the technique (a) analyzes the behavior of sample applications then (b) develops runtime DVS policy according to a given optimization metric. The analysis takes into account the architectural behavior while executing the given applications. ML-IDVS derives a policy using a supervised machine learning technique introduced in [86]. ML-IDVS derives separate policies for different optimization metrics. For systems that operate at different modes, the operating system loads the policy that corresponds to the current system mode. Only one analysis phase is needed for all optimization criteria. During runtime, the microcontroller executes the policy to determine the best frequency combination based on the values of the latest performance counters read. In the next section we focus on the policy construction process.

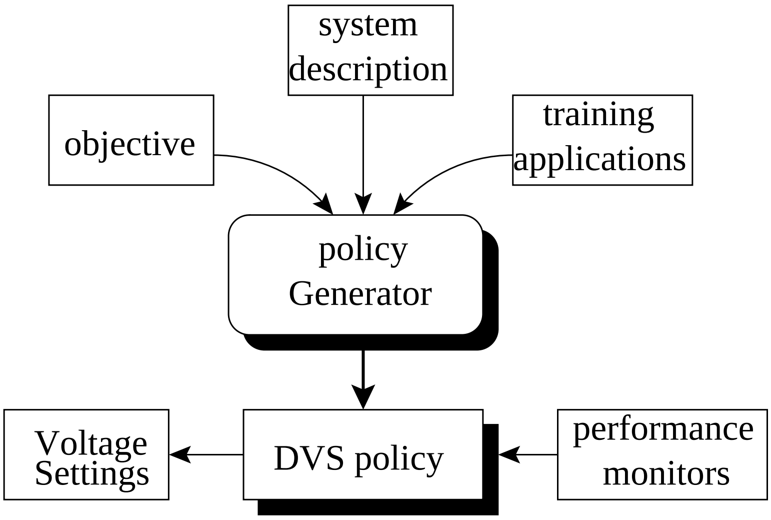


Figure 5.4: Information flow in ML-IDVS

### 5.4.2 Construction of IDVS Policy

To automatically construct a power management policy, ML-IDVS relies on a description of the *state* of the system under different program behaviors and run-time system characteristics. A program behavior description captures the instruction-level parallelism and cache/memory demands of the application. A separate run-time characteristics description captures program latencies during a given program phase. The goal is to identify for each possible system state the correct *action*. For example, an action determines how the CPU-core and L2 cache frequencies should be adjusted to minimize the energy-delay product. The power manager outputs a policy that maps states to actions with the objective of optimizing a performance metric (for example, energy and/or delay). The power manager creates the DVS policy in two stages: data analysis and policy learning. Figure 5.5 illustrates the main tasks for deriving the power management policy. Below, we describe the tasks performed in each of these two stages.

**5.4.2.1 Stage I: Data analysis** In this stage, we represent all possible system states by a set of performance counter readings. The power manager discovers the best operating frequencies for each state through an exhaustive search of the training data. The power manager then uses the training data to construct a table that maps a state to a CPU and cache frequency combination.

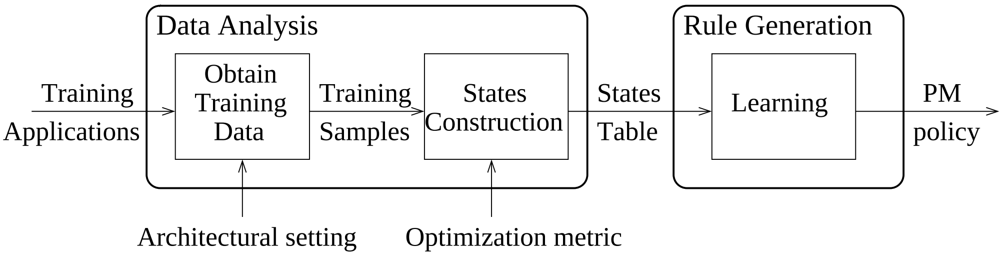


Figure 5.5: Stages for automatic DVS policy generation.

### State representation

To train our policy, we need a representation of the system state that encapsulates the program and architectural behavior with simple performance metrics. We can select the CPI, L2PI, and MPI, which can be determined from hardware performance counters. The CPI indicates the CPU utilization; however, it does not by itself fully describe program phases. For example, a high CPI corresponds either to a high cache miss ratio, high cache access latency, or long instruction latencies (such as division). Adding both L2PI and MPI into the state description can identify the reason behind the high/low CPI, and hence more fully describe application behavior. However, the CPI, L2PI and MPI do not take into account the effective latency of instruction execution or cache accesses (hits and misses), and to fully characterize the program, these latencies have to be factored into the state description. We describe the effective latency as a tuple of CPU-core and L2 cache frequencies. If there exist a technique that changes the memory speed, we can include the memory frequency in the state description. However, we still add MPI to the state description because of the significance of the memory traffic in defining an application behavior, especially memory-intensive ones.

Since we do not modify the memory speed, we do not include the memory frequency in our state representation. This representation (CPI, L2PI, MPI, CPU-core and cache frequencies) not only captures the timing characteristics of a given segment, but it also provides an estimate of the total energy, since it is closely related to the operating frequencies.

### Obtaining Training Data

The data used to learn the policy is obtained from training benchmarks in the following manner.

Let $N_{cpu}$ and $N_\$$ be the number of CPU and cache frequencies, respectively. We run all training benchmarks at all CPU and cache frequency combinations ($N_{cpu} \cdot N_\$$ combinations). A sample is defined as a continuous code section of fixed number of instructions equal to $size$. Thus, each training benchmarks with a total of $inst$ instructions will generate $K = inst/size$ code samples for one particular CPU/cache frequency, and $N_{cpu} \cdot N_\$ \cdot K$ samples for all frequency combinations. We denote the samples by $S_{ij} = \{CPI_{ij}, L2PI_{ij}, MPI_{ij}, M_{ij}\}$, where $i$ and $j$ are frequency indexes ($0 \leq i < N_{cpu}$ and $0 \leq j < N_\$$). Let $M_{ij}$ be the metric to be optimized. Based on the user setting, the ML-IDVS substitutes $M_{ij}$ by optimization metric of choice, for example, the segment's energy consumption ($E_{ij}$) or energy-delay product ($ED_{kij}$).

Thus, a state is described by five parameters: CPI, L2PI, MPI, CPU-core frequency and L2 cache frequency. CPI, L2PI, and MPI are continuous variables and need to be discretized. We choose a number of discrete intervals, discretization bins, in a way that the sample densities in each bin are almost equal. For example, because of the L2 cache efficiencies in current designs, if most samples have low L2PI, it consequently creates more L2PI ranges with lower values (i.e., finer granularity where the density is higher).

As an illustrative example, consider a system with two CPU and L2 cache frequencies: 0.5GHz and 1GHz. For ease of presentation, we use values for two performance monitors: CPI and L2PI. To reduce the state space, we discretize CPI and L2PI values into two bin intervals. CPI bins cover values [0,1.39] and ]1.39,∞], and L2PI bins cover [0,0.02] and ]0.02,∞]. Each sample lists its CPI and L2PI bin indexes and the energy-delay product when running at each frequency combination. Table 5.3 shows the collected samples in our example.

### $ST$ construction

After collecting data for all samples, $S_{ij}$, we construct the state table $ST$, which contains the correct action for each state as determined by the training data. $ST$ includes all possible system states. Let $B_{cpi}$, $B_{l2pi}$, and $B_{mpi}$ be the number of discrete values (bins) of the CPI, L2PI, and MPI, respectively. The state table is defined as: $ST[CPI_c][L2PI_l][MPI_m][i][j]$, where $CPI_c$, $L2PI_l$, and $MPI_m$ are the discretized values and $0 \leq c < B_{cpi}$, $0 \leq l < B_{l2pi}$, and $0 \leq m < B_{mpi}$ . For each state we want to determine the action that minimizes a user-selected optimization metric; for example, the energy-delay product.

We construct the table as follows. Since for each section of code all the possible frequency combinations are available, the best action can be determined by adding the energy-delay product of each

Table 5.3: Eight training samples: CPI (C), L2PI (L) and energy-delay product (ED) at all frequency combinations.

| $f_{cpu}$ | 0.5GHz | | | 0.5GHz | | | 1GHz | | | 1GHz | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $f_{\$}$ | 0.5GHz | | | 1GHz | | | 0.5GHz | | | 1GHz | | |
| s | C | L | ED | C | L | ED | C | L | ED | C | L | ED |
| 1 | 0 | 1 | 200 | 0 | 1 | 354 | 1 | 1 | 183 | 1 | 1 | 187 |
| 2 | 0 | 1 | 242 | 0 | 1 | 428 | 1 | 1 | 223 | 1 | 1 | 226 |
| 3 | 0 | 0 | 436 | 0 | 0 | 768 | 1 | 0 | 395 | 1 | 0 | 403 |
| 4 | 0 | 1 | 274 | 0 | 1 | 481 | 1 | 1 | 252 | 0 | 1 | 250 |
| 5 | 0 | 0 | 473 | 0 | 0 | 826 | 1 | 1 | 430 | 0 | 0 | 430 |
| 6 | 1 | 1 | 330 | 0 | 1 | 588 | 1 | 1 | 309 | 1 | 1 | 317 |
| 7 | 1 | 0 | 361 | 0 | 0 | 642 | 1 | 0 | 327 | 1 | 0 | 339 |
| 8 | 1 | 0 | 401 | 0 | 0 | 709 | 1 | 0 | 363 | 1 | 0 | 374 |

sample running at the new frequency. Since different sections of code may have the same state, an array that accumulates all values for the same state is used: $Acc[CPI_{ij}][L2PI_{ij}][MPI_{ij}][i][j][x][y]$, where $CPI_{ij}$, $L2PI_{ij}$,$MPI_{ij}$, $i$, and $j$ are the state parameters and $x$ and $y$ are the new CPU and cache frequencies (that is, the action). For each training sample $S_{ij}$ and each possible action $x$, $y$ ($x$ is the next CPU frequency, $y$ is the next cache frequency), we update the array as shown in Figure 5.6.

The array *Acc* accumulates the values of the optimization metric, $M_{xy}$, for all training samples and under all possible actions. After updating the array for all training samples, the action for each state in *ST* is the one that optimizes the metric stored $M_{xy}$. To find the best action, we iterate over all possible state values. We couple each possible state with one of all the possible actions ($\langle new f_{cpu}^{x}, new f_{\$}^{y} \rangle$). We then search in *Acc* for any possible entry that matches each *state + action*. We then select the best action that achieves the optimum metric stored in the table. Hence, the construction of ST is subject to the optimization metric of the system. As an illustrative example, Figure 5.7 shows the algorithm for constructing ST to minimize the energy-delay product. The resulting state table is a five dimensional table (three features and frequencies for two domains)

```
for all sample do
    for all CPU frequency, $f_{cpu}^{i}$, where $i = 0 \dots N_{cpu}$ do
        for all Cache frequency, $f_{\$}^{j}$, where $j = 0 \dots N_{\$}$ do
            for all future CPU frequency, $newf_{cpu}^{x}$, where $x = 0 \dots N_{cpu}$ do
                for all future Cache frequency, $newf_{\$}^{y}$, where $y = 0 \dots N_{\$}$ do
                    $Acc[CPI_{ij}][L2PI_{ij}][MPI_{ij}][i][j][x][y] += M_{xy}$
                end for
            end for
        end for
    end for
end for
```

Figure 5.6: *Acc* construction pseudocode

that represents all possible system states. $ST$ maps a state to the best frequency that optimizes the metric under consideration.

In Figure 5.7, Lines 1 and 2 iterate over all possible state values. Line 3 initializes the minimum energy-delay product, minEDP, and Line 4 initializes the best frequency combination (action) by equating it to the maximum CPU and cache frequencies, $\langle f_{cpu}^{max}, f_{\$}^{max} \rangle$. Line 5 iterates over all possible actions. Lines 6 to 9 find the action with the least energy-delay product among all possible actions. The best action resulting is stored in its corresponding location in ST in Line 10.

Table 5.4 shows the state table for the training samples shown in Table 5.3. State description is composed of CPI, L2PI and the two domain frequencies ($f_{cpu}$ and $f_{\$}$). The table shows the best frequency combinations ($newf_{cpu}^{x}$ and $newf_{\$}^{y}$) for each state description. Note that not all of the $ST$ states are populated (unpopulated states are marked by -).

Since the training data do not cover all possible states in the table (because some states may not be discovered from the training applications/data). We use a supervised machine learning algorithm to derive the DVS policy that can react (select new CPU and cache frequencies) to any possible system state. This includes actions to unseen states as well as already discovered states during training.

```
for all $CPI_c$, $L2PI_l$, $MPI_m$, where $c = 0 \ldots B_{cpi}$, $l = 0 \ldots B_{l2pi}$, $m = 0 \ldots B_{mpi}$ do
  for all current CPU $f^i_{cpu}$ and cache frequency, $f^i_{\$}$ where $i = 0 \ldots N_{cpu}$, $j = 0 \ldots N_{\$}$ do
    initialize minEDP
    initialize bestFreq = $\langle f^{max}_{cpu}, f^{max}_{\$} \rangle$
    for all next CPU frequency, $f^x_{cpu}$, and next cache frequency $f^y_{\$}$ where $x = 0 \ldots N_{cpu}$, $y = 0 \ldots N_{\$}$ do
      if minEDP $> Acc[CPI_c][L2PI_l][MPI_m][f^i_{cpu}][f^j_{\$}][f^x_{cpu}][f^y_{\$}]$ then
        minEDP = $Acc[CPI_c][L2PI_l][MPI_m][f^i_{cpu}][f^j_{\$}][f^x_{cpu}][f^y_{\$}]$
        bestFreq = $\langle f^x_{cpu}, f^y_{\$} \rangle$
      end if
      $ST[CPI_c][L2PI_l][MPI_m][f^i_{cpu}][f^j_{\$}]$= bestFreq
    end for
  end for
end for
```

Figure 5.7: *ST* construction pseudocode to minimize energy-delay product.

Table 5.4: Constructed $ST$ from samples in Table 5.3.

| State | | | | Action | |
|---|---|---|---|---|---|
| CPI | L2PI | $f_{cpu}$(GHz) | $f_{\$}$(GHz) | $new f_{cpu}$(GHz) | $new f_{\$}$(GHz) |
| 0 | 0 | 0.5 | 0.5 | 1 | 0.5 |
| 0 | 1 | 0.5 | 0.5 | 1 | 1 |
| 1 | 0 | 0.5 | 0.5 | 1 | 0.5 |
| 1 | 1 | 0.5 | 0.5 | 1 | 0.5 |
| 0 | 0 | 0.5 | 1 | 1 | 0.5 |
| 0 | 1 | 0.5 | 1 | 1 | 1 |
| 1 | 0 | 0.5 | 1 | - | - |
| 1 | 1 | 0.5 | 1 | - | - |
| 0 | 0 | 1 | 0.5 | - | - |
| 0 | 1 | 1 | 0.5 | - | - |
| 1 | 0 | 1 | 0.5 | 1 | 0.5 |
| 1 | 1 | 1 | 0.5 | 1 | 1 |
| 0 | 0 | 1 | 1 | - | - |
| 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 | 0.5 |
| 1 | 1 | 1 | 1 | 1 | 0.5 |

**5.4.2.2 Stage II: IDVS policy learning** There are many supervised learning techniques, including logistic classification, neural network, decision tree, and propositional rule. We prefer the propositional rule approach because it is more compact, more expressive, and more human readable than the other techniques. Furthermore, propositional rules are easy to implement in hardware. In fact, we tried all the aforementioned techniques on the training data and the propositional rule approach most closely models $ST$.

We use the Repeated Incremental Pruning to Produce Error Reduction (RIPPER) learner [88]. The RIPPER algorithm is known to achieve low error rates while being efficient on large data sets. RIPPER represents the collected states in the form of propositional (if-then) rules. Each rule specifies the desirable CPU frequency and cache frequency for the next program interval based on the current state. The learner is based on the Incremental Reduced Error learning IREP algorithm [89]. RIPPER repeatedly calls IREP to construct the rule set with low error rates.

IREP iteratively builds its rule set in a greedy fashion; that is, one rule at a time. IREP works in two phases: growing and pruning. First, it randomly partitions the data set in to two subsets: the growing and pruning sets. The rule growth phase constructs an initial rule set. It starts with an empty clause and then repeatedly adds sub-conditions to the antecedent. The sub-conditions maximize the coverage of the rule (represents more states). The stopping criterion for adding sub-conditions is either covering all the input states or not being able to improve the rule coverage. After growing a rule, the rule is immediately pruned in the pruning phase. Pruning is an attempt to prevent the rules from being too specific. IREP chooses the candidate literals for pruning based on a score that is applied to all the sub-conditions of the antecedent and evaluate the score using the pruning data. This process is repeated until all states are covered or the learned rules have very small error.

The resulting rules are generated in the form of: IF <*cond*> THEN <*set_freq*>, where *cond* is a conjunction of one or more of the following sub-conditions. $(CPI_{cur} \leq CPI_c)$, $(CPI_{cur} \geq CPI_c)$, $(L2PI_{cur} \leq L2PI_l)$, $(L2PI_{cur} \geq L2PI_l)$, $(MPI_{cur} \leq MPI_m)$, $(MPI_{cur} \geq MPI_m)$, $(c_f = i)$, and $(m_f = j)$ where $CPI_{cur}$, $L2PI_{cur}$, $MPI_{cur}$, $c_f$ and $m_f$ are the current CPI, L2PU, MPI, CPU frequency and cache frequency, respectively. *set_freq* specifies the value of the next CPU or cache frequencies. Rules learned from $ST$ in Table 5.4 are shown in Table 5.5. Note that the number of rules is very small because of the simplified system setting chosen in our hypothetical example.

Table 5.5: Example of a policy to minimize energy-delay product.

| # | Rule |
|---|---|
| 1 | if (L2PI $\geq$ 1) and (CPI $\leq$ 0) then $newf_{\$}$=1GHz |
| 2 | else $newf_{\$}$=0.5GHz |
| 3 | $newf_{cpu}$=1GHz |

### 5.4.3 Design Issues

**Feature Selection:** Ultimately, all DVS policy decisions are based entirely on the current system state. It is important, therefore, to characterize the state in terms of features which provide relevant information about the current application phase. Our hypothesis, based on architectural knowledge, is that CPI, L2PI, and MPI (along with the current CPU-core/cache frequencies) capture enough about the behavior of the application to make informed choices, while still being inexpensive to gather at runtime.

**Optimization metrics:** Once the training samples are collected, the data used to learn the DVS policy can be obtained for different metrics. The collection of training samples is a one-time step and is independent to the optimization metric. Thus, metrics can later be changed by simply updating $M_{kxy}$ in Figure 5.6. One of the strengths of our approach is that the power manager needs to train the system only once and therefore generate different policy for each optimization metric.

**Training applications:** Training applications are selected based on the diversity of the states each application can produce. Applications in the training set should exhibit a large number of different states to populate the majority of $ST$. The more $ST$ is populated, the more accurate the policy can derive actions for the unseen states. In general, it is desirable to use representative applications that include memory-bound and CPU-bound phases to cover more states in the table. Also, applications that have a large variation of the behavior in its phases, such as *gcc*, will highly contribute to $ST$ population.

## 5.5 EVALUATION

In this section, we evaluate the efficacy of our online-IDVS and ML-IDVS approaches. We compare both against a DVS policy that controls each domain independently. In both IDVS approaches, we evaluate the effect of considering domain interactions on reducing a chip's energy and energy-delay product.

### 5.5.1 Experimental setup

We use the SimpleScalar and Wattch architectural simulators with an MCD extension by Zhu et al. [40] that models inter-domain synchronization events and speed scaling overheads. To model the MCD design in Figure 2.1, we altered the simulator provided by Zhu et al. by merging different core domains into a single domain and separating the L2 cache into its own domain.

Since our goal is to devise a DVS policy for an embedded processor with MCD extensions, we use *Configuration A* from Table 5.6 as a representative of a simple embedded processor (SimpleScalar's StrongARM configuration [90]). We use MiBench benchmarks with the *long* input datasets. Since MiBench applications are relatively short, we fastforward only 500 million instructions and simulate the following 500 million instructions or until benchmark completion.

To extend our evaluation and verify if our policy can be extended to different arenas (in particular, higher performance processors), we also use the SPEC2000 benchmarks and a high-end embedded processor [40] (see *Configuration B* in Table 5.6). We run the SPEC2000 benchmarks with the *ref* data set. We use the same execution window (500M) for uniformity.

To show the benefit of accounting for domain interactions, we compare our IDVS techniques with a technique that uses a local policy in each domain similar to one presented in [91]. We call it local-DVS technique. The local-DVS policy periodically sets the speed of each domain independently based on the activity of the domain, which is measured through performance counters in that domain. We use IPC (or CPI) and L2PI as an indication of the activity in the core and L2 cache domains. Other performance counters are used to measure other domains' activity, such as number of integer and floating points instructions. The policy periodically monitors the variations in performance counter in each domain across interval periods. Based on the direction of change in a counter, the policy decides to change the speed of the corresponding domain. This scheme is efficient because it can be done locally and with very little overhead.

Table 5.6: Simulation configurations

| Parameter | Config. A<br>(simple embedded) | Config. B<br>(high-end embedded) |
|---|---|---|
| Dec./Iss. Width | 1/1 | 4/6 |
| dL1 cache | 16KB, 32-way | 64KB, 2-way |
| iL1 cache | 16KB, 32-way | 64KB, 2-way |
| L2 Cache | 256KB 4-way | 1MB DM |
| L1 lat. | 1 cycles | 2 cycles |
| L2 lat. | 8 cycles | 12 cycles |
| Int ALUs | 2+1 mult/div | 4+1 mult/div |
| FP ALUs | 1+1 mult/div | 2+1 mult/div |
| INT Issue Queue | 4 entries | 20 entries |
| FP Issue Queue | 4 entries | 15 entries |
| LS Queue | 8 | 64 |
| Reorder Buffer | 40 | 80 |

We use the same policy parameters and thresholds used by Zhu et al. [40]. The power management controller is triggered every 500K instructions.

### 5.5.2 Evaluation of online-IDVS policy

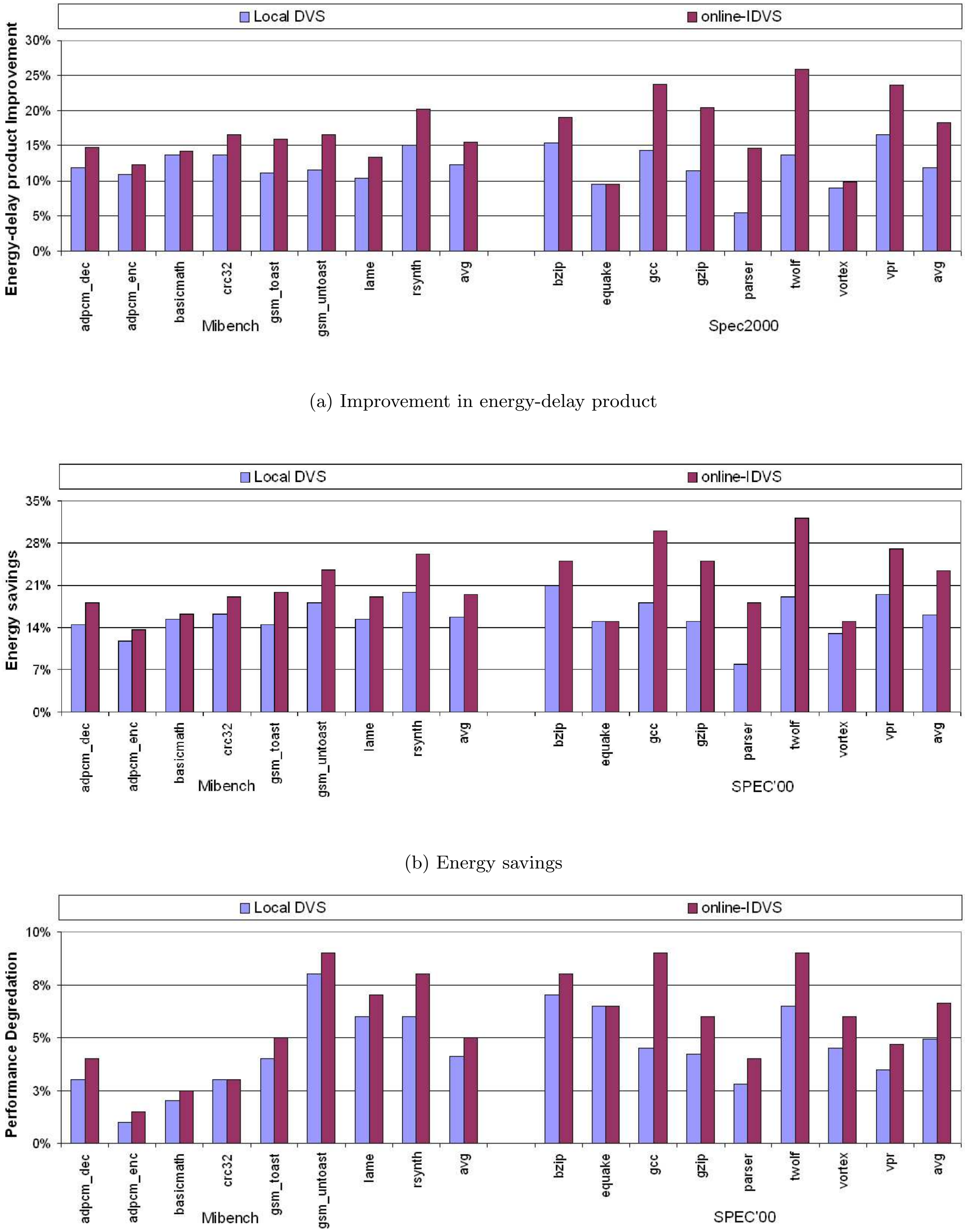
**5.5.2.1 Impact on performance and energy consumption** We first evaluate the policies using an embedded processor (Configuration A in Table 5.6). Figure 5.8-a shows the improvement in energy-delay product over no-power management (higher is better). For the MiBench applications, the energy-delay product improvement is 15.5% on average (up to 21% in *rsynth*) over no-DVS policy. For the SPEC2000 benchmarks, the improvement in the energy-delay product is 18% on average (up to 26% in *twolf*) over no-DVS policy. Most of the improvement is a result of energy savings (an average of 21% across applications) as seen in Figure 5.8-b, with much less performance degradation as seen in Figure 5.8-c.

The integrated, interaction-aware policy achieves an extra 7% improvement in energy-delay product above local-DVS gains. These savings are beyond what local-DVS can achieve over the no-DVS policy[2]. The improvement over local-DVS comes from avoiding the undesired positive feedback scenarios by using coordinated DVS control in the core and L2 cache domains. However, the energy-delay product improvement beyond the gain achieved by local-DVS is highly dependent on the frequency of occurrence of the positive feedback scenarios, the duration of the positive feedback and the change in speed during these positive feedback cases throughout the application execution. From Table 5.1, we notice that the frequency of the positive feedback in *adpcm, basicmath*, and *crc32* is almost negligible; accordingly, there are significantly smaller benefits from our policy as shown in Figure 5.8. On the other hand, applications like *gsm, rsynth, gcc, parser*, and *twolf* show high energy savings due to repeated occurrence of positive feedback cases.

With respect to performance, we note that our proposed integrated policy has a slowdown of 5% on average for MiBench (7% on average for SPEC2000). This slowdown is only 1% more than the slowdown of local-DVS.

To test whether a different policy that avoids the undesired positive feedback scenarios using alternative actions (specifically, different actions for rules 1 and 5 in Table 5.2) would perform better, we experimented with different rules for these two cases. Table 5.7 shows the actions of

[2] As also noted in [41], reported results of the local policy are not identical to the one reported in [35] due to few reasons. The latest distribution of the MCD simulation tool set has a different implementation of the speed change mechanism. We simulate two-domain processor versus five-domain processor in the original local-DVS policy. We execute applications with different simulation window, as well.



(a) Improvement in energy-delay product

(b) Energy savings

(c) Performance degradation

Figure 5.8: Energy and delay of Local-DVS and our online-IDVS relative to no-DVS policy in configuration A and two voltage domains processor.

Table 5.7: Variants of our proposed policy: actions of setting the core voltage ($V_c$) and the cache speed ($V_\$$) in rules 1 & 5 from Table 5.2.

| rule | P0 | | P1 | | P2 | | P3 | | P4 | | P5 | | P6 | | P7 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| # | $V_c$ | $V_\$$ | $V_c$ | $V_\$$ | $V_c$ | $V_\$$ | $V_c$ | $V_\$$ | $V_c$ | $V_\$$ | $V_c$ | $V_\$$ | $V_c$ | $V_\$$ | $V_c$ | $V_\$$ |
| 1 | ⇓ | ⇑ | ⇓ | – | ⇓ | – | ⇓ | – | – | ⇑ | – | – | – | ⇑ | – | ⇑ |
| 5 | – | ⇓ | – | ⇓ | – | – | ⇑ | – | – | ⇓ | ⇑ | – | ⇑ | – | – | – |

seven policy variants, in addition to our proposed integrated policy P0. Figure 5.9 shows the average degradation in energy-delay product relative to local-DVS. It is clear that other actions for dealing with positive feedback scenarios are not as effective in reducing the energy-delay product. The degradation in energy-delay product of the policy variants ranges from 2% to 12% over our proposed policy, P0.

**5.5.2.2 Varying system configurations** We study the benefit of using a domain interaction-aware DVS policy under different system configurations. We explore the state space by varying key processor configurations and the granularity of DVS control (that is, number of MCD domains). In addition to a simple embedded single-issue processor (configuration A in Table 5.6), we experiment with a more complex embedded processor (configuration B); the same processor configuration used in [40]. This test should identify the benefits of the interaction-aware policy in a simple embedded processor versus a more powerful one. This more powerful processor, such as Intel's Xeon 5140 and Pentium M, are used in portable medical, military and aerospace applications [92]. Figure 5.10-a compares configuration A versus configuration B in terms of energy-delay product, energy saving, and performance degradation. The figure shows the average values over the MiBench and SPEC2000 benchmarks for 2 domains. One observation is that we achieve larger energy-delay improvement in embedded single-issue processor (Config A) than the more complex one (Config B). This larger improvement is mainly due to higher energy savings. In single-issue processors, cache misses cause more CPU stalls (due to lower ILP) than in higher-issue processors. This is a good opportunity for the DVS policy to save energy by slowing down domains with low workloads while having small impact on performance.

Comparing our results with local-DVS, we notice that the average benefit of considering domain

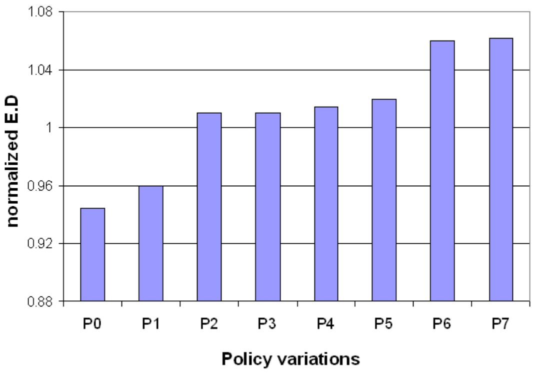
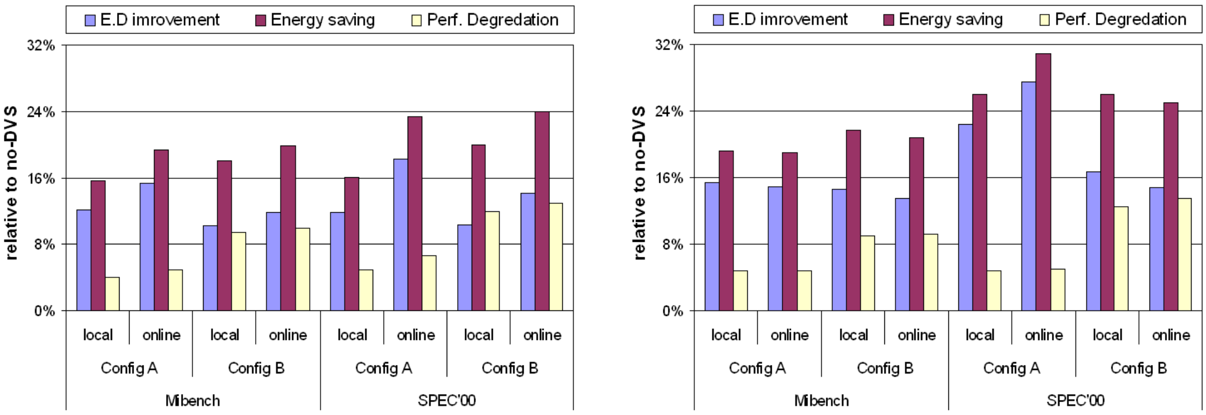


Figure 5.9: Average degradation in energy-delay product relative to the local-DVS policy

interactions decreases with the increase in issue width. This is because processors with small issue width are more exposed to stalls from the memory hierarchy, which makes it important to consider the core and L2 cache domain interaction. In contrast, with wider issue width, these effects are masked by the core's higher ILP.

Because we are also interested in the effect of interactions across multiple domains on energy savings, we test the effect of increasing the number of clock domains. To perform this test, we simulated the five domains used in [35], but added a separate domain for the L2 cache. The resultant domains are: reorder buffer domain, fetch unit, integer FUs, floating point FUs, load/store queue, and L2 cache domains. We use our policy to control the fetch unit and L2 domains, and set the speeds of the remaining domains using the local policy [35, 40]

Figure 5.10-b shows the results for the two processor configurations when dividing the chip into the six domains mentioned above. Comparing Figures 5.10-a and 5.10-b, we find that DVS in processors with large number of domains enables finer-grain power management, leading to larger energy-delay improvements. However, for embedded systems, a two-domain processor is a more appropriate design choice when compared to a processor with a larger number of domains (due to its simplicity). Figure 5.10 shows that increasing the number of domains had little (positive or negative) impact on the difference in energy-delay product between our policy and local-DVS. This indicates that the core-L2 cache interaction is most critical in terms of its effect on energy and delay, which yielded higher savings in the two-domain case. We can conclude that a small number of domains is the most appropriate for embedded processors, not only from a design perspective



(a) 2 domains

(b) 6 domains

Figure 5.10: Energy and delay for local-DVS and online-IDVS policy relative to no-DVS policy for processors with (a) two domains and (b) six domains.

but also for improving energy-delay.

### 5.5.3 Evaluation of ML-IDVS approach

Similar to the online-IDVS evaluation, we test the ML-IDVS approach under different settings: different application classes and different processor architectures. We first focus on the embedded domain that commonly use simple single-issue processors with two clock domains.

To reduce the state space for the profiling and table construction (described in Section 5.4.2), we use the discrete values for the collected performance counters. We discretize CPI values into 11 bins, L2PI into 8 bins, and MPI into 4 bins. The simulated frequencies for both domains vary from 250MHz to 1GHz with 250MHz steps. Voltage scales linearly with the frequency in the specified range. Memory is considered an external domain with a fixed latency.

To obtain the propositional rules, we use *JRip* from the WEKA data mining software package [93]. *JRip* is an optimized implementation of the RIPPER learner. The rules are produced based on the data collected for the given architectural configuration. Each rule specifies the desirable CPU frequency and cache frequency for the next program interval based on the current state (that is, CPI, L2PI, MPI, old CPU and cache frequencies).

An important aspect of using JRip is the format of the training data, which affects the quality of the generated rule set. Although all the state parameters of the training data are discrete (cache and CPU frequencies are discrete in nature, while CPI, L2PI, and MPI are discretized into bins), we specify in the input to JRip that all parameters are continuous to get a more compact rule set. Using JRip also involves tuning the parameters for the RIPPER algorithm. For instance, the RIPPER algorithm needs to partition the training data into a growing set and a pruning set. We choose the partition size to be two thirds for the growing set. Since RIPPER is a randomized algorithm, different randomization seeds will lead to different results. We experimented with different values and chose a seed value that reduced the error rate and rule set size for our input.

**5.5.3.1 Impact on performance and energy consumption** In this section, we show the energy-delay product results of the policies learned from ML-IDVS in comparison with a local CPU-core and L2 cache DVS policy [1]. We show how ML-IDVS is affected by the class of applications, architectural configurations, and optimization metric (as shown in Figure 5.4.1).

**Energy-delay product improvement:** Figure 5.11 shows the energy-delay product for local-DVS versus the ones generated by ML-IDVS. The generated policy (with the use of MPI) contains 33 rules. On average, ML-IDVS's policy improves energy-delay product over local-DVS by 21% and 22% for the MiBench and SPEC2000, respectively. The local policy being a heuristic-based can perform badly with some applications as seen in *crc32*, *dijkstra* and *mesa*. This is because the policy was unable to select the best frequency to minimize the energy-delay. It rather reacts to the CPI and L2PI changes by changing the frequencies in the same direction of their change, which does not guarantee operating on an efficient frequency.

**State description:** Figure 5.11 also shows the energy-delay product in case of removing the MPI feature from ML-IDVS state description (that is, only using CPI and L2PI). We test this case to show the significance of observing the memory activity and its impact of considering it in selecting the DVS decisions, even though we do change its speed. In case of not accounting for MPI, ML-IDVS generates a policy with fewer rules: 27 rules. In MiBench, using MPI does not improve the energy-delay product because most of the applications' data accesses occur in the caches. So, memory latency has trivial effect on the application's performance and energy. In contrast, SPEC2000 benchmarks have larger memory footprints, thus using MPI in the state

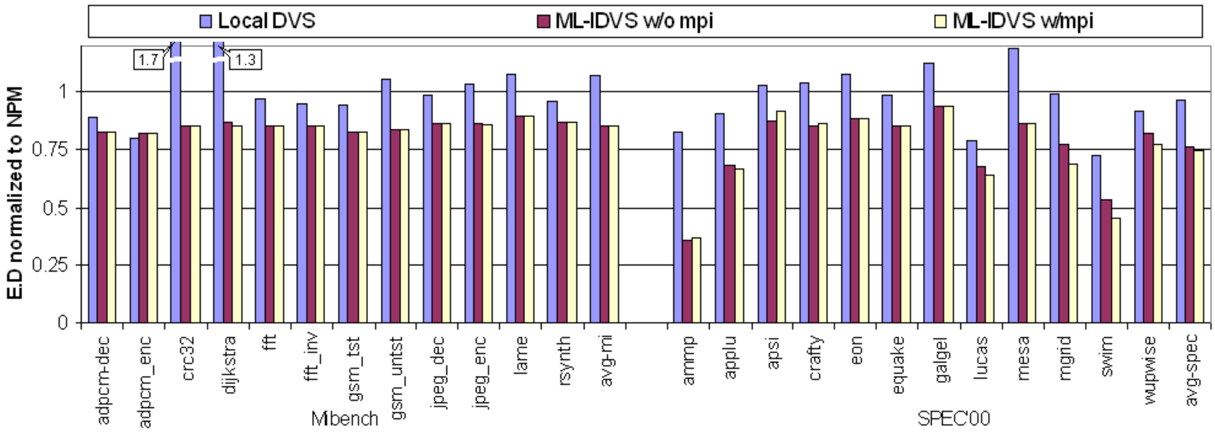


Figure 5.11: Energy-delay product for SPEC2000 and MiBench benchmarks when using local-DVS versus ML-IDVS.

description enables the rule learner to distinguish between memory bound versus L2 cache bound phases. Hence, ML-IDVS with MPI further improves energy-delay product by 1.8% on average and up to 8.6% (in *mgrid*).

**Optimization metric:** ML-IDVS analyzes the application and the architectural behavior once, and then it can generate policies geared toward optimizing a given metric. To show this, we use the same training samples obtained for energy-delay product and construct $ST$ to select the best frequency combination that minimizes the energy while maintaining the delay within a given bound. We show results for a new policy generated with 10% bound on performance degradation objective. Figure 5.12 shows the energy-delay product of our benchmarks (when accounting for MPI). On average, we achieve 21% and 14% improvement over local-DVS for the MiBench and SPEC2000, respectively.

**Processor configuration:** ML-IDVS can be used with different architectural configurations. To show the impact of using ML-IDVS with wide range of processor configurations, we experiment with a high-performance processor configuration. For this experiment, we use an Alpha-like configuration shown in Table 5.6 (Config B). Figure 5.13 shows an improvement of 22% and 31% for the MiBench and SPEC2000 benchmarks, respectively.

**Control Granularity:** One important parameter of a DVS policy is how often to trigger a speed

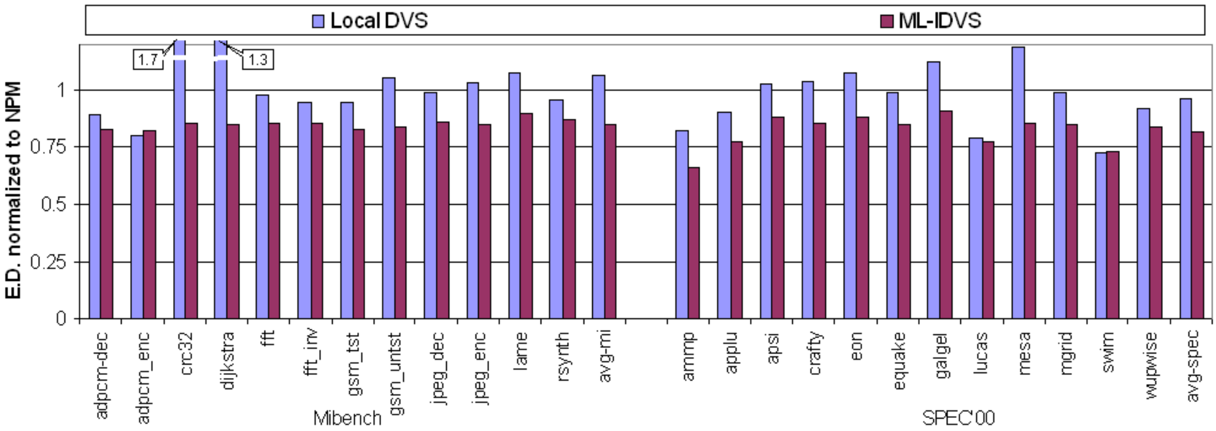


Figure 5.12: Energy-delay product when optimizing energy with delay bound.

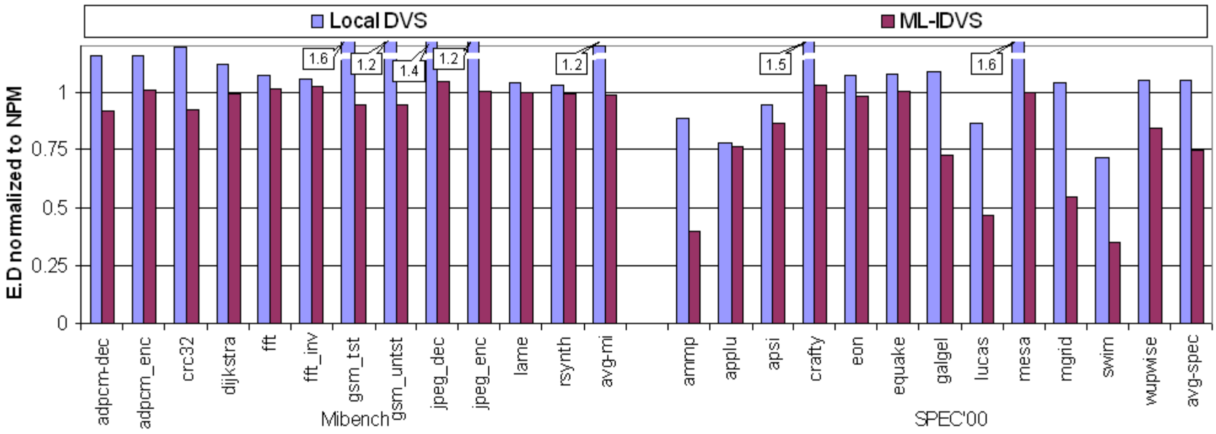


Figure 5.13: Energy-delay product for policies running on system with configuration Config B in Table 5.6.

change. Few speed changes reduce overhead but also eliminate the fine grain control to adapt to shorter program phases, and vice versa. In this experiment, we vary the period of triggering the DVS policy. We report the average energy-delay product– normalized to no-power management– of all reported MiBench and SPEC2000 benchmarks. Figure 5.14 shows that by increasing the control interval size, local-DVS reduces the number of speed changes, which reduces the policy overhead and thus reduces its energy-delay product. On the other hand, our DVS policy naturally has fewer speed changes because it selects the best frequencies for a given state rather than changing the frequency based on reaction to a change in feature value as in local-DVS. Hence, increasing the control interval size has minimal impact on the energy-delay product in the tested range.

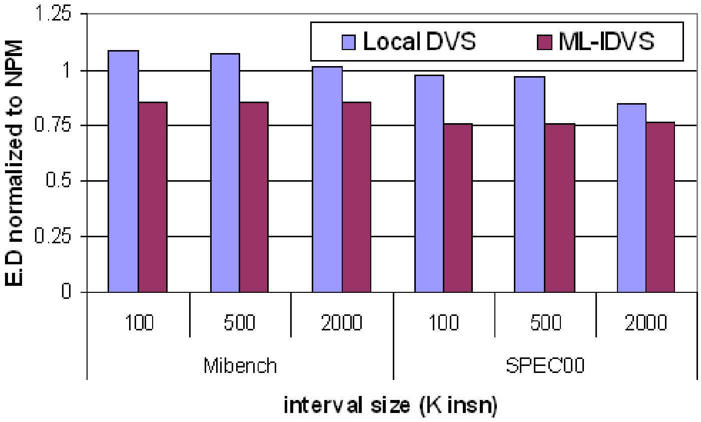


Figure 5.14: Average energy-delay product at different DVS control-interval sizes (using Config A).

From the results in this section, we conclude that our learning methodology is capable of generating policies that can used to optimize different systems. The policies being aware of the system state are effective in optimizing the system (for example, by reducing the energy-delay product or energy with limited performance degradation).

**5.5.3.2 Analysis of the training process** We further investigate the effectiveness of the training data in discovering the possible states in $ST$, which are used in generating the policy rules (as described in Section 5.3). The objective is to discover most of $ST$ from the training sample. Each of the applications used in training covers a number of states in the table with some applications having larger coverage than others. Figure 5.15 shows the number of distinct states that each application can discover. The applications are sorted in descending order by number of states. The

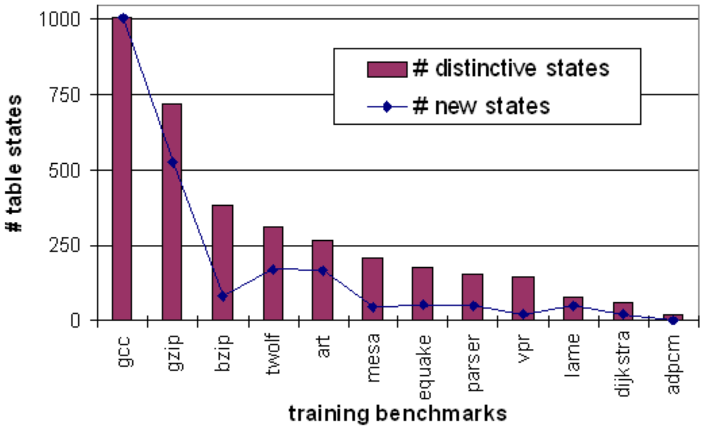


Figure 5.15: *ST* coverage.

line graph in the figure represents the number of new states contributed to *ST* by each application. Intuitively, using applications with a higher number of discovered states is more beneficial in the training process as they add more information to *ST*. For example, the first four applications (*gcc*, *gzip*, *bzip* and *twolf*) were responsible for 82% of the *ST* coverage. However, using applications with large number of distinctive states but populating states that were already discovered in *ST* is not useful. For example, *bzip*, exhibits similar behavior to *gzip*, thus, few new states were populated in *ST* by *bzip*. Conversely, *art* is very useful to populate an area not covered by the other applications. Hence, a desired characteristic for applications to use in training is to exhibit large variations in program behavior (phases) that are different than other training applications used. By carefully choosing a few applications with varying behavior, *ST* can be covered with relatively small number of training applications.

### 5.5.4 Online-IDVS versus ML-IDVS

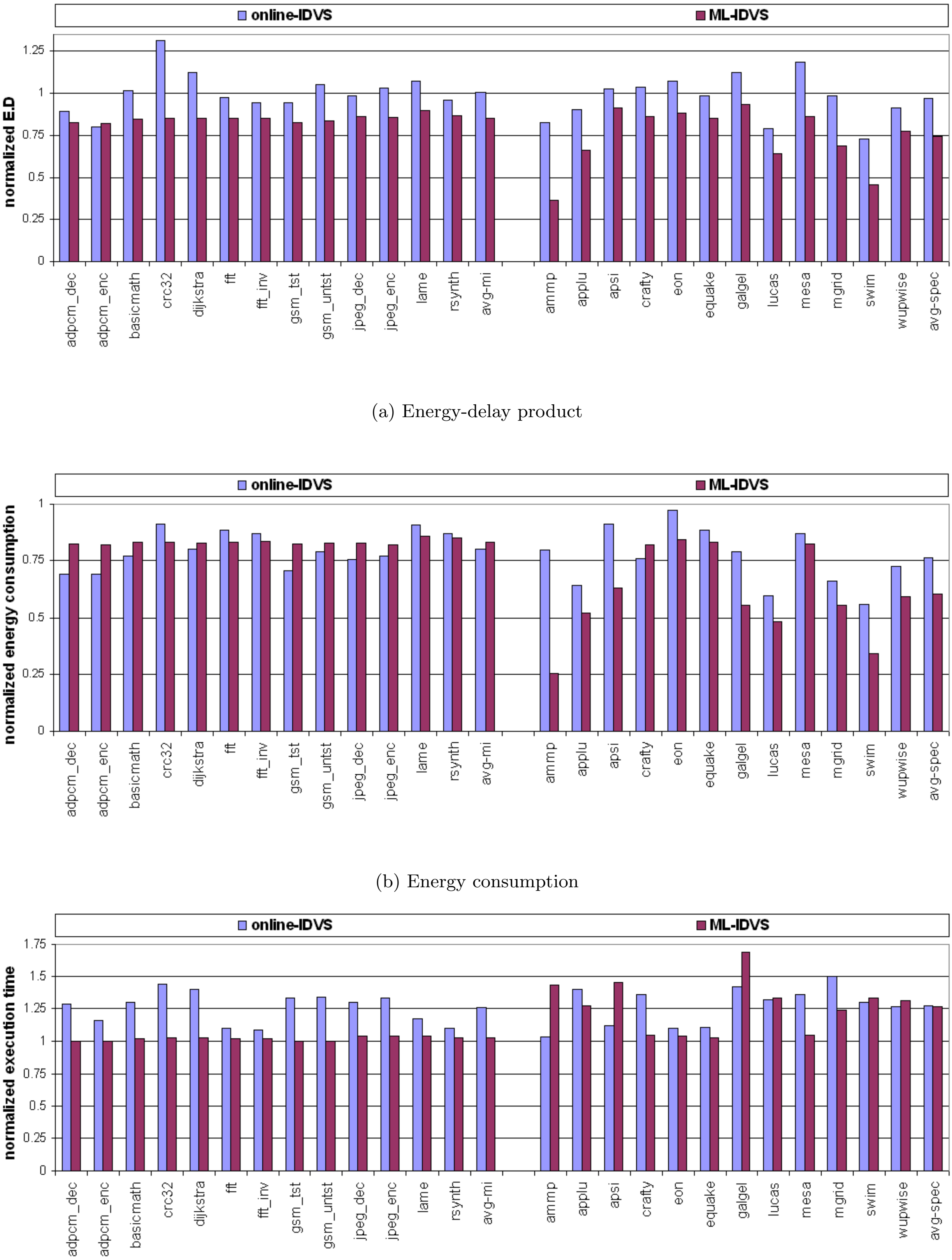
**5.5.4.1 Energy and delay savings** Figure 5.16 compares online-IDVS and ML-IDVS. Figure 5.16-a shows that savings in energy-delay product of online-IDVS diminishes compared to results in Section 5.5.2. This is primarily because of using limited number of discrete speed levels. At each interval, online-IDVS computes the magnitude of speed change which is proportional to the change in CPI or L2PI. Actual speed change does not occur unless the difference in speeds (computed *minus* current) is high/low enough to select the next available speed level. Accordingly, the number of speed changes in online-IDVS is much fewer then in ML-IDVS policy.

Figure 5.16-b shows that both IDVS techniques achieve energy savings for all tested applications at the expense of longer execution times as shown in Figure 5.16-c. Some applications experience higher than desired performance degradation ($\geq$50%) as in *galgel* and *mgrid*. In ML-IDVS, this delay can be bounded by choosing different optimization metric, such as delay-bound, when constructing the state table. In online-IDVS, delay penalty can be reduced by reducing the aggressiveness of magnitude of the speed change. In general, it is hard to provide guarantees on the delay since both techniques predict the future behavior of the application to be similar to the past. Table 5.8 summarizes the main differences between online-IDVS and ML-IDVS.

#### 5.5.4.2 Discussion

**Training overhead.** The major difference between online-IDVS and ML-IDVS is the training phase. Training gives ML-IDVS some knowledge of which operating frequency results in minimum energy consumption when executing application with certain behavior. However, this requires an extra phase for every class of applications. Training time is function of the number of applications used in training, $p$, the number of frequency levels, $f$, and the number of clock domains, $d$. The time complexity for training is $O(p \cdot f^d)$ for each class of applications. The exponential function makes the training and rule generation phases time consuming in case of dealing with large number of frequency levels and large number of domains. In such case, online-IDVS is a less expensive solution, but also with lower energy savings.

**Speed change overheads.** The overhead of a frequency change is typically just a few microseconds. However, voltage change overheads are higher, ranging from a few dozen microseconds to a few milliseconds (e.g., for StrongARM SA-1100 the voltage change overhead is $140\mu s$ [94]). The sample *size* should be selected based on the total overhead: a small sample size may have high frequency/voltage change overheads, while a large sample size may exceed typical code phases of applications. For example, changing the frequency/voltage every $1ms$ with an overhead of $140\mu s$ yields an overhead as large as 14%. The overhead can be mitigated by enforcing a limit on the number of speed changes. A simple scheme would be to enforce at most one speed change say every $10ms$, without changing the sample size. Alternatively, the sample size can be increased.



(a) Energy-delay product

(b) Energy consumption

(c) Execution time

Figure 5.16: Online-IDVS versus ML-IDVS normalized to no-DVS

Table 5.8: Comparison of online-IDVS and ML-IDVS

| | ML-IDVS | Online-IDVS |
|---|---|---|
| E.D improvement | Higher - due to offline profiling | Less efficient |
| High order of domains | (offline) Time-consuming rule generation | Less intuitive domain interactions |
| Diverse set of applications in the system | higher training overhead | Same policy |
| Optimizations | Work with multiple objectives | Not studied |

**Inefficient operating points.** Processors may have inefficient frequency/voltage combinations [95]. A frequency is inefficient if there exist a higher frequency that results in lower energy consumption. One advantage of ML-IDVS is that the power manager can determine the best action and inefficient operating points are naturally eliminated if they exist. This is not true in case of online-IDVS since it is oblivious to the available operating frequencies.

**Measurement-based versus theoretical models.** We use a measurement-based approach (i.e., experiments are run to derive a policy), as opposed to an analytic model-based approach. Thus, there is no implicit assumption of theoretical power models (such as power relationship with the voltage and frequency). This means that the policy works well in identifying the correct actions without assumptions about whether the system supports DVS or just frequency scaling (for example) and without assumption about the relationship among voltage, frequency and power. While a model can be inaccurate and difficult to construct, measurement-based approaches eliminate this problem from the start, at the expense of one-time offline measurements.

**Sample size unit.** We chose DVS control intervals measured in number of instructions instead of number of cycles (or time) to evaluate different actions for the same code sample. The table describing the derived policy can actually be used with a periodic timer-based mechanism. Different sample $size$ values result in the same policy rules, as long as $size$ is not larger than the application phases.

## 5.6 CONCLUSION

In MCD processors, applying DVS in each domain can significantly reduce energy consumption. However, varying the voltage and clock speed independently in each domain indirectly affects the workload (and hence power) in other domains. This results in an inefficient DVS policy. In this chapter, we identified these inefficiencies and proposed two integrated DVS techniques that account for inter-domain interactions. Our policies separately assign the voltage and clock of the core and L2 cache domains based on activity in **both** domains.

We present a heuristic-based online-IDVS policy and a machine learning approach to generate IDVS policies tailored toward a given class of applications. Our results show that both policies achieve higher energy and energy-delay savings than a local MCD DVS policy that is oblivious to domain interactions.

The online-IDVS uses a heuristic for controlling the clock of each domain based on the change in the monitored performance counters. The policy consists of few simple rules that are independent of the underlying architecture. Online-IDVS achieves average savings in energy-delay product of 7% for SPEC2000 and 3.5% for MiBench benchmarks over past local DVS approaches using the same hardware.

The ML-IDVS approach characterizes the system state by the running applications and collecting their behavior on a given architectural configuration. ML-IDVS approach constructs policies to optimize the system power according to a user selected optimization metric. The ML-IDVS approach generates efficient policies that save 22% on average (up to 46%) in energy-delay product over a DVS technique that apply local DVS decisions in each domain.

Results show that ML-IDVS is better than online-IDVS. This is primarily due to the extra information acquired by ML-IDVS during the training phase that helps in predicting the application behavior. ML-IDVS requires training for each different class of applications to generate accurate predictions for wide range of applications. Thus, ML-IDVS is best suited for systems running applications belonging to a few set of classes. Since online-IDVS does not assume any prior knowledge about applications in the system, it better suits systems running a diverse set of applications in which offline learning of all application classes is too expensive.

# 6.0 POWER-AWARE CACHED DRAM

With the continuing advancement in the design and manufacturing of faster and more powerful computing systems, more performance is demanded from the memory system. For example, in current chip multiprocessing (CMP) and simultaneous multithreading (SMT) processors, concurrent applications or threads allow the CPU(s) to issue more load and store requests in each cycle. Even with large caches, this higher demand, coupled with a slow memory access time, creates a potential performance bottleneck.

On the other hand, Memory has a huge internal bandwidth compared to its external bus bandwidth. Internal memory bandwidth can reach 1.1 TB/s while fast external memory bus is in the range of 10 GB/s [96]. To exploit the wide internal bus, *cached DRAM* (CDRAM) adds an SRAM cache to the DRAM array on the memory chip [97]. Such a *near-memory cache* acts as an extra memory hierarchy level, whose fast latency improves the average memory access time and potentially improves system performance, provided that the near-memory cache is appropriately configured. Moreover, in case of a cache miss, transferring a cache block over an external bus consumes four times more energy than transferring the same data over the internal bus (that is, it does not cross a chip boundary). This reduction is due to the smaller capacitance of internal buses (0.5pf) compared to external buses (20pf) [98].

In this chapter, we show how to reduce the combined energy consumption in DRAM memory and off-chip caches by placing SRAM caches closer to the memory, rather than closer to the CPU. We integrate a moderately sized cache within the chip boundary of a power-aware multi-banked memory. We call this organization *power-aware cached DRAM (PA-CDRAM)* [99]. In addition to improving performance, PA-CDRAM significantly reduces energy consumption in caches and in main memory. Cache energy is reduced because (1) using small caches distributed to the memory chips reduces the cache access energy compared to using a large non-distributed cache, and (2) near-memory caches allow the access of relatively large blocks from memory, which is not affordable with *near-processor* caches. Memory energy consumption is reduced by having longer memory idle period during which DRAM banks can be powered off. PA-CDRAM improves the original CDRAM by tackling the interplay of the cache and memory organizations to optimize the memory's performance and energy consumption.

The innovation of this work is the use of near memory caches to further reduce the memory's energy consumption beyond the savings achieved from traditional DRAM dynamic power management. CDRAM was proposed to improve performance; however, it may hurt energy consumption (as demonstrated in Section 6.2). Combining CDRAM and DRAM power management would potentially benefit from the performance gains and save energy.

The contribution of this chapter is threefold. First, we propose near-memory caches for energy reduction in the memory hierarchy. This reduction comes without affecting the performance gain provided by CDRAM. Second, we describe an implementation of PA-CDRAM that integrates a near-memory cache in a Rambus chip (RDRAM). This description includes the changes made to a RDRAM chip, the near-memory cache controller, and the communication protocol. We also describe how our implementation can maintain backward compatibility with existing Rambus memories. Finally, the chapter experimentally evaluates PA-CDRAM and shows that PA-CDRAM is more energy efficient than a traditional Rambus memory hierarchy employing a time-out power management policy.

## 6.1 CACHED DRAM

To decrease the average memory access time, Hsu et al. [97] proposed to integrate a small SRAM cache within the memory chip next to the DRAM-core, as shown in Figure 6.1. Due to high internal bandwidth, large chunks of data can be transferred between the DRAM-core and the near-memory cache with low latency. Average memory access time is improved by accessing the data through the fast near-memory cache rather than the slower DRAM. CDRAM chips were first manufactured by Mitsubishi [7], and are typically implemented using Synchronous DRAM (*SDRAM*). Each memory bank has its own cache.

Hsu et al. evaluated the performance of CDRAM in vector supercomputers. They showed an improvement in CPI compared to traditional memory without this extra cache. To improve the CDRAM performance, Koganti et al. [100] proposed a cached-DRAM with wide cache line ranging

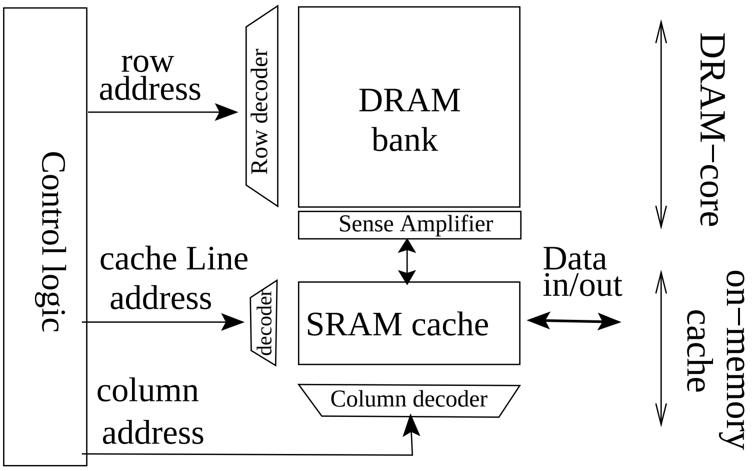


Figure 6.1: Functional block diagram of a CDRAM

from 4KB to 8KB interleaved across multiple DRAM banks. Although the SRAM cache's miss rate is reduced, the authors did not account for the extra delay and energy spent accessing these large cache block sizes, which may degrade the performance and increase the memory's energy consumption. Hegde et al. [101] proposed using variable width cache lines that fit the application access pattern to save energy consumed in unnecessary traffic between the DRAM-core and the near-memory cache. Zhang et al. [102] showed that cached DRAM is able to improve performance as the ILP degree increases. Past work did not explore the energy savings that can be achieved over a currently available alternative such as power-aware memory with the same overall system cache and memory capacity.

## 6.2 PA-CDRAM

CDRAM was originally proposed to improve system performance; however, it was not designed as a replacement to power-aware memory. Figure 6.2 shows the average performance and energy consumption of CDRAM versus a traditional memory hierarchy for the same near-memory cache configuration used by Hedge et al. [101].[1] Each CDRAM chip has a fully associative 4 KB cache. We show the results for different cache block sizes (256, 512 and 1024 bytes) for the embedded

[1] Data produced when running the SPEC2000 benchmarks on SimpleScalar.

SRAM cache. While CDRAM has a good performance improvement over traditional memory, the memory's total energy suffers dramatically, with an increase of 1.5x to 3.0x. This increase is due to the extra energy consumed from accessing the near-memory caches and transferring more data from the DRAM-core at large block sizes.

We propose to overcome the energy penalty of CDRAM by decreasing the miss rate of the near-memory cache and using power management for the DRAM-core. As we will show, making the CDRAM power-aware not only decreases the energy penalty of CDRAM, but also significantly improves *overall* memory energy in comparison to a traditional power-aware memory hierarchy.

Because DRAM power management typically relies on *idleness* to select power states, an improved miss rate in the near-memory cache increases the amount of idleness in the DRAM-core, leading to more effective power management. One way to improve miss rate is to increase the capacity of the near-memory cache. However, the total energy of the whole memory hierarchy is increased due to greater overall cache capacity. Instead, we propose to re-allocate existing cache capacity from the memory hierarchy's lowest cache level to near-memory cache. For example, it may be possible to allocate the capacity of the L3 cache to CDRAM's near-memory cache. The L3 cache could then be eliminated, possibly without harming application performance. Moving cache capacity to the near-memory cache has three advantages. First, the near-memory caches are distributed among the memory chips, which lead to lower energy consumption because the individual caches are smaller than one monolithic cache. Second, large data transfers are possible from the DRAM-core to the near-memory cache (i.e., there is more memory bandwidth, which makes large block sizes feasible). Finally, the near-memory caches can filter and avoid accesses to the DRAM-core. Since near-memory caches can achieve lower miss rates than near-processor caches, such filtering increases idleness and lets the DRAM-core stay in a low power state for longer periods of time.

To build a power-aware cached DRAM, there are two main challenges that must be addressed: (1) how to configure the DRAM-core's power management, assuming the use of multiple power states? and (2) what is the best configuration for the near-memory cache to balance energy and performance? We describe each of these challenges and our way of addressing them below.

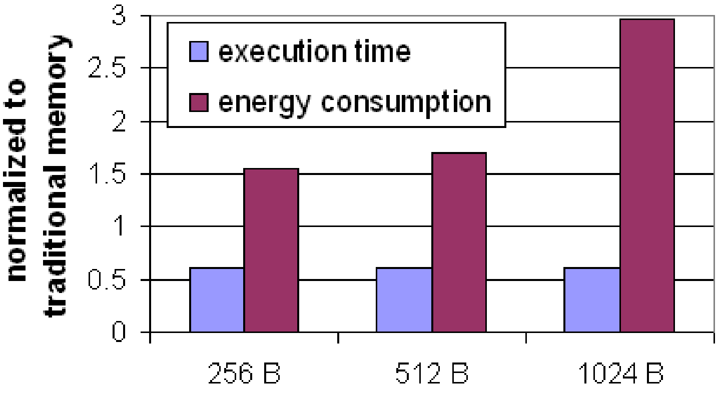


Figure 6.2: Average performance and energy consumption for different near-memory cache block sizes.

### 6.2.1 DRAM-core power management

With a near-memory cache, we propose applying aggressive power management in the DRAM-core. During a chip's idle time, the memory controller can immediately transition the DRAM-core to the idle state after servicing all outstanding requests. This is equivalent to the use of a timeout policy with an idle threshold of zero seconds. Although a zero-threshold policy increases the total inactive time, it can degrade performance and increase the total energy consumption when too many requests are directed to a memory chip. The extra delay and energy overheads are due to the transitional cost between power states.

In PA-CDRAM, we avoid this problem by choosing the near-memory cache configuration in a way that increases the hit rate while reducing the DRAM-core's energy consumption.[2] When most data requests are serviced as cache hits in the near-memory cache, the longer inter-arrival time between requests that reach the DRAM-core make it cost effective to immediately deactivate banks after servicing outstanding requests. We choose to keep the near-memory cache active all the time to avoid delays that may be caused by on-demand activation of the cache at each request.

[2] Different configurations for the L3 cache would not have the same effect: near-memory caches have a much wider bandwidth to memory than L3 caches.

### 6.2.2 DRAM-core versus near-memory cache energy trade-off

To reduce the memory's energy consumption, we need to consider the effect of the near-memory cache configuration on the energy consumption of both the near-memory cache and the DRAM-core. The two factors that affect the cache energy consumption and access latency —for a given cache size and fixed number of cache subbanks —are the *associativity* and the *block size* [66].

The cache associativity directly affects miss rate. Increasing associativity reduces cache miss rate and vice versa. One goal of PA-CDRAM is to keep the near-memory cache miss rate as low as possible because it directly influences memory energy consumption in two ways. First, the higher the miss rate, the more activity in the DRAM in terms of transitioning from idle to active state, performing address decoding, and transfer of data. Second, the lower the miss rate, the longer the DRAM-core idle time. To keep the miss rate at a minimum, we use fully associative caches to eliminate any conflict misses. Note that the hardware complexity of fully associative caches and their effect on access latency and power is not a limiting factor since we use relatively small caches with small number of blocks as shown in Section 6.5. We demonstrate that in most cases, performance improvement and energy saving from reducing near-memory miss rates (given the appropriate cache configuration) can outweigh extra delay and energy consumed in accessing higher associative caches [103].

Reducing the miss rate only is not sufficient to reduce the memory's energy consumption and application's overall delay. Thus, in PA-CDRAM, we account for the effect of the cache configuration on the overall memory's energy (in contrast to Koganti et al. [100]). After selecting the cache associativity, choosing a near-memory cache block size creates a trade-off between the near-memory cache and DRAM-core energy consumption. Small cache blocks have the advantage of fast hit time and low energy per access. However, smaller blocks imply frequent accesses and consequently increasing the DRAM-core energy due to the increased activity. The increase in the DRAM-core energy rises as a result of increasing the memory activity (such as power-state transitions, address decoding, and data transfer). Conversely, larger near-memory cache block sizes reduce the DRAM-activity but increase the near-memory cache energy consumption and latency due to accessing these large blocks.

This trade-off is illustrated in Figure 6.3. The figure shows the energy consumption in near-memory cache and DRAM-core at different cache block sizes. Energy values are obtained using a simplified energy model. The model estimates the near-memory cache energy consumption,

$E_{cache}$, and DRAM-core, $E_{Dcore}$ as a function of the number of near-memory cache and DRAM-core accesses. Per-access energy of near-memory cache is obtained from Cacti 3.0 [66] for a 256KB fully associative cache. We use Rambus 32MB memory chip specifications to compute the per-access energy in DRAM [46]. From this model, given the number of cache accesses and the approximate execution time for an application, we can roughly estimate the memory energy consumption at different block sizes. We use SimpleScalar [75] to estimate the input parameters (number of cache accesses and execution time).

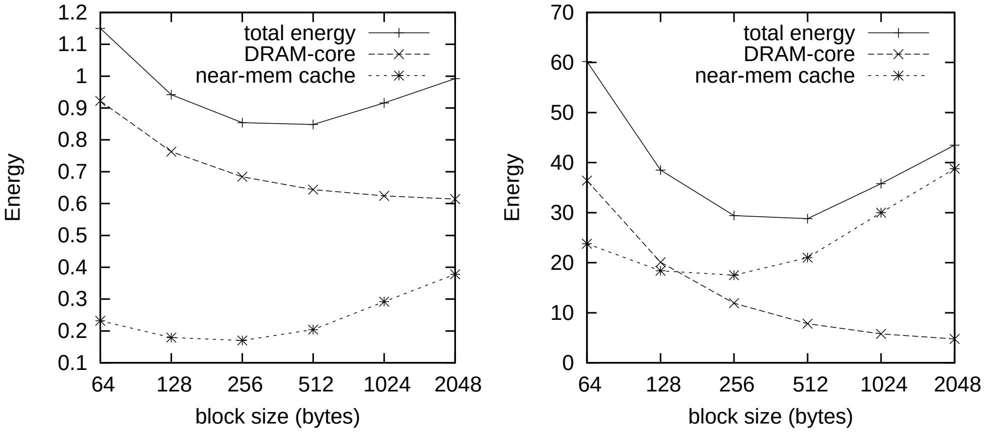


Figure 6.3: The effect of different cache block sizes on the memory energy consumption for the CPU-intensive *bzip* (left) and the memory-intensive *mcf* (right).

In Figure 6.3, we show estimated energy for two of the SPEC2000 benchmark: *bzip* and *mcf* as an example of CPU and memory intensive applications, respectively. In *bzip*, the DRAM-core idle energy dominates the PA-CDRAM energy, while in *mcf*, the frequent accesses to the near-memory caches makes the cache energy dominate the total energy at large block sizes. From the figure, we see that the trade-off between the near-memory cache and DRAM-core energy consumption creates a sweet spot between block sizes 256 and 512 bytes. Note that finding ideal block size is application dependent. However, from our simulation (shown in Section 6.5), in most of the applications, the minimum energy-delay product can be achieved at, or within a slight margin of, one of these two block sizes [104].

From this section, we conclude that for the given cache size, the near-memory cache should be fully associative (to reduce the miss rate) and have a block size of either 256 or 512 bytes (to balance the memory's energy and delay). For the DRAM-core, setting the chip to the idle state after servicing outstanding requests (that is, timeout = 0) is expected to save the memory energy consumption. The implementation described in the next section uses such configuration.

## 6.3 PA-CDRAM IMPLEMENTATION

In this section, we describe the architectural and operational modifications needed to integrate a near-memory cache in RDRAM. First, we describe the organizational issues of integrating a near-memory cache in the memory chip. Next, we discuss the design of the controller for the near-memory cache because it has energy, performance and backward compatibility implications. Finally, we describe the operation of the controller, including some changes needed to Rambus' bus protocol.

### 6.3.1 PA-CDRAM architecture

Our PA-CDRAM design modifies the original RDRAM design for power efficiency. Beside the addition of the near-memory cache, some alterations are needed in the main components of the RDRAM, namely: the DRAM-core, the control logic, and the memory and cache controllers (in Section 6.3.2).

We add a fully associative cache (depicted as dark blocks in Figure 6.4) with its data divided array into two sections. Since the original RDRAM design has a divided data bus, each section of the cache is connected to one of two internal data buses (DQA and DQB). Each cache section stores half of each cache block. We keep the original RDRAM write buffers before the sense amplifiers. The write buffers are used to store replaced dirty blocks from the near-memory cache to be written to the DRAM-core. The power state of this cache is independent of the power state of the other chip components. In the nap state, the RDRAM internal clock is periodically synchronized with the external clock [46]. Thus, the near-memory cache is accessible even when the DRAM is in the nap state.

To accommodate the large transfer sizes between the DRAM-core and the near-memory cache, wider internal data buses are required to connect the sense amplifiers with the near-memory cache. The width of each bus connecting the DRAM and near-memory caches is $b/2$ bits, where $b$ is the size of a cache block. The buses DQA and DQB remain at an 8-bit width.

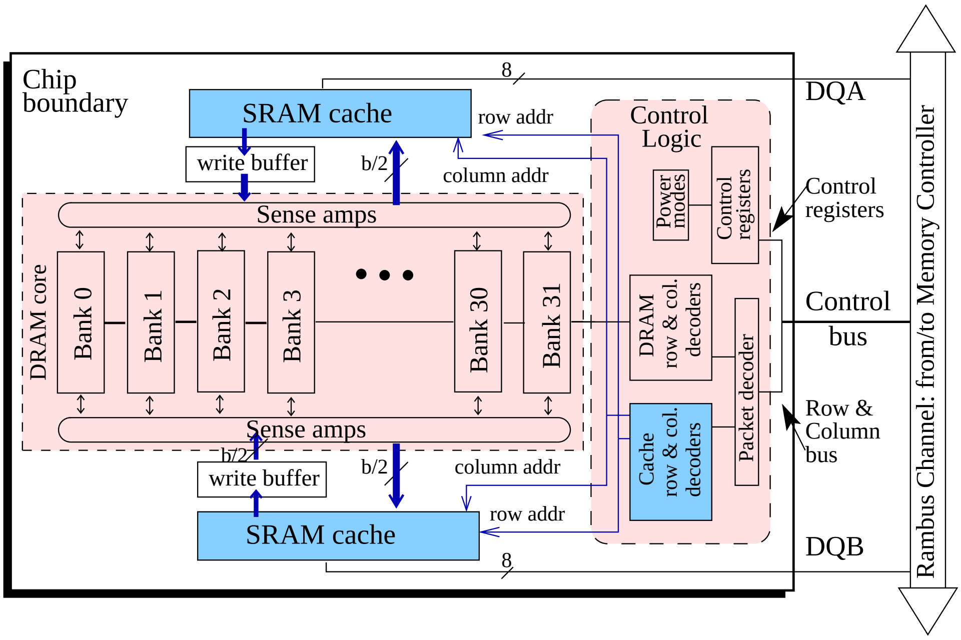


Figure 6.4: Functional block diagram of a PA-CDRAM. Dark blocks are the added components to an RDRAM architecture.

Extra cache row and column decoders are added in the chip's control logic for decoding cache addresses. In addition, the existing RDRAM packet decoder in each chip is modified to decode new cache commands. The new cache commands indicate whether the access is a hit/miss in the cache, and whether replacement is needed. The new commands and their fields are defined in Section 6.3.3.

### 6.3.2 Near-memory cache controller

In the design of PA-CDRAM, a controller is needed to handle hits and misses in the near-memory cache. An important design question is where this controller should be placed. One alternative is to have a single near-memory cache controller (called a *centralized controller*) that is integrated into the main Rambus memory controller. Another alternative is to have a near-memory cache controller *per* Rambus chip (called a *distributed controller*), with that controller integrated in the Rambus chip itself.

In the centralized controller case, there is a tag array for each memory chip. The centralized controller keeps track of the control data for each block such as a valid bit, dirty bit and LRU counters. This way, the cache controller is able to locally decide on replacement policies and update the tag arrays. The centralized controller communicates with cache data arrays by sending cache commands through the channel (as described in Section 6.3.3). In the distributed controller case, the cache controller is integrated more tightly with the near-memory cache (they are on the same chip). This design provides backward compatibility with current Rambus memory controller chips, as well as flexibility of connecting PA-CDRAM and RDRAM chips in the same channel. In the distributed design, each controller has its own tag array and the same functionality as the centralized controller. The distributed design has to intercept the control packets received from the memory controller through the channel and issue consequent command sequences to either the near-memory cache or the DRAM core.

The centralized controller has the advantage of performing a relatively faster tag comparison because it is implemented in the same technology as the memory controller. Thus, the centralized controller has a faster average memory access time compared to the distributed design, where the controller runs at a slower speed (due to the mixed use of logic and DRAM). For example, a delay penalty of 25% for logic cells with the distributed design has a 25% slower cache hit time versus roughly 16% penalty with a centralized controller design.[3] On the other hand, the bus activity (and hence the energy) in the Rambus channel is reduced by using a distributed design since only memory control packets are sent across the Rambus channel. Where as, in the centralized design, cache and memory control packets have to be sent across the channel. Hence, the distributed design uses less bus energy but suffers a performance penalty relative to the centralized design. In our evaluation (in Section 6.5), we examine the trade-off between these two alternatives. Next, we describe PA-CDRAM's operation.

### 6.3.3 PA-CDRAM operation

In the PA-CDRAM distributed cache controller design, after receiving the data request on the Rambus channel, the cache controller performs the tag-comparison and reads data from the near-memory cache if the tags match. Otherwise, the cache controller internally sends a sequence of control signals to activate a DRAM bank, read data and precharge the data line according to the DRAM-core timing constraints.

[3] According to Cacti, tag comparison takes around 35% of the total cache hit time for the selected configuration. That is: $x + (1 - 0.35)x \cdot 0.25 = 1.16x$.

For PA-CDRAM with centralized cache controller, we need to extend the Rambus communication protocol between the memory/cache controllers and the memory chips. This extension is needed because the cache controller resides within the memory controller chip; thus, it needs to drive the caches (invalidate lines, evict lines, etc) in the PA-CDRAM chips through the Rambus channel. For that, we define three new commands to communicate with the near-memory caches as shown in Table 6.1.

Table 6.1: PA-CDRAM cache commands send across the control bus

| command | description |
|---|---|
| cache_read_addr<br>*(CRA)* | Issues a read request for block_addr; data is sent on the data bus only if there is a *hit* in the tag comparison. A command packet is sent through the row bus.<br>Format: opcode(4b), chip_id(4b), block_addr(15b). |
| cache_write_addr<br>*(CWA)* | Issues a write request for block_addr (data is sent later on the data bus). A command packet is sent through the row bus.<br>Format: opcode(4b), chip_id(4b), block_addr(15b). |
| cache_tag_test<br>*(CTT)* | Sends the tag comparison result (tag_flag can be *hit* or *miss*). If *hit*, subsequent fields are ignored. If *miss*, it indicates, in subsequent fields, the address of block to be replaced and whether it is dirty or not. Command packet is sent through the column bus.<br>Format: opcode(4b), chip_id(4b), block_addr(15b), tag_flag(1b), dirty_flag(1b), replaced_addr(15b) |

To perform a data read/write request from the memory, a sequence of commands is issued along the Rambus channel. A read (write) can result in either a near-memory cache hit or miss. Figure 6.5 illustrates the sequence of commands and data on the row, column and data buses (timing diagrams) in the channel for a read hit and read miss. The communication protocol for command packets for a write hit/miss is similar to the read hit/miss, respectively, with the substitution of CRA by CWA command and the direction of data. The figure also shows the timing constraints imposed by the Rambus protocol on the earliest time to send subsequent packets to the same chip.

For each request, the memory controller identifies which chip the requested data resides in, and then lets the cache controller search the tag array corresponding to this target chip. During the tag comparison, the cache controller sends the requested block address through the channel using

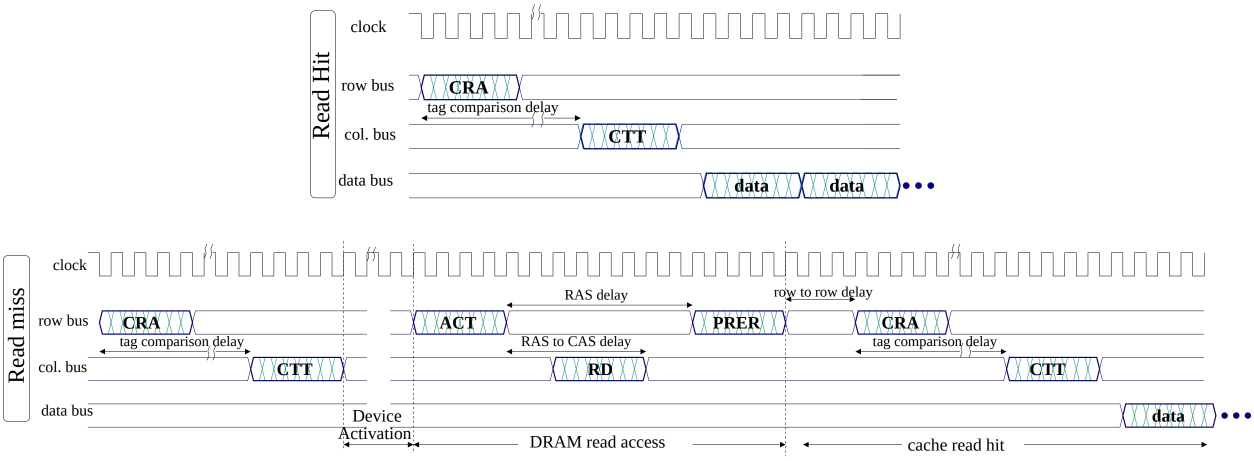


Figure 6.5: Timing diagram of the Rambus channel for near-memory cache read hit (upper) and read miss (lower) transactions.

the CRA command. After the comparison, the cache controller sends the comparison result using the CTT packet. In the case of a near-memory cache hit (tag_flag is set), data is read directly from the near-memory cache. In case of an near-memory cache miss, the cache controller uses the CTT packet to send the address of the block to be replaced and whether it is dirty or not. If the block is dirty (i.e., dirty_flag is set), then it is written to the write buffer. Meanwhile, the memory controller sends a packet to activate the chip (ACT), followed by a read command for a memory address (RD) then an optional command to precharge the data line (PRER). The memory command sequence RD-ACT-PRER is part of the original Rambus protocol. After data is read from the memory address, it is copied to the near-memory cache through the sense amplifiers. The cache controller then sends another command sequence requesting the new block, which is treated as a cache hit.

If there are no outstanding requests to a chip, then the RDRAM chip does an automatic copy of the write buffers to the correct banks during the chip idle periods. After writing the contents of the write buffers, the memory controller issues a command to send the chip to the nap state. Note that the cache commands use both the row and column control buses. This is to increase the memory pipelining, and decrease the delays for memory access (even if it is near-memory cache accesses).

## 6.4 ENERGY AND DELAY MODELING OF PA-CDRAM

In this section we develop a simple analytical energy model to highlight the main factors that affect PA-CDRAM energy savings compared to traditional memory. The memory hierarchy consists of L1, L2 caches, L3 or near-memory caches, and DRAM-core. PA-CDRAM uses the same cache capacity as in traditional memory; however, the capacity of the L3 cache is distributed on near-memory caches. Our model is based on system parameters (such as caches' access energy and latency, as well as time and power consumed by DRAM in each power state) and application parameters (such as number of L2, L3 and near-memory cache misses).

The model accounts for energy consumed in the caches ($E_{cache}$), the DRAM-core ($E_{Dcore}$), and system buses ($E_{bus}$). We do not account for the cache leakage energy (proportional to the size of the SRAM) as we compare our results against a base case with equal cache capacity, and we assume it to be negligible.

$$\begin{aligned} E_{cache} &= E_{c\_accs} \cdot \#c\_accs \\ E_{Dcore} &= E_{d\_accs} \cdot \#d\_accs + E_{trans} \cdot \#trans + P_{idle} \cdot T_{idle} \\ E_{bus} &= E_{b\_accs} \cdot \#b\_accs \end{aligned}$$

where $\#c\_accs$, $\#d\_accs$, and $\#b\_accs$ are the number of L3 (or near-memory) cache accesses, DRAM-core accesses, and bus transactions, respectively. The cache energy per access, $E_{c\_accs}$, is obtained from the Cacti tool [66] for both the L3 and near-memory caches, while the DRAM-core's energy per access ($E_{d\_accs}$), the power state transition energy ($E_{trans}$), and the idle power ($P_{idle}$) are specifications of the particular RDRAM memory chip used [46]. $T_{idle}$ is the time spent in the DRAM idle state. To simplify the memory's energy estimation, we assume an application with limited number of write-backs and capacity misses in the L3 or near-memory caches. We also assume a memory power management technique that immediately transitions DRAM-core to a low power state after each access. The low power state consume $P_{static}$. In other words, the number of memory power state transitions is double the number of DRAM accesses: $\#trans = 2 \cdot \#d\_accs$. We derive the number of cache and DRAM-core accesses as follows:

$$\begin{aligned} \#c\_accs^{B} &= \#c\_accs^{PA} = \#L2\_misses \\ \#d\_accs^{B} &= \#L3\_misses = \#c\_accs^{B} \cdot L3\_miss\_rate \\ \#d\_accs^{PA} &= \eta \cdot \#c\_accs^{PA} \cdot L3\_miss\_rate \end{aligned}$$

where the superscript indicates whether the metric represents PA-CDRAM ($PA$) or traditional memory (called $B$, the base case). $\eta = \frac{\#d\_accs^{PA}}{\#d\_accs^{B}} = \frac{near-memory\ miss\ rate}{L3\ miss\ rate}$ represents the improvement in the near-memory average miss rate with respect to the traditional L3. $\eta$ is affected by the spatial and temporal locality of the application's data, and can be estimated using cache models that predict cache miss rates [105].

To compute the amount of energy spent in the DRAM core, we estimate the time spent at each of the DRAM busy and idle periods. We compute $T_{idle}$ in terms of the execution time of an application ($T_{exec}$) and the time spent while the DRAM is reading or writing data ($T_{busy}$) as shown in the equations below. $T_{d\_accs}$ is the average DRAM access time. $T_{exec}$ is derived from the execution time of an application with access latency equals zero for data accesses beyond L2 ($T_{perfect}$),[4] and the average access time of the combined DRAM and L3 (or near-memory) cache ($T_{mem\_accs}$).

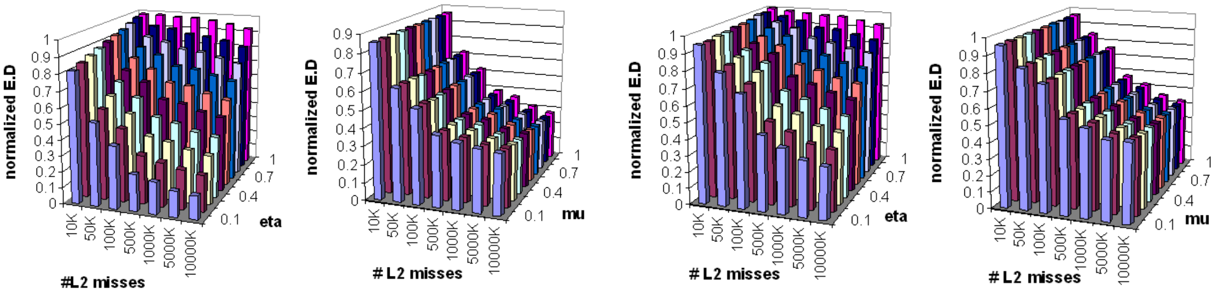
$$\begin{aligned} T_{idle} &= T_{exec} - T_{busy} \\ T_{busy} &= T_{d\_accs} \cdot \#d\_accs \\ T_{exec} &= T_{perfect} + \mu \ (T_{mem\_accs} \cdot \#d\_access) \end{aligned}$$

Where $\mu$ is the fraction of memory access time that does not overlap with processor's computation (resulting in CPU stalls). $\mu$ is affected by the processor's design (e.g., issue width, depth of Load/Store queue) and its effect on hiding memory latency. Both $\eta$ and $\mu$ are application dependent and their values range from 0 to 1 under the above assumptions. $\eta$ can exceed one in case of high capacity misses and large write-back traffic.

Figure 6.6 shows the effect of varying the number of L2 misses, $\eta$ and $\mu$ on the energy-delay product. PA-CDRAM memory consists of 256KB near-memory cache with 512B blocks versus 2MB L3 cache with 128B blocks in the base case.[5] The results shown for a duration of $T_{perfect} =$ 250msec and fixed L3 miss rates: 20% and 5% as examples for moderately high and moderately low miss rates. The graphs show that varying $\eta$ highly affects the savings in the energy-delay product in PA-CDRAM compared to the traditional memory. This is due to the fewer accesses to the DRAM-core and consequent idle time in memory. On the other hand, $\mu$ has less visible impact on the energy-delay product primarily because the majority of the L2 misses can be serviced by either the L3 or the near-memory caches, which have similar cache access latency. For smaller L3 miss rates, the PA-CDRAM energy-delay savings is smaller due to the lower memory traffic.

[4] $T_{perfect}$ is the execution time when using perfect memory and L3 (or near-memory) caches

[5] Cache hierarchies used are same as in the evaluation section (Section 6.5).



(a) L3 miss rate = 20%

(b) L3 miss rate = 5%

Figure 6.6: The effects of varying $\eta$ and $\mu$ on the energy-delay product

Thus, given the application's L2 misses, cache and memory power and delay characteristics, and estimation of the L3 and the near-memory miss rates, we can estimate whether using PA-CDRAM is more efficient than using traditional memory hierarchy under the above assumptions.

The intention of the above model is to highlight the runtime factors affecting PA-CDRAM energy and delay with respect to traditional memories. Other static factors such as per-access cache and DRAM energy and latency (considered as fixed parameters in the above model) play a role in evaluating the effectiveness of PA-CDRAM. Next, we present detailed analysis of the two memory models using simulation, and discuss the effect of the different system parameters on the energy and delay of PA-CDRAM.

## 6.5 EVALUATION OF PA-CDRAM

In this section, we describe several experiments that explore the energy and performance of PA-CDRAM. We evaluate the different PA-CDRAM configurations and their impact on energy and performance. Then, we analyze the performance and energy benefits of PA-CDRAM compared to conventional power-management employed by Rambus. We also evaluate the energy and performance implications of the centralized and distributed near-memory cache controller designs, and how well PA-CDRAM works in a multi-tasking environment, where Rambus memories are most likely to be used. Finally, we present a sensitivity analysis of PA-CDRAM with respect to varying the different system parameters. We begin with a discussion of our experimental methodology.

### 6.5.1 Methodology

To evaluate PA-CDRAM, experiments are performed using the SimpleScalar architecture simulator [75] combined with a memory module [106] that models a set of RDRAM chips connected to a single Rambus channel. The memory model also simulates the memory controller (*MC*) and its request scheduling. We extended the memory model by implementing the CDRAM memory structure and RDRAM's power management scheme with multiple power levels and a timeout policy for power mode transitions.[6] Simulations are performed using a set of applications from the SPEC2000 benchmark suite. To avoid the cold start effect, we fast forward the simulation two billion instructions and simulate the following 200 million instructions as in [107].

Our study evaluates the energy consumption and delay of PA-CDRAM against a base case that employs traditional power saving policies implemented by Rambus. Since we are concerned with the memory subsystem, we measure the energy-delay product accounting for memory energy consumption and overall delay. This metric would improve if we account for the CPU energy. Since we do not apply any processor power management and PA-CDRAM reduces execution time (as will be shown in Section 6.5.3), PA-CDRAM reduces the processor static energy by reducing the application's runtime; however, we only include memory energy in our results.

Figure 6.7 illustrates the memory models evaluated in our experiments. Table 6.2 summarizes the system configurations used in the two memory models. The base case is illustrated using a specific cache hierarchy configuration similar to the Pentium4 EE processor [108]. Latency values are given in terms of CPU cycles. Note that this is a more current cache hierarchy setting that is different than the one used in Figure 6.2. In the base case, data allocation is done linearly to keep the least number of chips in the active state [53, 54]. In PA-CDRAM, we use interleaved memory mapping to make use of all near-memory caches and use the same L1 and L2 configurations as the base case. DRAM power management in the base case uses a timeout policy for the deactivation of the RDRAM chips to the nap state.

We compute the energy consumption in the DRAM-core, caches and buses. During DRAM-core accesses, dynamic energy is consumed, which is proportional to the number of DRAM accesses. Otherwise, the DRAM-core consumes static energy that is proportional to the time spent in each

[6] We limit our approach to use the *nap* state as the only low power state, because the results in [53] and [54] showed that the nap state is energy efficient and has relatively low transition delay.

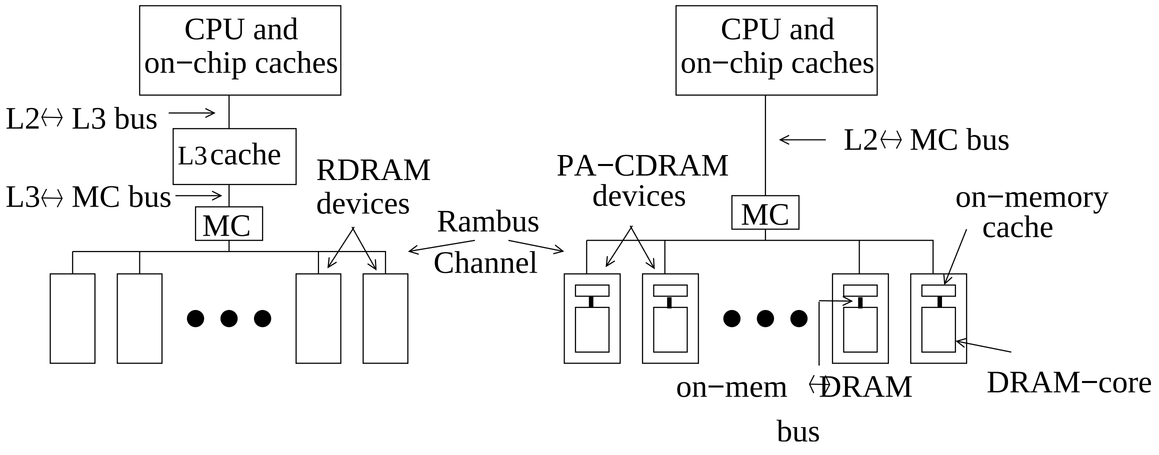


Figure 6.7: Memory organizations of the base case (left) and the proposed PA-CDRAM (right).

power state. The timing and power characteristics of the simulated RDRAM chip are for a typical RDRAM chip, namely the 256Mb/1066MHz/32 split bank architecture [46].

Access energy and latency for each cache configuration is obtained using Cacti 3.0 for 130 nm (same manufacturing technology as the Pentium EE). In Cacti, the per-access energy and latency is divided into portions consumed in tag-array and data-array (including sense amplifiers and output latches). In n-way set associative caches, tag and data arrays are accessed concurrently to reduce the total access time. In fully associative caches, tag array is replaced by fully associative decoder, after which the data array is accessed. Tag comparison takes place in the decoder. Then, the decoder drives the wordline associated with the cache entry. This optimization reduces cache per-access energy; however, it increases per-access latency. We obtain access latencies and energy when operating at cache voltage $V_{dd} = 1.3$ V. As discussed in Section 6.1, we add a conservative delay penalty of 35% for accessing logic cells in the memory chip [60]. We also add four cycles delay penalty for accessing off-chip caches.

We refer to a bus (address and data) by the two memory elements that the bus connects; for example L2↔L3 is the bus connecting L2 and L3 caches. In our evaluation, we account for L2↔L3, L3↔MC, and the Rambus channel in the base case, while accounting for L2↔MC, the Rambus channel, near-memory↔DRAM in PA-CDRAM as shown in Figure 6.7. Energy of external and internal buses are computed using the model presented in [98] and [109]. Unless stated otherwise, we evaluate a centralized cache controller design. PA-CDRAM and the base case energy models

Table 6.2: System configuration

| |
|---|
| **Processor**: 4-issue out-of-order, 2GHz |
| **Cache hierarchy**:<br>L1: on-chip 32KB iL1 & dL1 caches, DM, 32B, 2 cyc. lat.<br>L2: on-chip 256KB, 8-way, 64B, 5 cyc. lat.<br>L3: off-chip 2MB, 8-way, 128B, 15 cyc. lat<br>near-memory: 8x256KB, FA, 512B, 19 cyc.(14+5 slowdown) |
| **DRAM-core**: 8 x 32MB RDRAM , 85-120 cyc. lat |
| **Buses**: address width = 32b, $C_{in}$ = 0.5 pF, $C_{ex}$= 10pF |

are detailed in [110].

We set our experiments using memory/cache configurations that exhibited the best energy-delay product results across all applications. From experiments with different timeout values (0, 100, 500, 1000 and 5000 and 10000 cycles), we found that a timeout of 1000 cycles is the best fixed threshold for the base case, while a zero timeout threshold achieved the lowest energy-delay product on average for the PA-CDRAM. Experimenting with the cache block sizes (128, 256, 512 and 1024 bytes) and associativity (8-way and fully associative), the least energy-delay product for the L3 cache in the base case was realizable using an 8-way cache with 128B blocks. The best configuration for the PA-CDRAM was the fully associative with 512B blocks.

### 6.5.2 Energy and delay

Across all Spec2000 benchmarks, Figure 6.8 shows the average savings in energy-delay product is 28% (shown in the last column) and up to 84% (in *ammp*). The energy-delay product is an aggregate behavior; to analyze the benefits of PA-CDRAM on energy and performance independently, we decompose the energy-delay product into execution time and energy consumption. Figure 6.8 shows these metrics for each application normalized to the base case.

**6.5.2.1 Effect on delay** For most applications, there is no significant improvement in the delay over the base case due to two reasons. First, in most of the applications, L2 can service a large

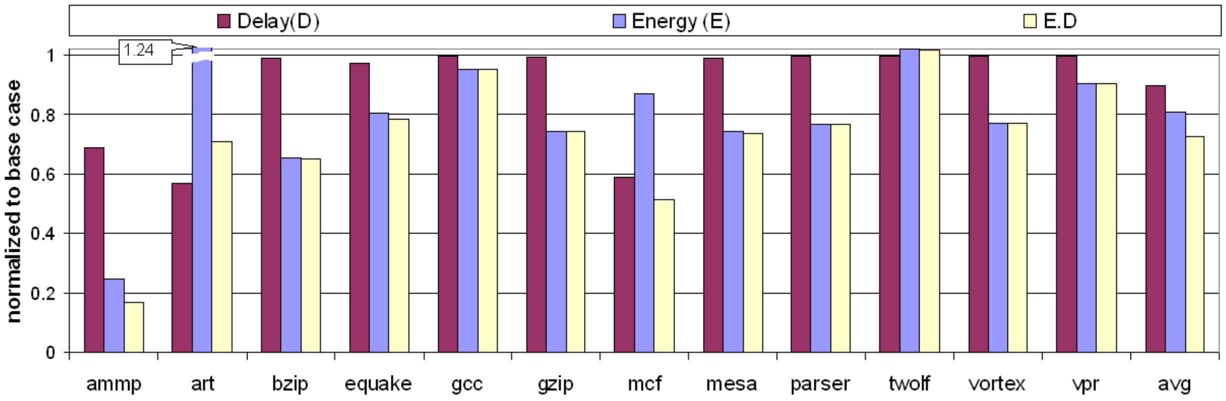
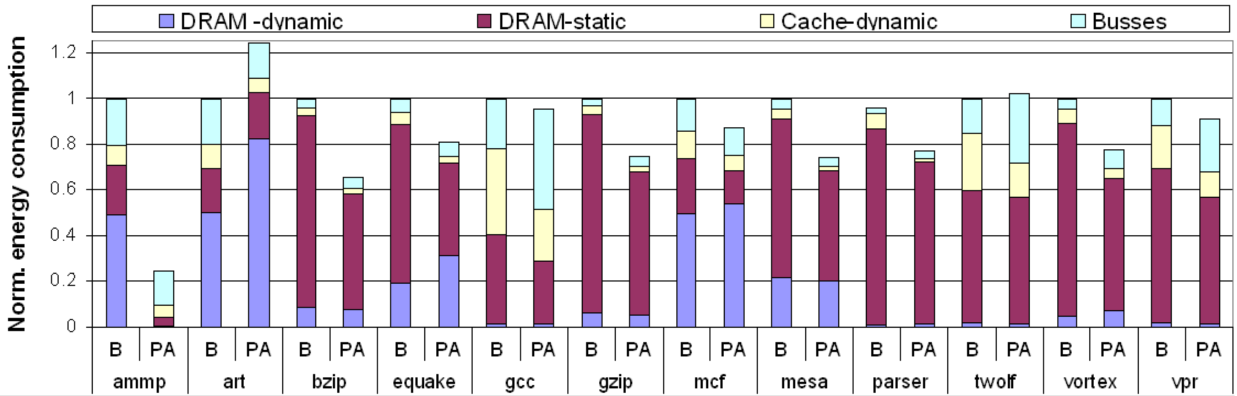


Figure 6.8: PA-CDRAM energy-delay break down normalized to the base case.

number of the memory requests. Second, most of the CPU stalls resulting from L3 misses can be masked by the execution of other independent instructions in the pipeline, that is, $\mu$ is very small. Three exceptions in Figure 6.8 are *ammp, art* and *mcf*. For these benchmarks, the total number of DRAM-core accesses is two orders of magnitude larger than the other applications. With so many memory accesses, a reduction in average memory access time significantly improved performance. This shows that near-memory caching is well suited for memory intensive applications.

Compared to the motivational example presented in Section 6.2, there is a difference in performance improvement. This difference is due to the use of a different cache hierarchy in each experiment. In Figure 6.2, we use the same cache hierarchy used in [101], which proposes a different cache configuration. Additionally, the results in Figure 6.2 use an extra cache level in memory chips for CDRAM (resulting in larger overall cache capacity than the base case). The extra cache capacity improves execution times. However, for a more fair comparison we use the same overall capacity in both memory models and a more current cache hierarchy that includes L2 cache. Hence, the overall execution time becomes less sensitive to the memory latency, which is a desirable effect in system design.

**6.5.2.2 Effect on energy consumption** From Figure 6.8, we also see that savings in energy consumption alone reached up to 76% (for *ammp*). All applications exhibit energy savings except for *art* and *twolf*. To analyze these savings (and higher consumption in *art* and *twolf*), we further



| | ammp | art | bzip | equake | gcc | gzip | mcf | mesa | parser | twolf | vortex | vpr |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| B | 49.96 | 40.10 | 22.19 | 22.73 | 0.30 | 13.97 | 23.90 | 25.66 | 4.30 | 0.59 | 5.28 | 0.80 |
| PA | 0.09 | 20.00 | 5.59 | 10.57 | 0.08 | 3.51 | 8.00 | 6.98 | 1.82 | 0.15 | 2.05 | 0.21 |

Figure 6.9: PA-CDRAM and the base case energy break down and cache miss rates

decompose the energy consumption to show where the energy is spent in each of the memory's individual components. Figure 6.9 shows the relative consumption of the DRAM-core dynamic energy (DRAM-dynamic), DRAM-core static energy (DRAM-static), cache access energy (cache-dynamic), and bus energy. The bottom of Figure 6.9 shows the cache miss rates for L3 in the base case and average miss rate of all near-memory caches in PA-CDRAM. Note that miss rates in PA-CDRAM are lower than the base case in all applications. Thus, there exist fewer accesses to the DRAM-core in case of PA-CDRAM. From figure, $\eta$ ranges from 0.25 to 0.5 (except in *ammp* = 0.002), which indicate a potential for energy saving as discussed earlier.

In all applications, the DRAM-static energy was reduced due to the increase in the duration of DRAM idle periods versus active periods in the base case. With respect to the DRAM dynamic energy, some applications, namely *ammp, bzip, gcc, gzip, mesa, twolf*, and *vpr*, have lower energy due to lower miss rates in the near-memory cache that filter some of the accesses to the DRAM-core. All the above applications have relatively small $\eta$ (around 0.26). Note the effect of the extremely low value of $\eta$ on reducing the energy-delay product of *ammp*. However, applications like *art, equake, mcf, parser*, and *vortex*, suffer an increase in DRAM dynamic energy, even though near-memory cache miss rates were reduced ($\eta$ values around 0.42). This increase is due to the relatively large near-memory cache block size that caused these applications to access unnecessary data from the DRAM-core.[7] During these excessively large transfers, the DRAM consumes extra energy by

[7] This factor was not considered in the model in Section 6.4 for simplicity.

Table 6.3: Per-access latency and energy break down of L3 and near-memory caches.

| cache | tag-array | data-array | total |
|---|---|---|---|
| L3 latency (ns) | 3.69 | 5.41 | 5.41 |
| L3 energy (nJ) | 0.388 | 4.614 | 5.00 |
| PA latency (ns) | - | 4.94 | 4.94 |
| PA energy (nJ) | - | 3.084 | 3.44 |

reading/writing data that is never used by the application. We deduce that increasing the spatial locality for an application saves further energy in PA-CDRAM dynamic energy compared to the base case, and vice versa.

As cache access energy is proportional to the length of activated bitlines and wordlines at each access, dividing the cache into smaller near-memory caches reduces energy. Table 6.3 lists the breakdown of per-access energy and delay for both L3 and near-memory caches as obtained from Cacti. For near-memory caches, tag decoder latency and energy is included in the data-array side. Near-memory access latency in the table excludes the 35% slowdown penalty. The figure shows that when using PA-CDRAM there is a decrease in cache energy across all applications, even with small reduction in miss rates.

The total energy of a bus depends on its capacitance and activity. PA-CDRAM can reduce bus energy consumption (as in *ammp, art, equake, mcf*, and *mesa*) by reducing the total bus capacitances compared to the base case (three external buses versus two external and one internal bus for PA-CDRAM). However, in the other applications, bus energy increases due to the increased Rambus channel activity. Thus, applications with high L2 misses and low L3 misses consume more bus energy in PA-CDRAM. On the other hand, for applications with relatively high L3 misses, frequent activity occurs in L3↔MC and the channel, resulting in higher bus energy in the base case.

### 6.5.3 Near-memory versus near-processor caches

In this study, we evaluate the potential energy and performance benefit of allocating the capacity of L3 cache closer to the processor versus closer to the memory. In our experiments, we compared

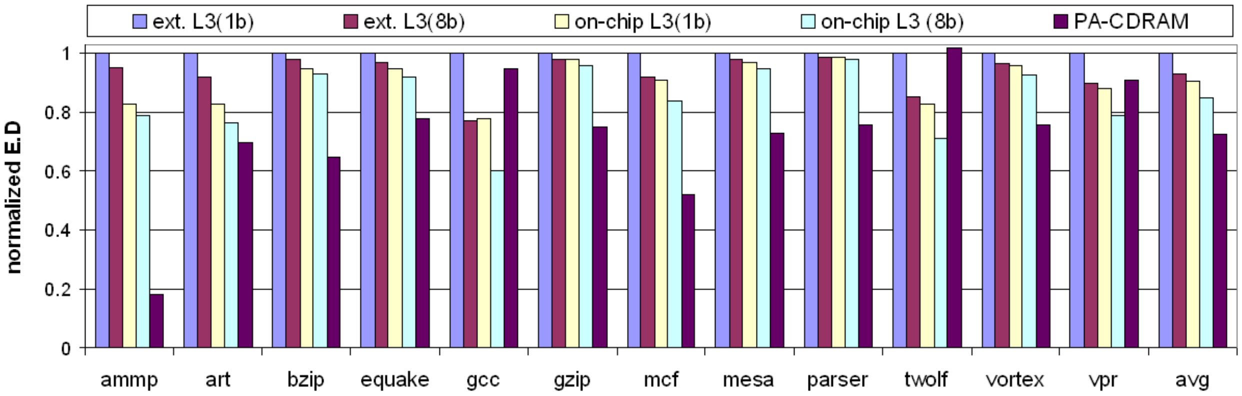


Figure 6.10: Energy-delay product of near memory versus near-processor cache organizations normalized to the base case.

PA-CDRAM with systems where the majority of the cache capacity is allocated as: (1) large on-chip L2 cache and no L3 cache (not shown here), (2) large on-chip L3 cache, or (3) large off-chip cache. The capacity of the large cache in each of these systems is equal to 2MB. In the first case, we observed that the energy-delay product is much higher than having a large L3 cache (our base case described earlier). The increase is, on average, 2.97 times and up to 5.34 times the base case. Larger L2 size causes longer access latency and energy per access than a smaller L2. Since L2 is very frequently accessed, increasing its access latency and energy results in a significant degradation in performance and increase in the total energy consumption. Thus, we show the more fair comparison against a system with a large L3 cache.

Figure 6.10 compares the energy-delay product of using traditional memory with off-chip L3 or on-chip L3 caches versus PA-CDRAM with near-memory caches. On-chip L3 improves over off-chip L3 as it saves the energy consumption and the extra delay spent in accessing the L2↔L3 external bus. Comparing PA-CDRAM against a memory hierarchy with large on-chip L3 cache shows that most of the applications achieve lower energy-delay product. This is due to the use of smaller near-memory caches and the lower miss rates in PA-CDRAM. On the other hand, *gcc*, *twolf*, and *vpr* experience high L2↔MC and MC↔near-memory bus traffic which overshadow savings in the other memory components. However, across all applications, PA-CDRAM improves the average energy-delay product over using a large on-chip L3 cache by 17%.

Moreover, we compare the benefit of using an L3 cache (on-chip and off-chip) that is divided into 8-banks. We find that even when compared with 8-banks on-chip L3 cache (configuration with lowest energy consumption in the base case), PA-CDRAM achieves 15% lower energy-delay product.

### 6.5.4 Cache controller location

We quantify the trade-offs of choosing the location of the near-memory cache controller with respect to the overall performance and energy consumption. We simulate the centralized and distributed cache controllers and penalize the near-memory cache access latency accordingly. For the centralized controller, we use Cacti's latency computation to slowdown the access latency, except for the tag comparison (done outside the DRAM chip). For the distributed design, we apply the penalty (slowdown) to the entire near-memory access latency (including tag comparison which takes place inside the DRAM chip). As expected, the total execution time of an application is slightly faster (by up to 1.5%) when using a centralized controller. On the other hand, the energy consumed in the memory buses is up to 5% less for the distributed controller design. Figure 6.11 shows the effect of the cache controller location on the bus energy, the application's execution time and the overall energy-delay product. Although the distributed design consumes less bus energy, the overall energy was actually increased. This is due to the longer execution times that increases the DRAM-static energy, which overshadows the bus energy saving. Although the centralized design has an 0.4% average improvement in energy-delay, the distributed design has the advantage of being backward compatible. Hence, we stress an important feature of the distributed controller: it enables the use of traditional Rambus and PA-CDRAM chips interchangeably while connected to the same memory controller. In that sense, it is possible to do incremental/partial deployment of the PA-CDRAM chips, if it is not possible to do it homogeneously. We leave further study of this issue for future work.

### 6.5.5 Effect of multiprocess and multithreaded environments

The energy profile of an application running in a single task environment may differ from running the same application in a multi-process system with preemption or in a multi-threaded environment with multiple processors. This difference arises from the fact that an application's locality of reference is disturbed by the execution of other applications, which will populate the cache with their own data. The result is higher miss rates experienced by each application. It is important

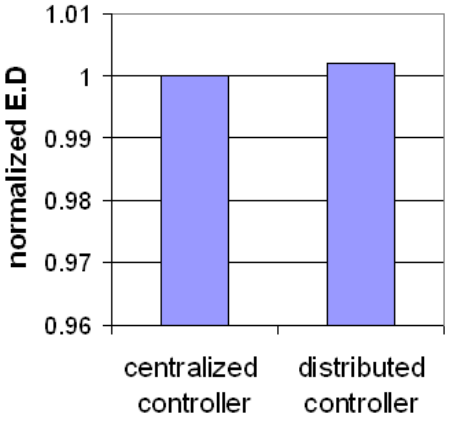


Figure 6.11: Energy-delay product of distributed controller normalized to centralized cache controller design.

to evaluate how these higher miss rates affect energy-delay of PA-CDRAM since it is intended for such environments.

We start first with single processors running multiple applications. Although our simulation infrastructure does not directly support a multi-tasking workload, we can approximate the effect of context switches. Our approximation uses a task scheduler that invalidates all the cached data for the expiring application (task) before resuming execution of the ready task.[8] The invalidation writes back all the dirty cache blocks to memory. We trigger an interrupt service routine every 10ms (similar to the time slice in Linux). The interrupt drains the processor pipeline and flushes the data in the caches. Upon the interrupt termination, the application's execution is resumed.

Figure 6.12 illustrates the overhead of context switching compared to single task execution. In all applications except *mesa* and *vortex*, the overhead of context switching is lower in PA-CDRAM than in the base case, for two reasons. First, the time overhead of flushing the L3 cache is larger than the near-memory caches as flushing all the small near-memory caches can be done simultaneously, while flushing the L3 cache serializes the write backs to the memory chips. Second, since the number of blocks in the L3 cache (2MB/128B) is larger than the near-memory caches (2MB/512B), the L3 cache has more address decoding and thus more energy is consumed. In *mesa* and *vortex*, the

[8] This is a worse-case behavior since usually the cache will not be completely flushed.

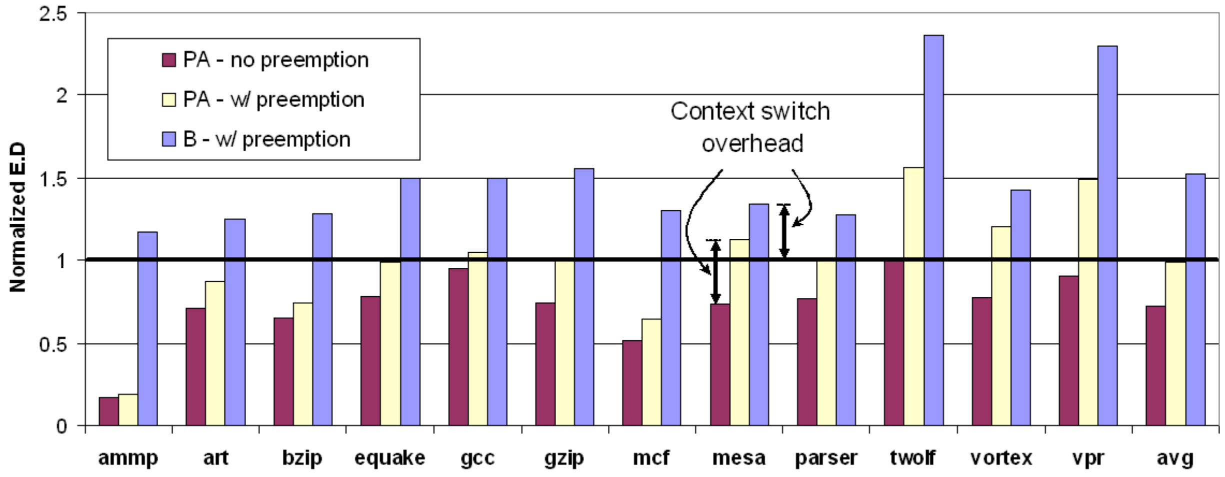


Figure 6.12: Energy-delay product with and without pre-emption, normalized to the base case without preemption.

energy-delay product of the context switching overhead is higher in PA-CDRAM due to an increase in DRAM dynamic energy that exceeds the time savings from the cache invalidation as mentioned above. This energy increase is due to the relatively large number of near-memory cache writebacks factored by the large block size to be written to the DRAM-array.

Furthermore, we evaluate PA-CDRAM energy consumption when used in a multi-processor environment. To emulate the multi-processor effect, we generate traces of L3 accesses for all applications. For each application pair, we merge their traces based on the timestamp of each cache access. We divide the memory address space such that each application has access to half of the total memory size. To have all data resident in memory, we increase the memory size to 512MB divided among 8 chips. We maintain the L3 (in the base case) and near-memory (in PA-CDRAM) caches at the same capacity.

Table 6.4 shows the results of running pairs of application traces on our simulator. We report energy values normalized to the base case. Delay results are irrelevant since we are running time-stamped traces rather than actual execution of applications. As expected, running multiple applications simultaneously increases the number of cache misses compared to the sum of cache misses when running each application individually. This observation is true for both PA-CDRAM and the base case. Although the number of misses in near-memory caches is higher than the single processor case, the total number of near-memory cache misses is still lower than L3 misses. The lower misses in PA-CDRAM is because PA-CDRAM originally achieves much lower miss rates than the base case when running single application as shown in Figure 6.9. From Table 6.4, most application pairs experience lower energy consumption by using PA-CDRAM memory than base case. In the base case, access to multiple DRAM chips (from the two applications) increases the number of active chips, hence increases the DRAM-core static energy. However, less energy impact is noticed in PA-CDRAM due to the interleaved data allocation and the immediate deactivation of the DRAM-core. Average energy savings across all application-pairs is 20%. Memory intensive applications like *art* and *mcf* have a large memory working set that reside in the near-memory cache. When running another application concurrently, large number of conflict misses occur. The higher miss rate increase the performance degradation as well as the energy consumed in DRAM-core and near-memory caches. The effect is less severe in the base case because L3 cache has larger number of cache blocks.

### 6.5.6 Sensitivity to design parameters

We investigate the effect of varying the different system parameters on the energy and performance of PA-CDRAM with respect to traditional memory. We vary the following parameters: (1) cache capacity for both the L3 and the near-memory caches to study the effect of the capacity misses on the system, (2) the CPU frequency to study the effect of improving the average memory access time on the total performance, and (3) the slow-down experienced by the logic embedded in the DRAM chip.

**6.5.6.1 Effect of varying the cache size** Varying the cache size affects both the miss rate and the cache access costs (latency and energy). Figure 6.13 shows the average delay, energy, and energy-delay product of PA-CDRAM while varying the total near-memory cache size from 512KB to 4MB (excluding L1 and L2). The results are normalized to the base case with L3 cache of corresponding size. Note that while the small cache sizes tested (512KB and 1MB) are too small for a typical L3 cache, we test at those sizes to demonstrate the trend. Increasing the cache size reduces the L3 (or near-memory) miss rates, thus better performance is achieved in both cases. When the miss rate is reduced, lower accesses to the DRAM-core occur, and accordingly, the relative savings in delay of PA-CDRAM to the base case is lower for large cache sizes.

Table 6.4: PA-CDRAM energy consumption normalized to the base case when running application pairs interleaved.

| | ammp | art | bzip | equake | gcc | gzip | mcf | mesa | parser | twolf | vortex | vpr |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ammp | 1.82 | 0.94 | 0.14 | 0.16 | 0.66 | 0.14 | 1.10 | 0.14 | 0.13 | 0.73 | 0.17 | 0.80 |
| art | 0.94 | 1.48 | 1.31 | 1.30 | 1.24 | 1.23 | 1.36 | 1.18 | 1.26 | 1.35 | 1.19 | 1.32 |
| bzip | 0.14 | 1.31 | 0.63 | 0.80 | 0.65 | 0.66 | 1.01 | 0.72 | 0.69 | 0.75 | 0.71 | 0.70 |
| equake | 0.16 | 1.30 | 0.80 | 1.00 | 0.69 | 0.74 | 1.06 | 0.73 | 0.75 | 0.82 | 0.74 | 0.81 |
| gcc | 0.66 | 1.24 | 0.65 | 0.69 | 0.63 | 0.66 | 1.01 | 0.61 | 0.66 | 0.64 | 0.66 | 0.70 |
| gzip | 0.14 | 1.23 | 0.66 | 0.74 | 0.66 | 0.72 | 1.02 | 0.67 | 0.72 | 0.71 | 0.70 | 0.70 |
| mcf | 1.10 | 1.36 | 1.01 | 1.06 | 1.01 | 1.02 | 1.06 | 0.79 | 0.81 | 0.92 | 0.90 | 0.95 |
| mesa | 0.14 | 1.18 | 0.72 | 0.73 | 0.61 | 0.67 | 0.79 | 0.73 | 0.70 | 0.60 | 0.73 | 0.63 |
| parser | 0.13 | 1.26 | 0.69 | 0.75 | 0.66 | 0.72 | 0.81 | 0.70 | 0.75 | 0.72 | 0.75 | 0.74 |
| twolf | 0.73 | 1.35 | 0.75 | 0.82 | 0.64 | 0.71 | 0.92 | 0.60 | 0.72 | 0.77 | 0.66 | 0.89 |
| vortex | 0.17 | 1.19 | 0.71 | 0.74 | 0.66 | 0.70 | 0.90 | 0.73 | 0.75 | 0.66 | 0.71 | 0.70 |
| vpr | 0.80 | 1.32 | 0.70 | 0.81 | 0.70 | 0.70 | 0.95 | 0.63 | 0.74 | 0.89 | 0.70 | 0.97 |

With respect to energy consumption, the effect of varying the cache capacity is not as trivial. The general trend is that increasing the cache size reduces the capacity misses; thus, less energy is consumed due to fewer cache replacements and DRAM accesses. However, when increasing the cache size, the cache energy per hit access increases causing an increase in the total memory energy consumption. Comparing the energy consumption of the base case versus PA-CDRAM, we find that applications are divided into two groups: (1) where PA-CDRAM consumes less energy at all cache sizes as in *bzip, gzip, mesa, parser*, and *vortex*, or (2) where PA-CDRAM consumes less energy only at large cache sizes. For the first group, PA-CDRAM performs better due to accessing smaller individual caches (with lower per-access energy). For the second group, at the small cache sizes tested (512KB and 1MB), too many near-memory cache replacements takes place due to the limited number of blocks ($\frac{512KB}{8 \cdot 512B}$ = 128 blocks) per individual near-memory cache. Results for individual applications can be found in [110].

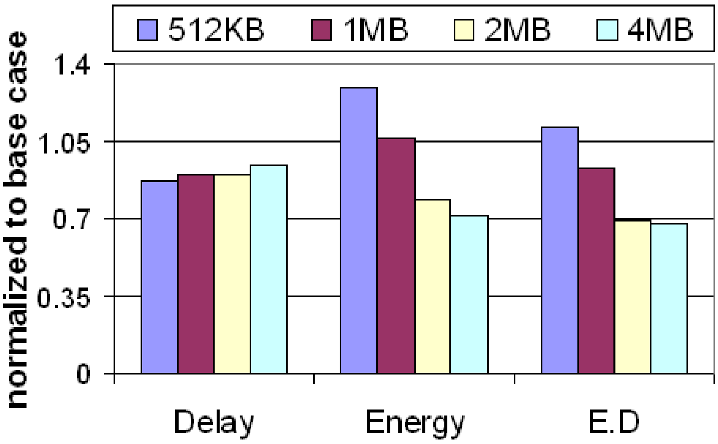


Figure 6.13: The effect of varying the cache size on execution time, energy and energy-delay product normalized to the base case.

**6.5.6.2 Effect of varying the CPU frequency** Increasing the CPU frequency increases the speed gap between the memory and the CPU; thus, increasing the total execution cycles of an application. Figure 6.14 shows the energy-delay product for PA-CDRAM at different clock rates normalized to the base case at 2 GHz. We choose to fix the frequency at the base case to demonstrate the effect on the energy consumption. In our experiments, most of the applications exhibit minimal variation in execution times. These applications are mainly CPU bound where the processor is able to mask most of the L2 miss stalls. Thus, the relative benefit of PA-CDRAM on delay is negligible compared to the base case (as described in Section 6.5.3). In memory bound applications (*ammp, art* and *mcf*), increasing the CPU frequency compounded with the larger average memory access times in the base case, results in the base case suffering from significantly larger delays than PA-CDRAM. That is, at higher CPU frequency the value of $\mu$ increases significantly in these memory intensive applications, making PA-CDRAM an efficient solution.

From the energy perspective, increasing the CPU frequency reduces the application's execution time, and thus the duration of the idle periods in the DRAM. This reduces the consumption of the static energy in the DRAM-core. In contrast, the dynamic energy in the DRAM and the caches is not affected by the processor frequency as they are mainly dependent on the size of the transfer rather than the frequency. Although higher processor frequencies – with all other factors unchanged – reduces the memory's total energy (shorter idle periods), memory's energy consumption can increase due to other factors like larger memory capacities and higher memory traffic. Figure 6.14 shows the effect of varying the CPU frequency on the energy-delay product.

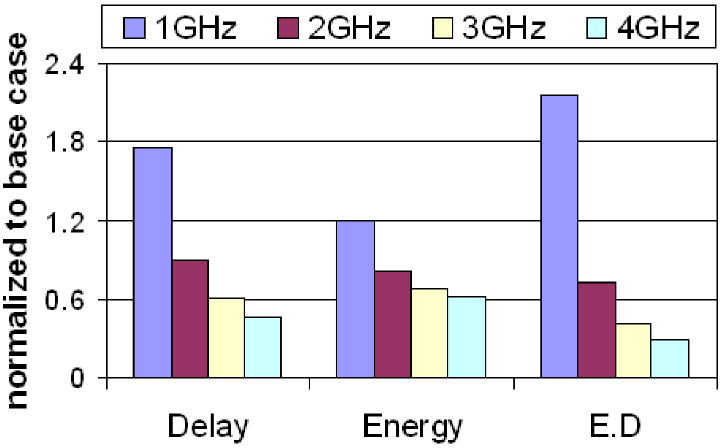


Figure 6.14: The effect of varying the CPU frequency for the PA-CDRAM, normalized to the base case on execution time, energy and energy-delay product.

**6.5.6.3 Effect of logic slowdown** To show the effect of the embedded logic manufacturing technology on the overall performance of systems using PA-CDRAM, we vary the logic slowdown factor of the near-memory data array (assuming a slower distributed cache controller). We add a delay penalty to the overall cache access delay obtained from Cacti. This penalty ranges from 0%

(fast SRAM) to 50%. The execution time is normalized to the PA-CDRAM case where there is no performance penalty for the logic cells. From Figure 6.15, we notice that the performance degradation in the near-memory cache controller design is relatively insignificant. Even when compared to a centralized cache controller design (not shown here), the overall slowdown in performance over fast SRAM is 1.5% on average (up to 6.3%) with a 50% slowdown penalty. This result shows that delays resulting from manufacturing embedded logic in DRAM chips do not represent any critical delays on the overall system when using PA-CDRAM memory.

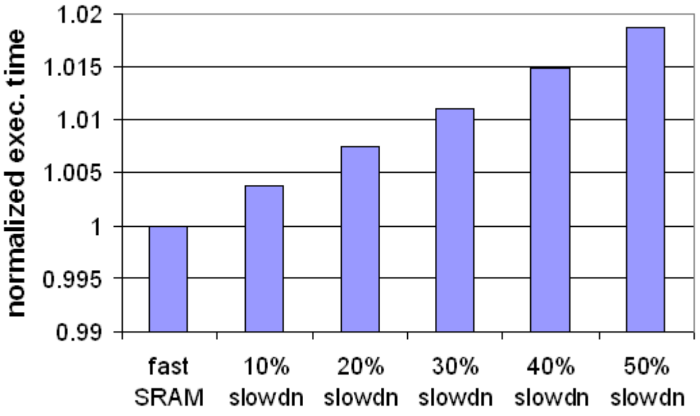


Figure 6.15: The effect of the logic slowdown in the near-memory cache on the total execution time normalized to fast SRAM case.

**6.5.6.4 Effect of CPU and memory bus bandwidth** External buses can pose performance bottleneck for accessing external caches and memory. Our earlier results assume ideal external buses with infinite bandwidth to avoid the impact of such a bottleneck. To study the impact of bus latencies on the overall performance, we limit the bandwidths of the CPU and memory buses in the base case and PA-CDRAM. The CPU bus bandwidth is limited by the bus speed, the bus width and the data rate (single, dual, or quad). We assume a 64b width CPU bus. Intel uses quad-rate (sends 4 bits/cycle) buses that operate at 200, 266, or 333 MHz [111]. Hence, CPU bus bandwidth can range from 6.25 GB/s to 10.4 GB/s ($\frac{64}{8} * bus\_freq * 4$). The limited bandwidth increases the latency of filling an L2 block between 13 and 21 cycles in our setting. RDRAM memory bus provides data bandwidth between 1.6 GB/s and 12.4 GB/s [67]. This range of bus bandwidth increases the latency of reading an L2 block from memory by 11 to 80 cycles. It is typical for CPU buses to be faster than memory buses.

Figure 6.16, shows the normalized energy-delay product using a mixture of CPU and memory bus bandwidths. The figure shows that energy-delay saving is more sensitive to memory bus latency since near-memory caches use this bus to fill the L2 cache. Although it is faster to transfer an L2 block from an L3 cache than from a near-memory cache (faster CPU buses), an L3 miss has a much higher latency than a near-memory cache miss. Hence, PA-CDRAM can improve performance even with limited bus bandwidth. In general, the larger the bandwidth of both buses, the higher the saving PA-CDRAM can achieve.

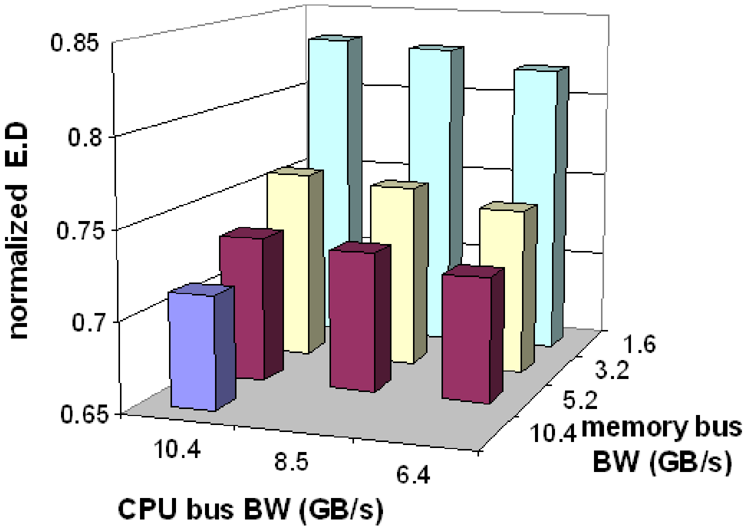


Figure 6.16: The effect of different CPU and memory bus bandwidth.

## 6.6 DISCUSSION

Manufacturing of PA-CDRAM can benefit from new advances in embedded DRAM technology (eDRAM). New fabrication technologies are successful in manufacturing eDRAM cells with significantly smaller size than traditional SRAM cells. Each eDRAM cell consists of a single transistor and a single capacitor (1T1C cell) as opposed to six transistors in case of SRAM cell. eDRAM from NEC and 1T-SRAM from Mosys are examples of technologies that apply the 1T1C concept. Due to the smaller cell sizes, higher densities can be achieved. Hence, larger capacity in smaller chip area is realizable. On the other hand, access latency of embedded DRAM is slightly higher as mentioned in Section 2.5.

The smaller form factor of 1T1C reduces both active and leakage energy. The relatively large SRAM cell size contains longer metal lines, which create higher capacitance than the ones found in 1T1C cell. Lower cell capacitance translates to lower current draw and power consumption; hence, reducing the cell active power. In addition, 1T1C cell contains one leakage path as opposed to four leakage paths in an SRAM cell. Hence, lower leakage current is drawn in an eDRAM cell. Furthermore, the eDRAM optimized refresh current is significantly less than the leakage current of an equivalent 6T memory array [112].

Looking for the impact of PA-CDRAM beyond off-chip memory and caches, we qualitatively analyze the impact on the CPU energy consumption and performance. Since PA-CDRAM does not influence the CPU activity or affect the number of accesses to on-chip caches, we expect that the deployment of PA-CDRAM does not affect the dynamic energy in both. However, it does affect their static power. Among the benefits of PA-CDRAM is improvement in overall system performance. Hence, CPU-core and all on-chip units are expected to be powered up for smaller periods of time. Accordingly, we can achieve a reduction in static energy due to the shorter active periods of these components.

## 6.7 CONCLUSION

In this chapter, we propose an architectural optimization to CDRAM to save energy in DRAM memory and off-chip caches. We explore the energy efficiency of *near-memory caches* rather than conventional cache hierarchies, where most of the cache capacity is allotted “closer” to the CPU. PA-CDRAM can be used as an alternative to traditional power-aware memories to conserve energy and improve performance. PA-CDRAM reduces the memory’s energy consumption by (1) bringing the cache closer to the memory to exploit the high memory bandwidth, and (2) distributing the external cache into smaller caches that have a low access energy and latency, and (3) increasing the DRAM-core idle periods due to the low miss rates of near-memory caches. Three main parameters affect the degree of benefit of PA-CDRAM over traditional memory: higher L2 misses, lower $\eta$ (miss rate ratio), and higher $\mu$ (memory stalls), all lead to higher energy and delay benefits obtained from PA-CDRAM. Compared to traditional memory using a time-out power management, PA-CDRAM saves up to 76% energy consumption (19% on average). Moreover, PA-CDRAM reduces the energy-delay product by up to 84% (28% on average), where the highest gains are for memory-intensive applications and for applications with relatively high spatial locality.

Our evaluation shows that, PA-CDRAM is more energy-delay efficient than allocating a large fraction of the total cache capacity to on-chip L2, on-chip L3 or off-chip L3. In a multi-tasking environment, PA-CDRAM can lower the context switch overhead of cache invalidation through simultaneous flushing of dirty blocks in all on-memory caches. With the increase in cache capacity and CPU frequency in current and future processors, the energy savings in PA-CDRAM is expected to increase.

# 7.0 CONCLUSIONS AND FUTURE WORK

## 7.1 CONCLUSIONS

To achieve the maximum system power efficiency, it is desired to reduce the energy consumption in most of the system components. The majority of power management techniques proposed in the literature focus on reducing the energy consumption in a single component in the system. We have shown examples to demonstrate that applying simultaneous power management policies to independently save power in multiple components do not necessarily guarantee reducing the overall system energy. This counter-intuitive result is due to policies being oblivious to their effect on other components' energy consumption, which may result in increasing the overall system energy.

This dissertation tackles the problem of optimizing the overall system energy by addressing the system components that contribute to the majority of the system power consumption. We focus on CPU, caches, and DRAM memory. The dissertation proposes three power management techniques that optimize the **c**ombined energy consumption of at least two system components at a time. First, we present a collaborative OS and compiler power management technique for reducing the energy consumption in real-time systems. The technique focuses on reducing the CPU (including on-chip caches) power while accounting for the memory energy consumption with the objective of reducing their combined energy consumption. The collaborative approach enables better detection of dynamic slack time in the system. To detect and utilize this slack, we use compiler-inserted hints that evaluate–at run-time– the worst-case remaining number of cycles for an application to finish execution at each hint location. At run time, information from hints is passed to the operating system to periodically schedule the proper speed. The operating system decides on the best operating frequency that reduces the combined energy consumption of the CPU and the memory subsystems. The contribution of this work is threefold.

1. We present a novel technique that involves the collaboration between the OS and the compiler

to collect both run-time information and path-dependent information. The strength of our collaborative scheme lies in three properties. First, a separate PMH placement algorithm can be devised to supply the OS with the necessary timing information about an application at a rate proportional to the PMP invocation. Second, the actual speed-change is done seldom enough by the OS at pre-computed time intervals (every PMP-interval) to keep the overhead low. Finally, by giving the OS control to change speed, our scheme controls the number of executed PMPs independent of any execution path.

2. We show the effect of reducing the CPU frequency on increasing the memory energy consumption. We present speed scheduling schemes that compute the proper CPU frequency such that the sum of the CPU energy and the memory energy consumption is minimized.
3. We evaluate our technique on time-sensitive applications running on two commercially available processors with dynamic voltage scaling. We show that our technique can achieve significant energy reduction over no power management, OS-directed and compiler-directed power management techniques.

The main results of this work are:

- The number of speed changes during the application's execution can be determined such that the overhead of excessive speed changes is avoided. The best number of speed changes is dependent on the overhead of each speed change and the variability in the execution times among each program invocation.
- Collaboration between the operating system and compiler is more efficient than techniques implemented in a single system-layer in terms of detecting slack that can be used to lower the CPU power. The collaboration enables the detection of dynamic slack before the application finishes execution.
- The speed scheduler can limit the static energy consumed by other components in the system by preventing the operating frequency from going below a break-even frequency. That frequency is chosen to avoid excessive energy consumption in different parts of the system.

Second, we propose an integrated DVS policy for reducing the energy consumption in chips with Multiple-clock domains. We mainly focus on the CPU-core and L2 cache. We present two techniques: (1) an online heuristic-based policy and (2) an offline machine learning based approach. Both techniques follow an integrated power management approach that controls the speeds in each domain based on the knowledge of activity in the other domains. The integrated approaches achieve better energy savings than an local DVS approach that only have localized view of activities. The contribution of this work is fourfold.

1. We identify a significant inefficiency in current online DVS policies, and show the sources and implications of this inefficiency.
2. We propose a heuristic-based online integrated DVS policy that adapts the core and L2 cache speeds in a way that avoids these inefficiencies and taking into account domain interactions.
3. We present a more flexible machine learning based integrated DVS approach. The approach automatically generates integrated DVS policies customized for a given class of applications running on a given system.
4. We experimentally evaluate the two integrated DVS techniques against a local-DVS policy, which forms its decisions based on local information collected about each domain. Both integrated DVS techniques show positive gains in terms reducing the energy-delay product with minimal impact on performance.

The main results of this work are:

- There are interactions among different domains that affect the workloads in each domain especially when varying the domains' frequency independently. These interactions can mislead the DVS speed scheduler when applied in isolation.
- Simple embedded processors are more likely to benefit from integrated policy compared to a local one. However, the energy savings from an integrated policy is higher in high-end processor when compared to no-power management.
- As expected, online integrated DVS policies show lower energy savings over a learning-based policy. This is due to more information obtained during the learning phase that enables the learning-based policy to make better decisions. However, in case the learning phase in expensive or infeasible, then an online policy represents a good solution.

Third, we address the energy consumption in off-chip DRAM memory and caches. We proposed a Power-Aware Cached DRAM (PA-CDRAM) that exploits the wide internal bus to improve the average memory latency, and to reduce energy consumption in main memory and external caches. PA-CDRAM improves the original CDRAM model by managing the interaction of the cache and memory. This improves the performance and energy consumption in the memory hierarchy. The contribution of this work is threefold.

1. We propose near-memory caches for energy reduction in the memory hierarchy. This reduction comes while maintaining part of the performance gains provided by CDRAM.
2. We describe an implementation of PA-CDRAM that integrates a near-memory cache in a Rambus chip (RDRAM). This description includes the changes made to a RDRAM chip, the near-memory cache controller, and the communication protocol. We also describe how our implementation can maintain backward compatibility with existing Rambus memories.
3. We experimentally evaluate PA-CDRAM and shows that PA-CDRAM is more energy efficient than a traditional Rambus memory hierarchy employing a time-out power management policy.

The main results of this work are:

- Having near-memory caches can improve performance and energy consumption, given that they are designed with the proper configuration.
- PA-CDRAM performs better than the traditional memory hierarchy for three reasons: (1) PA-CDRAM exploits the high internal memory bandwidth by bringing the cache closer to the memory, (2) it reduces the effective access energy and latency by distributing the external cache into smaller caches that have low access energy and latency, and (3) it increases the DRAM-core idle periods by lowering the miss rate of the near-memory caches.
- In a multi-tasking environment, PA-CDRAM can lower the context switch overhead of cache invalidation through simultaneous flushing of dirty blocks in all on-memory caches.

## 7.2 FUTURE WORK

Building upon this doctoral thesis, we plan to pursue two orthogonal research directions. First, we would like to investigate several open questions that were revealed during the course of this thesis research. These issues are worth investigation to be used in improving the efficiency of most power management techniques proposed in the literature. In particular, questions regarding the following issues:

**System-wide power management.** Throughout this dissertation, we presented techniques that optimize the energy consumption of two components in the systems. A natural extension to this work is to explore the interaction of more than two components in the system to achieve a system-wide power optimization. In this dissertation, we explored the interaction among the

CPU, cache, and memory. Are there other critical interactions among other components such as: disks, buses or network devices that need to be accounted for?

**Synergy of power management policies.** An alternative way to system-wide power management is to consider integrated multiple power management policies that target different system components. In this dissertation, we proposed three techniques for different component combinations in the system. It is interesting to discover what impact on the overall system power when we integrated two or more of these techniques. Does integrating multiple power management techniques introduce unnecessary complexity or achieve significant power savings compared to using one power management technique in the system?

**Dynamic- AND leakage- aware power management in deep submicron technology.** With the decreasing feature size, leakage power becomes increasingly the critical factor in a chip's total power dissipation. An important, yet currently open question is: How to effectively combine schemes for the management of dynamic power and leakage power so that the total power consumption is reduced with little impact on performance?

**Cost-effective power management control granularity.** Both temporal granularity (how often to trigger power management controls), and spatial granularity (what to control: entire chip versus individual functional units) of power management decisions play a direct role in the achievable power savings. In most cases, finer control results higher power savings at the expense of higher design complexity. Is there a unified quantitative framework that applies to wide range of power management designs and is powerful enough to express the power savings versus complexity trade-offs in the process of selecting control granularity?

Second, we plan to further explore the application of statistical machine learning techniques for providing solutions that adapt the hardware configurations based on the run-time behavior of the workload. Our preliminary work with machine learning for dynamically adapting the processor and cache voltages showed significant energy savings over heuristic approaches. We believe that there are plenty of computer architecture problems (such as adaptive power-management and thermal-management) where machine-learning can improve over traditional techniques. For example in memory-power management, building a machine learning technique that reacts to changes in resource demands at run-time (e.g., due to increased request arrival rate) will likely outperform the widely-used fixed timeout power-management policy. This direction is aligned with the idea of self-optimizing, and self-managing autonomic systems proposed by IBM. This area is relatively new, and is a fertile ground for novel applications of machine learning solutions to architectural problems.

The ultimate goal of power management technique is to significantly reduce the overall system energy. Toward this goal, we presented techniques that were able to reduce the energy consumption in multiple system components. We focus mainly on the major power consumers in the system. The techniques in this dissertation highlights the importance of designing power management schemes that consider multiple components and their interactions (in terms of power and latency) in the system rather than applying multiple isolated power management polices. This study should lay the foundation for further research in the domain of integrated power management, where multiple system components are controlled by a single power manager.

# APPENDIX

# DERIVATION OF DVS FORMULAS

## A.1 DERIVATION FOR PROPORTIONAL CLOSED-FORM EQUATION

In this section we first present Lemma 1 (and its derivation), which we use to derive the closed-form formula for Proportional speed scheduling scheme (Equation 4.3).

### A.1.1 Derivation for Lemma 1

**Lemma 1.**

$$\phi_i = A_i(B + \sum_{l=1}^{i-1} \phi_l) \Longrightarrow \phi_i = A_i B \Pi_{l=1}^{i-1}(1 + A_l) \tag{.1}$$

**Proof** (by induction):

Base case: at $i = 1$, it is trivial to see that the left hand side (LHS) of Equation (.1) is the same as the right hand side (RHS):

$$LHS = A_1 B = RHS$$

Induction step: Let Equation (.1) hold for all $n < i$. We prove that it also holds for $n = i$. By substituting the RHS of Equation (.1) for $\phi_l$, it is sufficient to prove that:

$$\phi_i = A_i(B + \sum_{l=1}^{i-1} A_l B \Pi_{k=1}^{l-1}(1 + A_k)) = A_i B \Pi_{l=1}^{i-1}(1 + A_l)$$

as below:

$$\begin{aligned}
\phi_i &= A_i(B + \textstyle\sum_{l=1}^{i-1} A_l B \Pi_{k=1}^{l-1}(1 + A_k)) \\
&= A_i B(1 + \textstyle\sum_{l=1}^{i-1} A_l \Pi_{k=1}^{l-1}(1 + A_k)) \\
&= A_i B(1 + A_1 + \textstyle\sum_{l=2}^{i-1} A_l \Pi_{k=1}^{l-1}(1 + A_k)) \\
&= A_i B((1 + A_1) + \textstyle\sum_{l=2}^{i-1} A_l (1 + A_1) \Pi_{k=2}^{l-1}(1 + A_k)) \\
&= A_i B((1 + A_1) + A_2 + (1 + A_1) \ + \ \textstyle\sum_{l=3}^{i-1} A_l \Pi_{k=2}^{l-1}(1 + A_k)) \\
&= A_i B((1 + A_1)(1 + A_2) \ + \ \textstyle\sum_{l=3}^{i-1} A_l \Pi_{k=2}^{l-1}(1 + A_k))) \\
&= .... \\
&= A_i B((1 + A_1)(1 + A_2).....(1 + A_{i-1})) \\
&= A_i B \Pi_{l=1}^{i-1}(1 + A_l) \\
&= RHS
\end{aligned}$$

**End of Lemma 1.**

### A.1.2 Derivation of Proportional closed-form formula

Here, we use Lemma 1 to derive the formula for the Proportional speed scheduling scheme (Equation 4.3). We start from the Proportional scheme speed adjustment formula (Equation 4.1):

$$f_i = \frac{WCR/f_{max}}{d - ct - T_{total}} f_{max}$$

Let $avgc$, $ac_i$ be the average case, and actual case execution time of each execution_segment $i$, respectively, running at $f_{max}$. Recall that our assumption for the theoretical model asserts that $ac_i = avgc$. Then $ct$ equals the sum of the average execution times of the past segments divided by the segments' operating frequency normalized to the maximum frequency. Since $d$ equals $WCET/load$ then $d$ equals $n \ wcc/load$. Now, let $\alpha = avgc/wcc$. Then,

$$\begin{aligned}
f_i &= \frac{\sum_i^n wcc}{n \ wcc/load \ - \sum_{l=1}^{i-1}(avgc \ f_{max}/f_l) \ - T_{total}} f_{max} \\
&= \frac{(n-i+1)wcc}{n \ wcc/load \ - \ avgc \sum_{l=1}^{i-1}(f_{max}/f_l) \ - T_{total}} f_{max} \\
&= \frac{n-i+1}{n/load \ - \alpha \sum_{l=1}^{i-1} \phi_l \ - (T_{total}/wcc)} f_{max} \\
f_{max}/f_i &= \tfrac{1}{n-i+1} \big[ \tfrac{n}{load} - \alpha \textstyle\sum_{l=1}^{i-1} \phi_l \ - \tfrac{T_{total}}{wcc} \big] \\
\phi_i &= \tfrac{-\alpha}{n-i+1} \big[ \tfrac{1}{-\alpha} \big( \tfrac{n}{load} \ - \tfrac{T_{total}}{wcc} \big) + \textstyle\sum_{l=1}^{i-1} \phi_l \big]
\end{aligned}$$

Let $A_i = \frac{-\alpha}{n-i+1}$ and $B = \frac{1}{-\alpha}(\frac{n}{load} - \frac{T_{total}}{wcc})$. Starting from the above equation, we substitute in Lemma 2 to obtain Equation 4.3

$$\phi_i = \frac{1}{n-i+1}(\frac{n}{load} - \frac{T_{total}}{wcc})\Pi_{k=1}^{i-1}[1 - \frac{\alpha}{n-k+1}] \tag{.2}$$

## A.2 DERIVATION FOR GREEDY CLOSED-FORM EQUATION

In this section we first present Lemma 2 (and its derivation), which we use to derive the closed-form formula for Greedy speed scheduling scheme (Equation 4.4).

### A.2.1 Derivation for Lemma 2

**Lemma 2.**

$$\phi_i = i + D + C\sum_{l=1}^{i-1}\phi_l \implies \phi_i = \frac{(1+C)^i - 1}{C} + D(1+C)^{i-1} \tag{.3}$$

**Proof** (by induction):

Base case: at $i = 1$, it is trivial to see that LHS of Equation (.3) is the same as the RHS:

$$LHS = \phi_1 = 1 + D = RHS$$

Induction step: Let Equation (.3) hold for all $n < i$. We prove that it also holds for $n = i$. By substituting the RHS of Equation (.3) for $\phi_l$, it is sufficient to prove that:

$$\phi_i = i + D + C\sum_{l=1}^{i-1}\frac{(1+C)^l - 1}{C} + D(1+C)^{l-1} = \frac{(1+C)^i - 1}{C} + D(1+C)^{i-1}$$

as follows:

$$\begin{aligned}
\phi_i &= i + D + C\textstyle\sum_{l=1}^{i-1}[\frac{(1+C)^l-1}{C} + D(1+C)^{l-1}] \\
&= i + D + \textstyle\sum_{l=1}^{i-1}((1+C)^l - 1) + DC\sum_{l=1}^{i-1}(1+C)^{l-1} \\
&= i + D - (i-1) + \textstyle\sum_{l=1}^{i-1}(1+C)^l + DC\sum_{l=1}^{i-1}(1+C)^{l-1} \\
&= D + 1 + \tfrac{(1+C)^i-1}{C} - 1 + DC\tfrac{(1+C)^{i-1}-1}{C} \\
&= \tfrac{(1+C)^i-1}{C} + D(1+C)^{i-1} \\
&= RHS
\end{aligned}$$

**End of Lemma 2.**

### A.2.2 Derivation of the Greedy closed-form formula

Here, we use Lemma 2 to derive the formula for the Greedy speed scheduling scheme (Equation 4.4). We start from the Greedy scheme speed adjustment formula (Equation 4.2):

$$f_i = \frac{wcc/f_{max}}{d - ct - (WCR - wcc)/f_{max} - T_{total}} f_{max}$$

Using the same assumptions as in the Proportional scheme, we get:

$$\begin{aligned}
f_i &= \frac{wcc}{n\ wcc/load\ -\ \sum_{l=1}^{i-1} avgc(f_{max}/f_l)\ -\ (n-i)wcc - T_{total}}\ f_{max} \\
&= \frac{wcc}{(n/load\ - n + i)\ wcc\ -\ avgc \sum_{l=1}^{i-1} (f_{max}/f_l)\ - T_{total}}\ f_{max} \\
&= \frac{1}{(n/load\ - n + i)\ -\ \alpha \sum_{l=1}^{i-1} (f_{max}/f_l)\ - T_{total}/wcc}\ f_{max} \\
\phi_i &= i + (\tfrac{n}{load}\ - n - \tfrac{T_{total}}{wcc}) - \alpha \textstyle\sum_{l=1}^{i-1} \phi_l
\end{aligned}$$

Let $C = -\alpha$ and $D = \frac{n}{load} - n - \frac{T_{total}}{wcc}$. Starting from the above equation, we substitute in Lemma 2 to obtain Equation 4.4.

$$\phi_i = \frac{1 - (1-\alpha)^i}{\alpha} + (\frac{n}{load} - n - \frac{T_{total}}{wcc})(1-\alpha)^{i-1} \tag{.4}$$

# BIBLIOGRAPHY

[1] Grigorios Magklis, Greg Semeraro, David H. Albonesi, Steven G . Dropsho, Sandhya Dwarkadas and Michael L. Scott, "Dynamic Frequency and Voltage Scaling for a Multiple Clock Domain Microprocessor", *IEEE Micro*, vol. 23, n. 6, pp. 62–68, 2003.

[2] Greg Semeraro, Grigorios Magklis, Rajeev Balasubramonian, David H. Albonesi, Sandhya Dwarkadas and Michael L. Scott, "Energy-Efficient Processor Design Using Multiple Clock Domains with Dynamic Voltage and Frequency Scaling", in *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, page 29, Washington, DC, USA, 2002, IEEE Computer Society.

[3] Vincent W. Freeh, David K. Lowenthal, Feng Pan, Nandini Kappiah and Rob Springer, "Exploring the Energy-Time Tradeoff in MPI Programs on a Power-Scalable Cluster", in *IPDPD'05: Proc. International Parallel and Distributed Processing Symposium.* IEEE Computer Society, 2005.

[4] Charles Lefurgy, Karthick Rajamani, Freeman Rawson, Wes Felter, Michael Kistler and Tom W. Keller, "Energy Management for Commercial Servers", *Computer*, vol. 36, n. 12, pp. 39–48, 2003.

[5] Ozgur Celebican, Tajana Simunic Rosing and III Vincent J. Mooney, "Energy estimation of peripheral devices in embedded systems", in *GLSVLSI '04: Proceedings of the 14th ACM Great Lakes symposium on VLSI*, pp. 430–435. ACM Press, 2004.

[6] Todd Austin, David Blaauw, Scott Mahlke, Trevor Mudge, Chaitali Chakrabarti and Wayne Wolf, "Mobile Supercomputers", *Computer*, vol. 37, n. 5, pp. 81–83, 2004.

[7] Brian Davis, *Moderan DRAM Architectures*, PhD thesis, University of Michigan, Ann Arbor, 2000.

[8] Vinodh Cuppu, Bruce Jacob, Brian Davis and Trevor Mudge, "High-Performance DRAMs in Workstation Environments", *IEEE Trans. Comput.*, vol. 50, n. 11, pp. 1133–1153, 2001.

[9] Bruce Childers, H. Tang and Rami Melhem, "Adapting processor supply voltage to instruction-level parallelism", in *Koolchips 2000 Workshop, during 33th Annual International Symposium on Microarchitecture (MICRO-33)*, 2000.

[10] H. Zeng, C. Ellis, A. Lebeck and A. Vahdat, "Ecosystem: Managing energy as a first class operating system resource", 2002.

[11] Krisztian Flautner, Steve Reinhardt and Trevor Mudge, "Automatic performance setting for dynamic voltage scaling", *Wireless Networks*, vol. 8, n. 5, pp. 507–520, 2002.

[12] Krisztian Flautner and Trevor Mudge, “Vertigo: automatic performance-setting for Linux”, *SIGOPS Oper. Syst. Rev.*, vol. 36, n. SI, pp. 105–116, 2002.

[13] Daniel Mossé, Hakan Aydin, Bruce Childers and Rami Melhem, “Compiler-assisted dynamic power-aware scheduling for real-time applications”, in *COLP'00: Workshop on Compilers and Operating Systems for Low Power*, 2000.

[14] Flavius Gruian, “On energy reduction in Hard Real-Time Systems Containing Tasks with stochastic Execution times”, in *IEEE Workshop on Power Management for Real-Time and Embedded Systems*, 2001.

[15] Jacob R. Lorch and Alan Jay Smith, “Improving dynamic voltage scaling algorithms with PACE”, in *SIGMETRICS '01: Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pp. 50–61, New York, NY, USA, 2001, ACM Press.

[16] Padmanabhan Pillai and Kang G. Shin, “Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems”, in *18th ACM Symposium on Operating Systems Principles*, 2001.

[17] Hakan Aydin, Rami Melhem, Daniel Moss and Pedro Meja-Alvarez, “Determining Optimal Processor Speeds for Periodic Real-Time Tasks with Different Power Characteristics”, in *ECRTS '01: Proceedings of the 13th Euromicro Conference on Real-Time Systems*, page 225, Washington, DC, USA, 2001, IEEE Computer Society.

[18] Trevor Pering, Thomas Burd and Robert Brodersen, “Voltage scheduling in the lpARM microprocessor system”, in *ISLPED '00: Proceedings of the 2000 international symposium on Low power electronics and design*, pp. 96–101, New York, NY, USA, 2000, ACM Press.

[19] Chung-Hsing Hsu and Ulrich Kremer, “Single vs multiple regions: A comparison of different compiler-directed dynamic voltage scheduling approaches”, in *Power Aware Computer Systems (PACS)*, 2002.

[20] Chung-Hsing Hsu and Ulrich Kremer, “The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction”, in *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pp. 38–48, New York, NY, USA, 2003, ACM.

[21] A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum and A. Nicolau, “Profile-Based Dynamic Voltage Scheduling Using Program Checkpoints”, in *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, page 168, Washington, DC, USA, 2002, IEEE Computer Society.

[22] Dongkun Shin, Jihong Kim and Seongsoo Lee, “Intra-Task Voltage Scheduling for Low-Energy, Hard Real-Time Applications”, *IEEE Des. Test*, vol. 18, n. 2, pp. 20–30, 2001.

[23] H. Saputra, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, J. S. Hu, C-H. Hsu and U. Kremer, “Energy-conscious compilation based on voltage scaling”, in *LCTES/SCOPES '02: Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pp. 2–11, New York, NY, USA, 2002, ACM Press.

[24] Woonseok Kim, Dongkun Shin, Han-Saem Yun, Jihong Kim and Sang Lyul Min, "Performance Comparison of Dynamic Voltage Scaling Algorithms for Hard Real-Time Systems", in *RTAS '02: Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)*, page 219, Washington, DC, USA, 2002, IEEE Computer Society.

[25] Nam Sung Kim, Todd Austin, David Blaauw, Trevor Mudge, Krisztián Flautner, Jie S. Hu, Mary Jane Irwin, Mahmut Kandemir and Vijaykrishnan Narayanan, "Leakage Current: Moore's Law Meets Static Power", *Computer*, vol. 36, n. 12, pp. 68–75, 2003.

[26] Rajiv Ravindran, Michael Chu and Scott Mahlke, "Compiler-managed partitioned data caches for low power", in *LCTES '07: Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools*, pp. 237–247, New York, NY, USA, 2007, ACM.

[27] Ozcan Ozturk and Mahmut Kandemir, "Energy management in software-controlled multilevel memory hierarchies", in *GLSVSLI '05: Proceedings of the 15th ACM Great Lakes symposium on VLSI*, pp. 270–275, New York, NY, USA, 2005, ACM Press.

[28] G. Edward Suh, Larry Rudolph and Srinivas Devadas, "Dynamic Partitioning of Shared Cache Memory", *J. Supercomput.*, vol. 28, n. 1, pp. 7–26, 2004.

[29] Kimish Patel, Luca Benini, Enrico Macii and Massimo Poncino, "STV-Cache: a leakage energy-efficient architecture for data caches", in *GLSVLSI '06: Proceedings of the 16th ACM Great Lakes symposium on VLSI*, pp. 404–409, New York, NY, USA, 2006, ACM.

[30] Ismail Kadayif, Mahmut Kandemir and Feihui Li, "Prefetching-aware cache line turnoff for saving leakage energy", in *ASP-DAC '06: Proceedings of the 2006 conference on Asia South Pacific design automation*, pp. 182–187, Piscataway, NJ, USA, 2006, IEEE Press.

[31] Ismail Kadayif, Mahmut Kandemir and Guilin Chen, "Studying interactions between prefetching and cache line turnoff", in *ASP-DAC '05: Proceedings of the 2005 conference on Asia South Pacific design automation*, pp. 545–548, New York, NY, USA, 2005, ACM.

[32] Xiangrong Zhou and Peter Petrov, "Low-power cache organization through selective tag translation for embedded processors with virtual memory support", in *GLSVLSI '06: Proceedings of the 16th ACM Great Lakes symposium on VLSI*, pp. 398–403, New York, NY, USA, 2006, ACM.

[33] Peter Petrov and Alex Orailoglu, "Energy frugal tags in reprogrammable I-caches for application-specific embedded processors", in *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, pp. 181–186, New York, NY, USA, 2002, ACM.

[34] Mirko Loghi, Paolo Azzoni and Massimo Poncino, "Tag Overflow Buffering: An Energy-Efficient Cache Architecture", in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pp. 520–525, Washington, DC, USA, 2005, IEEE Computer Society.

[35] Greg Semeraro, David H. Albonesi, Steven G. Dropsho, Grigorios Magklis, Sandhya Dwarkadas and Michael L. Scott, "Dynamic frequency and voltage control for a multiple clock domain microarchitecture", in *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pp. 356–367, Los Alamitos, CA, USA, 2002, IEEE Computer Society Press.

[36] Anoop Iyer and Diana Marculescu, "Power and performance evaluation of globally asynchronous locally synchronous processors", in *ISCA'02: Proc Intl Symp on Computer architecture*, pp. 158–168, 2002.

[37] National Semiconductor, "PowerWise Technology", http://www.national.com/appinfo/power/powerwise.html, 2007.

[38] Juha Pennanen, "Optimizing the Power for Multiple Voltage Domains", Spring Processor Forum, Japan, 2006.

[39] G. Magklis, M.L. Scott, G. Semeraro, D.H. Albonesi and S. Dropsho, "Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor", in *ISCA'03: Proc Intl Symp on Computer Architecture*, pp. 14–27, 2003.

[40] YongKang Zhu, David H. Albonesi and A. Buyuktosunoglu, "A High Performance, Energy Efficient GALS Processor Microarchitecture with Reduced Implementation Complexity", in *ISPASS'05: Proc Intl Symp on Performance Analysis of Systems and Software*, pp. 42–53, 2005.

[41] Qiang Wu, Philo Juang, Margaret Martonosi and Douglas W. Clark, "Formal online methods for voltage/frequency control in multiple clock domain microprocessors", in *ASPLOS-XI: Proc Intl Conf on Architectural support for programming languages and operating systems*, pp. 248–259, 2004.

[42] Qiang Wu, Philo Juang, Margaret Martonosi and Douglas W. Clark, "Voltage and Frequency Control With Adaptive Reaction Time in Multiple-Clock-Domain Processors", in *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pp. 178–189, Washington, DC, USA, 2005, IEEE Computer Society.

[43] John Oliver, Ravishankar Rao, Paul Sultana, Jedidiah Crandall, Erik Czernikowski, Leslie W. Jones IV, Diana Franklin, Venkatesh Akella and Frederic T. Chong, "Synchroscalar: A Multiple Clock Domain, Power-Aware, Tile-Based Embedded Processor", in *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 150, Washington, DC, USA, 2004, IEEE Computer Society.

[44] Philo Juang, Qiang Wu, Li-Shiuan Peh, Margaret Martonosi and Douglas W. Clark, "Coordinated, distributed, formal energy management of chip multiprocessors", in *ISLPED '05: Proceedings of the 2005 international symposium on Low power electronics and design*, pp. 127–130, New York, NY, USA, 2005, ACM.

[45] Sonia Lopez, Steve Dropsho, David H. Albonesi, Oscar Garnica and Juan Lanchares, "Dynamic Capacity-Speed Tradeoffs in SMT Processor Caches", in *High Performance Embedded Architecure and Compilation (HiPEAC)*, January 2007.

[46] Rambus, "RDRAM products", http://www.rambus.com/products.

[47] Anand Ramachandran and Margarida F. Jacome, "Xtream-Fit: an energy-delay efficient data memory subsystem for embedded media processing", in *DAC '03: Proceedings of the 40th conference on Design automation*, pp. 137–142, New York, NY, USA, 2003, ACM Press.

[48] Jayaprakash Pisharath and Alok Choudhary, "An integrated approach to reducing power dissipation in memory hierarchies", in *CASES '02: Proceedings of the 2002 international*

*conference on Compilers, architecture, and synthesis for embedded systems*, pp. 88–97, New York, NY, USA, 2002, ACM Press.

[49] Xiaobo Fan, Carla Ellis and Alvin Lebeck, "Memory controller policies for DRAM power management", in *ISLPED '01: Proceedings of the 2001 international symposium on Low power electronics and design*, pp. 129–134, New York, NY, USA, 2001, ACM Press.

[50] Xiaobo Fan, Carla Ellis and Alvin Lebeck, "Modeling of DRAM Power Control Policies Using Deterministic and Stochastic Petri Nets", in *Workshop on Power-Aware Computer Systems.* Springer-Verlag, 2002.

[51] Xiaodong Li, Zhenmin Li, Francis David, Pin Zhou, Yuanyuan Zhou, Sarita Adve and Sanjeev Kumar, "Performance directed energy management for main memory and disks", *SIGOPS Opererating Syststem Review.*, vol. 38, n. 5, pp. 271–283, 2004.

[52] Le Cai and Yung-Hsiang Lu, "Joint Power Management of Memory and Disk", in *DATE'05: Design, Automation and Test in Europe.* IEEE Press, 2005.

[53] Alvin R. Lebeck, Xiaobo Fan, Heng Zeng and Carla Ellis, "Power aware page allocation", *SIGPLAN Not.*, vol. 35, n. 11, pp. 105–116, 2000.

[54] Hai Huang, Padmanabhan Pillai and Kang G. Shin, "Design and implementation of power-aware virtual memory", in *ATEC'03: Proceedings of the USENIX Annual Technical Conference 2003 on USENIX Annual Technical Conference*, pp. 5–5, Berkeley, CA, USA, 2003, USENIX Association.

[55] V. De La Luz and Mary Jane Irwin, "DRAM Energy Management Using Software and Hardware Directed Power Mode Control", in *Proc. Intl. Symp. on High-Performance Computer Architecture*, page 159. IEEE Computer Society, 2001.

[56] V. De La Luz, M. Kandemir and I. Kolcu, "Automatic data migration for reducing energy consumption in multi-bank memory systems", in *DAC '02: Proceedings of the 39th conference on Design automation*, pp. 213–218, New York, NY, USA, 2002, ACM.

[57] Ozcan Ozturk and Mahmut Kandemir, "Nonuniform Banking for Reducing Memory Energy Consumption", in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pp. 814–819, Washington, DC, USA, 2005, IEEE Computer Society.

[58] Peter Grun, Nikil Dutt and Alexandru Nicolau, "Access pattern based local memory customization for low power embedded systems", in *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, pp. 778–784, Piscataway, NJ, USA, 2001, IEEE Press.

[59] M. Kandemir and A. Choudhary, "Compiler-directed scratch pad memory hierarchy design and management", in *DAC '02: Proceedings of the 39th conference on Design automation*, pp. 628–633, New York, NY, USA, 2002, ACM Press.

[60] Doris Keitel-Schulz and Norbert Wehn, "Embedded DRAM development: Technology, physical design, and application issues", *IEEE Trans. on Design & Test of Computers*, vol. 18, n. 3, pp. 7–15, 2001.

[61] NEC embedded DRAM, http://www.necelam.com/edram90/.

[62] Steve Tomashot, “IBM embedded DRAM approach”, http://www-306.ibm.com/chips/techlib/techlib.nsf/products/Embedded_DRAM.

[63] Transmeta, “Crusoe Processor Specification”, http://www.transmeta.com.

[64] Intel developers website, “Intel XScale processors”, http://developer.intel.com/design/intelxscale.

[65] Rex Min, Travis Furrer and Anantha Chandrakasan, “Dynamic Voltage Scaling Techniques for Distributed Microsensor Networks”, in *WVLSI ’00: Proceedings of the IEEE Computer Society Annual Workshop on VLSI (WVLSI’00)*, page 43, Washington, DC, USA, 2000, IEEE Computer Society.

[66] Premkishore Shivakumar and Norman Jouppi, “CACTI 3.0: An Integrated Cache Timing, Power, and Area Model”, Technical Report 2001.2, Compaq research labs, 2001.

[67] Rambus Memories, “RDRAM Technology Summary”, http://www.rambus.com/assets/documents/products/RDRAMTechnology/Summary_091905-.pdf, 2005.

[68] Nevine AbouGhazaleh, Daniel Mosse, Bruce Childers, Rami Melhem and Matthew Craven, “Collaborative Operating System and Compiler Power Management for Real-Time Applications”, *ACM Transactions on Embedded Computing Systems (TECS)*, 2006.

[69] Alexander Vrchoticky, “Compilation Support for Fine-Grained Execution Time Analysis”, Technical report, Technical University of Vienna, 1994.

[70] Emilio Vivancos, Christopher Healy, Frank Mueller and David Whalley, “Parametric Timing Analysis”, in *LCTES ’01: Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*, pp. 88–93, New York, NY, USA, 2001, ACM.

[71] Nevine AbouGhazaleh, Bruce Childers, Daniel Mosse, Rami Melhem and Matthew Craven, “Collaborative Compiler-OS Power Management for Time-sensitive Applications”, Technical Report TR-02-103, Department of Computer Science, University of Pittsburgh, 2002.

[72] Nevine AbouGhazaleh, Bruce Childers, Daniel Mosse, Rami Melhem and Matthew Craven, “Energy management for real-time embedded applications with compiler support”, in *LCTES ’03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pp. 284–293, New York, NY, USA, 2003, ACM Press.

[73] Nevine AbouGhazaleh, Daniel Mosse, Bruce Childers, Rami Melhem and Matthew Craven, “Collaborative Operating System and Compiler Power Management for Real-Time Applications”, in *RTAS’03: IEEE Real-Time Embedded Technology and Applications Symposium*, Piscataway, NJ, USA, 2003, IEEE Press.

[74] Tohru Ishihara and Hiroto Yasuura, “Voltage scheduling problem for dynamically variable voltage processors”, in *ISLPED ’98: Proceedings of the 1998 International symposium on Low power electronics and design*, pp. 197–202, New York, NY, USA, 1998, ACM Press.

[75] Todd Austin, Eric Larson and Dan Ernst, “Simplescalar: An infrastructure for Computer System Modeling”, *IEEE Computer*, vol. 35, n. 2, pp. 59–67, February 2002.

[76] Stefanos Kaxiras, Zhigang Hu and Margaret Martonosi, "Cache decay: exploiting generational behavior to reduce cache leakage power", in *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pp. 240–251, New York, NY, USA, 2001, ACM.

[77] Krisztián Flautner, Nam Sung Kim, Steve Martin, David Blaauw and Trevor Mudge, "Drowsy caches: simple techniques for reducing leakage power", in *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pp. 148–157, Washington, DC, USA, 2002, IEEE Computer Society.

[78] Yingmin Li, Mark Hempstead, Patrick Mauro, David Brooks, Zhigang Hu and Kevin Skadron, "Power and thermal effects of SRAM vs. Latch-Mux design styles and clock gating choices", in *ISLPED '05: Proceedings of the 2005 international symposium on Low power electronics and design*, pp. 173–178, New York, NY, USA, 2005, ACM.

[79] Jason Clark and Ross Whitehead, "Low Power Server CPU Redux: Quad-Core Comes to Play", http://www.anandtech.com/printarticle.aspx?i=3095, 2007.

[80] Hakan Aydin, Rami Melhem, Daniel Mosse and P. Alvarez, "Determining Optimal Processor Speeds for Periodic Real-Time Tasks with Different Power Characteristics", in *ECRTS'01: IEEE Euromicro Conference on Real-Time Systems*, Piscataway, NJ, USA, 2001, IEEE Press.

[81] MSSG, "MPEG Software Simulation Group, MPEG2 decoder source code", http://www.mpeg.org/MPEG/MSSG.

[82] A. Hemani, T. Meincke, S. Kumar, A. Postula, T. Olsson, P. Nilsson, J. Oberg, P. Ellervee and D. Lundqvist, "Lowering power consumption in clock by using globally asynchronous locally synchronous design style", in *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*, pp. 873–878, New York, NY, USA, 1999, ACM.

[83] T. D. Burd and R. W. Brodersen, "Energy Efficient CMOS Microprocessor Design", in *Proc. of The HICSS Conference*, Jan. 1995.

[84] 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)*, 2003.

[85] Nevine AbouGhazaleh, Bruce Childers, Daniel Mosse and Rami Melhem, "Integrated CPU and Cache Power Management in Multiple Clock Domain Processors", in *HiPEAC'08: Proc. of the International Conference on High Performance Embedded Architectues & Compilers*, January 2008.

[86] Cosmin Rusu, Nevine AbouGhazaleh, Alexandre Ferreria, Ruibin Xu, Bruce Childers, Rami Melhem and Daniel Mossé, "Integrated CPU and L2 cache Frequency/Voltage Scaling using Supervised Learning", in *Workshop on Statistical and Machine learning approaches applied to ARchitectures and compilation (SMART)*, January 2007.

[87] Nevine AbouGhazaleh, Alexandre Ferreria, Cosmin Rusu, Ruibin Xu, Frank Liberato, Bruce Childers, Rami Melhem and Daniel Mossé, "Integrated CPU and L2 cache Voltage Scaling using Supervised Learning", in *LCTES'07: Proc. of ACM SIGPLAN on Language, compiler, and tool for embedded systems*, June 2007.

[88] William W. Cohen, “Fast Effective Rule Induction”, in *Proceedings of the 12th International Conference on Machine Learning*, June 1995.

[89] Johannes Furnkranz and Gerhard Widmer, “Incremental Reduced Error Pruning”, in *International Conference on Machine Learning*, pp. 70–77, 1994.

[90] SimpleScalar/ARM, “SimpleScalar-Arm Version 4.0 Test Releases”, http://www.simplescalar.com/v4test.html/.

[91] Greg Semeraro, David H. Albonesi, Steven G. Dropsho, Grigorios Magklis, Sandhya Dwarkadas and Michael L. Scott, “Dynamic frequency and voltage control for a multiple clock domain microarchitecture”, in *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pp. 356–367, Los Alamitos, CA, USA, 2002, IEEE Computer Society Press.

[92] Intel Embedded products, “High-Performance Energy-Efficient Processors for Embedded Market Segments”, http://www.intel.com/design/embedded/downloads/315336.pdf, 2006.

[93] I. H. Witten and E. Frank, *Data Mining: Practical machine learning tools and techniques*, Morgan Kaufmann, San Francisco, 2005.

[94] Johan Pouwelse, Koen Langendoen and Henk Sips, “Application-Directed Voltage Scaling”, *IEEE Transactions on Very Large Scale Integration (TVLSI)*, vol. 11, n. 5, pp. 812–826, September 2003.

[95] “Critical power slope: understanding the runtime effects of frequency scaling”, in *ICS ’02: Proceedings of the 16th international conference on Supercomputing*, pp. 35–44, New York, NY, USA, 2002, ACM.

[96] Duncan Elliott, Michael Stumm, W. Martin Snelgrove, Christian Cojocaru and Robert McKenzie, “Computational RAM: Implementing Processors in Memory”, volume 16, pp. 32–41, Los Alamitos, CA, USA, 1999, IEEE Computer Society Press.

[97] W. Hsu and J. Smith, “Performance of cached DRAM organizations in vector supercomputers”, in *Proc. Intl. Symp. on Computer Architecture*, pp. 327–336. ACM Press, 1993.

[98] I. Kadayif, T. Chinoda, M. Kandemir, N. Vijaykirsnan, M. J. Irwin and A. Sivasubramaniam, “vEC: virtual energy counters”, in *Workshop on Program analysis for software tools and engineering*, pp. 28–31. ACM Press, 2001.

[99] Nevine AbouGhazaleh, Bruce Childers, Daniel Moss and Rami Melhem, “Energy Conservation using Power-Aware Cached-DRAM”, *IEEE Transaction on Computers (TC)*, vol. 56, n. 11, pp. 1441–1455, 2007.

[100] Ram P. Koganti and Gershon Kedem, “WCDRAM: A Fully Associative Integrated Cached-DRAM with Wide Cache Lines”, Technical report, Duke University, CS dept., 1997.

[101] Ananth Hegde, N. Vijaykrishnan, Mahmut Kandemir and Mary Jane Irwin, “VL-CDRAM: variable line sized cached DRAMs”, in *CODES+ISSS ’03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pp. 132–137, New York, NY, USA, 2003, ACM.

[102] Zhao Zhang, Zhichun Zhu and Xiaodong Zhang, "Cached DRAM for ILP Processor Memory Access Latency Reduction", *IEEE Micro*, vol. 21, n. 4, pp. 22–32, 2001.

[103] Nevine AbouGhazaleh, Bruce Childers, Daniel Mossé and Rami Melhem, "Near-memory Caching for Improved Energy Consumption", in *ICCD'05: International Conference on Computer Design*, pp. 105–110, Washington, DC, USA, 2005, IEEE Computer Society.

[104] Nevine AbouGhazaleh, Bruce Childers, Daniel Mosse and Rami Melhem, "Energy Conservation in Memory Hierarchies using Power-Aware Cached-DRAM", in *Proc. of the Dagstuhl Seminar on Power-aware Computing Systems*. Dagstuhl Research Online Publication Server, April, 03–08 2005.

[105] Yutao Zhong, Steven G. Dropsho and Chen Ding, "Miss Rate Prediction across All Program Inputs", in *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, page 79, Washington, DC, USA, 2003, IEEE Computer Society.

[106] "SDRAM and RDRAM modeling for Simplescalar simulator",
http://www.tik.ee.ethz.ch/~ip3/software/simplescalar_mem_model.html.

[107] Wi fen Lin, S. Reinhardt and D. Burger, "Reducing DRAM Latencies with an Integrated Memory Hierarchy Design", in *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, page 301, Washington, DC, USA, 2001, IEEE Computer Society.

[108] Intel products website, "Pentium 4 EE processor",
http://www.intel.com/products/processor/pentium4HTXE/index.htm, 2004.

[109] Y. Aghaghiri, F. Fallah, and M. Pedram, "Transition reduction in memory buses using sector-based encoding techniques", *IEEE Trans. on Computer Aided Design*, vol. 23, n. 8, pp. 1164–1174, 2004.

[110] Nevine AbouGhazaleh, Bruce Childers, Daniel Mossé and Rami Melhem, "Energy Conservation in Memory Hierarchies using Power-Aware Cached-DRAM", Technical Report TR-05-123, Department of Computer Science, University of Pittsburgh, 2005.

[111] Intel products website, "Desktop Chipset datasheets",
http://www.intel.com/products/desktop/chipsets/- index.htm, 2006.

[112] Jud Bond, "Memory Amnesia Could Hurt Low-Power Design",
http://www.commsdesign.com/design_corner/OEG20030730S0018, 2003.