RC23313 (W0408-172) August 30, 2004
Computer Science

IBM Research Report

Cooperative Software–Hardware Power Management for DRAM

Hai Huang, Tom W. Keller, Eric Van Hensbergen, Kang Shin,
Karthick Rajamani, Charles Lefurgy, Freeman L. Rawson III
IBM Research Division
Austin Research Laboratory
11501 Burnet Road
Austin, TX 78758

Cooperative Software–Hardware Power Management for DRAM

Abstract

Energy is becoming a critical resource to not only small battery-powered devices but also large server systems, where high energy consumption translates to excessive heat dissipation, which, in turn, increases cooling costs and causes machine to become more prone to failures. DRAM is one of the most energy-consuming components in many systems. In this paper, we propose and evaluate a novel DRAM power management technique that exploits the cooperation between the system software and the memory controller hardware, in which the system software provides the memory controller with only a small amount of information about the current state of the system. This enables the memory controller to more intelligently react to the changing state in the system, and therefore, is able to make more accurate and more aggressive power management decisions. The proposed technique is evaluated against previously implemented power management techniques running synthetic, SPECjbb2000 [40] and various SPEC CPU2000 benchmarks [41]. Using SPEC benchmarks, we are able to show that the cooperative technique consumes 14.2–17.3% less energy than previously proposed hardware-only techniques, 16.0–25.8% less than software-only techniques, and 71.6–75.8% less than no power management.

1 Introduction

With semiconductor fabrication technology continuously improving and with workloads kept scaling at a similar, if not a faster, pace, hardware components are becoming faster, denser, and more highly integrated. Unfortunately, they also consume more energy. To alleviate this growing energy demand, more components are designed with power management capabilities, which enable them to operate at lower power states when not being actively used. Previous research has demonstrated that by judiciously managing power states for each of the components subject to the workload, a significant amount of energy can be saved. The reason behind such findings is that many systems are designed to be capable of providing continuous service even under certain predetermined *peak* workload. This is usually accomplished by over-allocating resources to these systems. However, when the system is operating under the *typical* load, some system resources will be under-utilized, thus creating opportunities to put certain components in low-power states or power them down. Subsequently, when the workload increases in a later time, any relevant system components can be switched back to higher-performance/power levels. Effectively, this provides performance on-demand while conserving energy during non-peak periods. However, due to non-negligible delays in transitioning between an energy-saving state and an operational state, both system performance and energy efficiency may degrade if these transitions are not controlled properly.

In this paper, we are interested in reducing power dissipated by the main memory, or DRAM. This is motivated by a continuous increase in the power budget allocated to the main memory. For example, as much as 40% of the system energy is consumed by the main memory in a mid-range IBM eServer machine [25]. Power dissipated by the memory is largely dependent on its capacity and bus frequency. Therefore, as applications become increasingly data-centric, for the performance of the system to continue to scale, we would need more power to sustain a larger-capacity and higher-performance memory system, which can easily dominate the total system energy budget.

The main contributions of this paper are summarized as follows.

- Design of a novel power management technique that enables the system software to provide the memory controller hardware with critical system-state information which was previously unavailable in the hardware level. This allows the memory controller to more intelligently react to the changing state in the system, and therefore, significantly improves the energy-performance efficiency of DRAM.

- Use of a full machine simulator (Mambo [37]) and a systematic evaluation methodology to accurately simulate the behavior of the proposed power management unit in the memory controller and its performance and energy effects on the system. Combined with a modified 2.6.5 Linux kernel, it enables us to precisely identify problems and benefits associated with the proposed cooperative management technique running various types of workload.

- In this work, we studied registered (server-grade) DRAM, which has mostly been ignored in power-related research in the past, but it is becoming more important as registered memory is almost always used in today's server-type systems.

The rest of the paper is organized as follows. Section 2 gives background information on the current state of DRAM technology and various DRAM architectures. In Section 3, we describe the detail in the proposed cooperative technique which consists of (i) Power-Aware Virtual Memory (PAVM) implemented in the OS, (ii) a thin power management layer in the memory controller hardware, and (iii) a software-hardware interface. Experimental setup and detailed evaluation are given in Section 4, where we demonstrate the significant benefit in using this new approach in terms of energy and performance. Section 5 discusses related work, and Section 6 highlights some future research directions and finally concludes the paper.

2 Memory System Model

In this section, we discuss performance and energy implications of DRAM power management. Since 1980, the performance gap between the memory and the processor has been widening continuously — DRAM has been only improving at an annual rate of 7% while processor speed has been improving at an annual rate of 40% [45]. Furthermore, frequent interactions of memory with other I/O components makes it one of the most crucial components in the overall system's performance. Unfortunately, power reduction is only possible when memory is operating at lower performance states, therefore, it is important to ensure that either this performance degradation can be hidden or that the energy saved in DRAM justifies any performance degradation and potential increase in the energy consumption of other components in the system. Before illustrating these tradeoffs between performance and energy in more detail, we will first briefly describe the basics in DRAM technology.

2.1 DRAM

DRAM core consists of large arrays of cells, each of which is a transistor-capacitor pair. To counter current leakage, each capacitor must be periodically refreshed to retain its bit information, making memory a continuous energy consumer. In reality, however, energy consumed by periodic refresh is actually very small, whereas most of the energy is consumed by row and column decoders, sense amplifiers, and external bus drivers due to large arrays with very long and high capacitance internal bus lines. To reduce power, one or more of these subcomponents need to be disabled by switching a device to one of several pre-defined low-power states when the device is not being actively accessed. However, when the device is to be accessed again, a certain performance penalty, called a *re-synchronization cost*, is incurred to transition from a low-power state to an active state by re-enabling the disabled components. This non-negligible is the cause of performance degradation when power management is not done carefully.

The above holds true for all Synchronous DRAM (SDRAM) architectures including single-data-rate (SDR), double-data-rate (DDR), and Rambus (RDRAM) architectures. In this paper, we mainly concentrate on DDR as it is becoming the most-widely used memory type. Nevertheless, our technique is architecture-independent and can be easily applied to other memory types as we will discuss in Section 4.4.2.

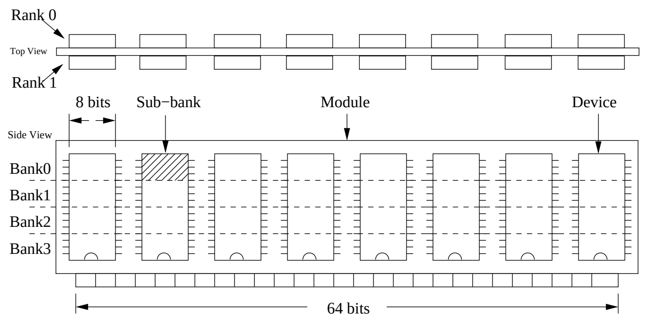
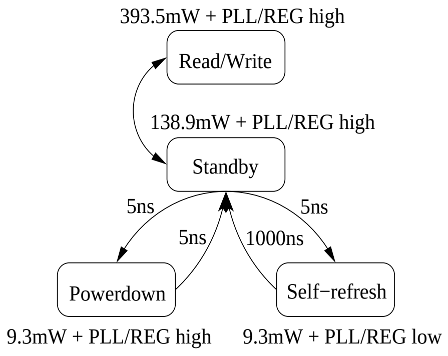


Figure 1: A memory module, or a DIMM, that is composed of 2 ranks, 8 devices per rank, and each of which is quad-banked.

2.2 Double-Data Rate DRAM Model

DDR memory is usually packaged as modules, or DIMMs, each of which contains either one or two ranks, which is commonly composed of 4, 8 or 16 number of devices. Each device is then divided into sub-banks as shown in Figure 1. Each time memory is accessed, a line size of 64 bits is read or written. Since each sub-bank can supply either 4, 8, or 16 bits at a time, it would require multiple sub-banks to act simultaneously in order to provide a line size of 64 bits on each memory access. In Figure 1, we assume that each sub-bank supplies 8 bits, and therefore, 8 devices are needed to form a complete bank. More precisely, if we assume that each device is quad-banked, 8 devices form 4 completely-independent banks, or a single rank, as shown. Even though each bank is accessed independently, we cannot manage the power of the memory in units of banks. The smallest unit we can manage power is a rank.



(a)

PLL	
Low−power	0.046 mW
High−power	810 mW

Register	
Low−power	92.6 mW
High−power	286.6 mW

(b)

Figure 2: (a) Power dissipated in each power state and the delays to transition between these states for a single 512-Mbit DDR device. (b) Power dissipation of a TI CDCVF857 PLL device (one per DIMM) and a TI SN74SSTV32867 registered buffer.

DDR architecture has many power states defined and even more possible transitions between these different power states [31, 21]. For simplicity of presentation, we only consider four of these power states — Self-Refresh, Powerdown, Standby, and Read/Write — listed in an increasing order of power dissipation. The power dissipation in these states and the transitional delays between them are shown in Figure 2(a). Note that the power numbers shown here are for a single device. Therefore, to calculate the total power dissipated by a rank, we need to multiply this power by the number of devices used per rank. We now detail each of these power states.

- **Read/Write:** Dissipates the most power, and is only briefly entered when a read/write operation is in progress.
- **Standby:** When a rank is neither reading nor writing, Standby is the highest power state, or the most-ready state, in which read and write operations can be initiated immediately at the next clock edge.
- **Powerdown:** When this state is entered, the input clock signal is gated except for the refresh signal. I/O buffers, sense amplifiers and row/column decoders are all deactivated in this state.
- **Self-refresh:** In addition to all components that are deactivated in Powerdown, the phase-lock loop (PLL) and registered buffer are also put to the low-power state to maximize energy savings as the PLL and the registered buffer (Figure 2(b)) can consume a significant portion of the total energy consumed on each DIMM. However, when exiting Self-refresh, a 1 μsec delay is needed to re-synchronize both the PLL and the registered buffer.[1]

Due to a large power differential between Standby and Powerdown / Self-refresh, we want to minimize the time a rank stays in Standby and maximize the time it spends in either Powerdown or Self-refresh. However, at the same time, we want to minimize performance degradation caused by accessing ranks that were previously put to a low-power state. Therefore, determining which ranks to power down, when to power down, and into which low-power state to transition are critically important in terms of both energy and performance. For the time being, we refer to Standby as the high-power state, and both Powerdown and Self-refresh as low-power states. In Section 4, we make the distinction between these two low-power states and illustrate how to best utilize each to maximize energy savings and minimize performance impact.

3 Design

This section details the cooperative power management paradigm. It begins with a brief design overview in Section 3.1. Hardware and software-side control mechanisms are described in Section 3.2 and Section 3.3, respectively.

3.1 Overview

Power in the memory system has traditionally been managed either solely in the hardware domain [8, 10] or in the software domain [11, 19], but not in both. However, we discovered that a small amount of cooperation between

[1]Registered memory is almost always used in server systems to better meet timing needs and provide higher data integrity, and the PLL and registered buffers are critical components to take into account when evaluating registered memory in terms of performance and energy.

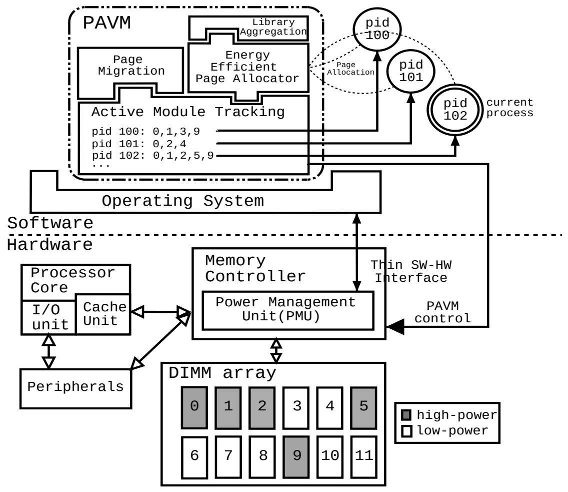


Figure 3: Architectural overview of cooperative power management system.

these two domains can lead to a significant energy benefit. In the hardware-controlled power management approach, memory traffic is monitored by the memory controller which permits implementation of a very fine-grained and highly-adaptive control mechanism, which ideally can be used to glean all possible energy-saving opportunities. However, the effectiveness of this approach is usually limited by how well the hardware can predict future references from the past access behavior. Accurate prediction is very difficult to accomplish at such a low level, especially in a complex multitasking system, where the memory can be referenced by many different processes at the same time. Any incorrect predictions will translate into both performance and energy penalties. On the other hand, in the software-controlled, or more precisely OS-controlled, approach, system and process state information (e.g., which memory regions are used by which process) can be easily tracked by the system software. This information then enables the OS to avoid performance penalty when managing the power for the memory as it can keep all ranks that may be used by the currently running process in a high-power ready state while having all other ranks in low-power states. System software alone, however, is not capable of fine-grained power control, as the OS is not generally aware of which ranks a process is *accessing* at run-time, or how actively is it accessing each rank, or whether or not there are any memory access patterns that can be exploited. It only knows which ranks are allocated to each process, thus missing many energy-saving opportunities.

Based on this observation and our discovery of a complementary relationship between these two types of approaches, we propose a cooperative power-management approach that exploits the unique features in each domain that can be used to aid the other. For example, fine-grained control mechanisms available in the hardware level can be used to aid the system software to re-capture the missed energy-saving opportunities described previously. Conversely, the system software can export useful system and process state information down to the memory controller, so that the observed memory traffic can be better interpreted at the hardware level, thus allowing the hardware to make more accurate power management decisions. Figure 3 depicts the system architecture for this cooperative power management approach. In the next section, we give details on the design of a new power management unit (PMU) in the memory controller and illustrate how it can cooperate with the system software to more intelligently manage power. We briefly describe Power-Aware Virtual Memory (PAVM) in Section 3.3. This additional software layer mostly operates orthogonally to the PMU in the memory controller, but it can provide useful information to the PMU to save additional energy and/or reduce the performance impact due to power management.

3.2 Context-Aware PMU

Memory-controller-based power management [8, 12, 13] has been previously proposed to provide fine-grained monitoring and power control, which is usually performed by a separate power management unit (PMU) implemented within the memory controller. This PMU is typically implemented as a set of simple logic devices that (i) monitor main memory accesses, (ii) predict threshold values to determine when to power down, and (iii) instruct the memory controller to perform power-down operations when certain conditions are met.

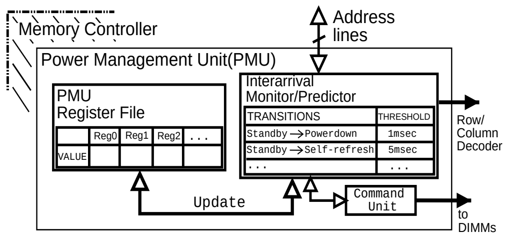
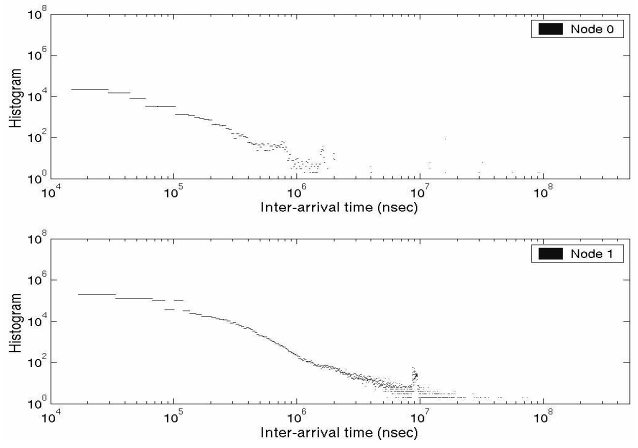


Figure 4: A simple PMU in the memory controller.

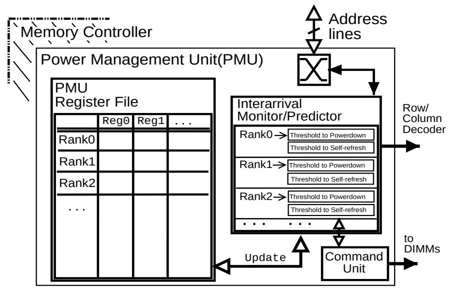
A schematic diagram of a simple PMU is shown in Figure 4. It monitors memory accesses by snooping the address lines and keeps track of the past access behavior in an internal register file, where the number of registers is dependent on how accurate we need the prediction logic to be. Based on the history, a threshold value is derived to determine how much idle time should elapse before putting a ranks in a low-power state. When multiple energy-saving states are implemented, one can derive multiple thresholds, each used to transition the rank to a different low-power state. In sections below, we propose various architectural modifications to improve upon this simple PMU design.

3.2.1 Per-Rank Power Management

The first improvement to the simple PMU design is based upon the observation that each rank is accessed differently from other ranks, and therefore, power management can be more accurately performed if we individually monitor memory accesses, keep history and control power state for each. Figure 3.2.1(a) shows a histogram (in log scale) of inter-arrival times (in log scale) between memory accesses observed on two different ranks. It is apparent from this



(a)



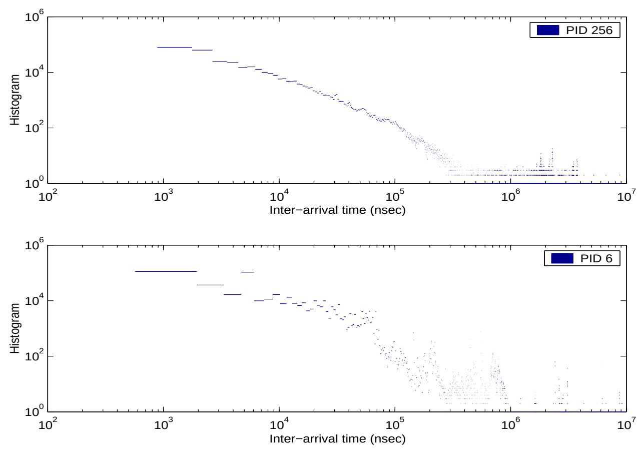
(b)

Figure 5: (a) Inter-arrival time observed on two different ranks (or nodes). (b) Architecture of a Rank-aware PMU implemented in the memory controller.

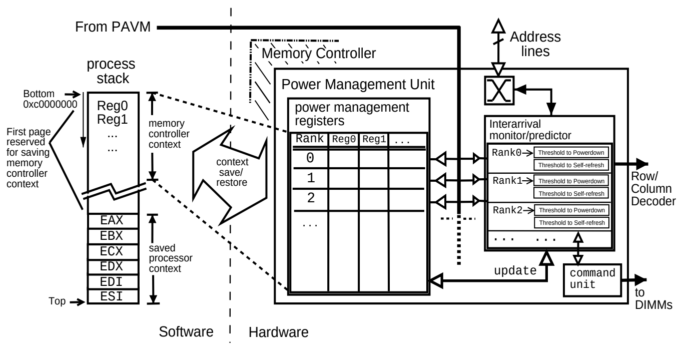
figure that the access characteristics observed on these two ranks are very different. On rank 0, we can observe that with most inter-arrival times being very short, nearly every memory access comes within 1 msec after the previous one. On rank 1, however, there are many larger gaps (indicated by a heavy-tailed distribution) between memory accesses, suggesting that we have more energy-saving opportunities and also the fact that different thresholds should be used on the two ranks to maximize energy savings on each. This *per-rank* management scheme can be easily implemented in the PMU by keeping separate monitor/predictor circuits and registers for each rank as shown in Figure 3.2.1(b). However, this would also require additional logic and storage devices which not only adds manufacturing costs but also energy costs. Later, we will show how to use the process-state information exported by the system software to amortize this additional cost.

3.2.2 Per-Process Power Management

In the previous section, we illustrated the mechanism to monitor memory traffic and manage power on a per-rank basis. Now, to take this concept a step further in enabling the controller to better interpret the monitored memory traffic, the observed per-rank memory traffic can be further partitioned on a *per-process* basis. The reason this is important is that different processes can exhibit vastly different memory access behaviors, and even for processes with similar access behaviors, how they access each individual memory rank can be quite different (Figure 6(a)) as the virtual-to-physical page mapping is controlled by the OS. So, if the PMU has no understanding of processes, the observed per-rank memory traffic is essentially “polluted” by all processes that access this rank interleavingly. Therefore, the PMU will likely make inefficient power management decisions based on an “average” access behavior observed from all of these processes. We illustrate this by an example shown in Figure 7. In this example, process 1 rarely accesses



(a)



(b)

Figure 6: (a) Inter-arrival time incurred by two different processes observed on the same memory rank. (b) Architecture of the Process-Aware PMU in the memory controller.

rank 0, whereas Process 2 accesses this rank intensively. If the controller monitors the memory traffic on this rank without differentiating between the two processes, it will conclude that this rank is accessed “moderately”, thus might make less-than-optimal power management decisions. However, by making the memory controller *context-aware*, the PMU can easily detect that process 1(*2*) rarely(*frequently*) accesses this rank, and therefore, can select more suitable thresholds depending on which process is currently executing. The problem, however, is that unlike in the case of per-rank management, the memory controller is totally oblivious to the concept of a process, which ironically, to a large extent, determines how the memory is being accessed.

This improvement to make the PMU context-aware can be easily augmented with a very small amount of hardware modifications in the PMU and would only require minor changes to the system software. For the system software, in addition to saving the processor context (i.e., registers) onto the stack of the switched-out process at each context switch, in parallel, we would also need to save the values of the history-keeping registers used by the PMU as shown in Figure 6(b) (Ignore the `PAVM line` for now). Subsequently, when this process is switched back in a later

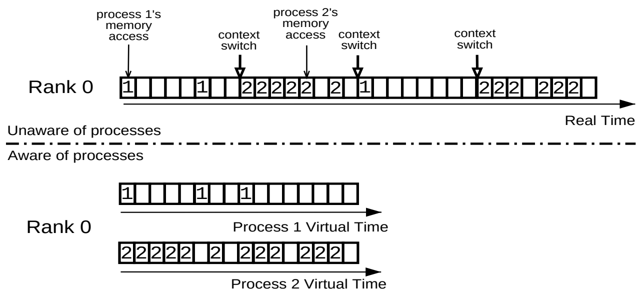


Figure 7: An example that gives some intuition on why it is beneficial to make the memory controller context-aware.

time, both the processor context and the PMU context associated with this process are restored. The PMU context saving/restoring operations can be done synchronously by the processor, or it can be done asynchronously by the memory controller when the processor sends it a context-switching signal and gives it a physical memory region for saving/restoring the PMU context. On the hardware side, only a simple I/O interface needs to be implemented for saving and restoring the PMU context. Essentially, this allows the memory controller to more efficiently manage the DRAM power for each process when it is executing as the PMU can now make power management decisions solely based upon this process's past memory access behavior.

3.3 PAVM

Power-Aware Virtual Memory (PAVM) was first proposed and implemented by Huang *et al.* in [19]. It leverages OS-level information and can make very accurate power management decisions, thus only negligibly affecting performance when performing power management. We discovered that the information collected by PAVM in the operating system can be used by the PMU to turn off unnecessary monitor/predictor circuits and reduce performance impact in PMU's power management. Its detail is provided in Section 3.3.2, but first we will give a brief overview of PAVM to see how the system software can be used to complement hardware in its power management mechanism.

3.3.1 PAVM Basics

Since all page allocation/deallocation and mapping/demapping operations are handled by the OS, where PAVM resides, PAVM knows precisely when and which ranks may be accessed by a process. This is accomplished by keeping track of a set of ranks, called the *active ranks* [19], for each individual process. Since the total number of ranks in a system is usually small, ranging from only a few in small embedded systems to up to 256 in large server systems, time and space overheads of keeping track of this information is shown to be negligible. To save energy, at each context switch, PAVM puts all *inactive ranks* of the newly-scheduled process in a low-power state. As a process can only access memory regions residing within its active ranks, powering down all other ranks will not incur any performance penalty as these ranks will not be accessed by this process. To avoid performance penalty when accessing active ranks, these ranks are put to the most-ready state at the earliest possible time during each context switch. Furthermore, an energy-efficient page allocator is implemented to effectively group allocated memory resources so that they are aggregated within a minimum number of ranks, allowing more ranks to be in low-power states without affecting performance.

In this work, due to the availability of a full-system simulator, we can run real workloads under PAVM-enabled Linux kernel and observe the memory access behavior when running these workloads. We found that a small but also a non-negligible number of memory accesses goes to ranks that are outside of the running process' set of active ranks. These were later found to be memory accesses incurred by the kernel (i.e., through system call, interrupt, exception) while in user process' context. This was resolved by tagging all pages that are used only by the kernel and aggregating them onto the first rank in the system and always keep this rank in most ready state. In our experiment, a single 64MB memory rank seems to be much more than enough to do the job.

3.3.2 PAVM-to-Hardware Interface

As indicated in Section 3.2, even though only a small amount of modifications is needed to implement the aforementioned energy-conserving mechanisms in the hardware, but the additional hardware does not come for free — a small but non-negligible additional power is dissipated. To amortize this cost, PAVM can inform the PMU which are the active ranks used by the running process so that the PMU can completely gate off all the monitor/predictor circuits and history-keeping registers for those inactive ranks without affecting the effectiveness of the power management mechanism. This information is passed down from the `PAVM control` path shown in Figure 6.

Cooperations with PAVM also have certain performance benefit. So far, we have only discussed policies and mechanisms to power down ranks but not to power them up. As premature power-ups waste energy, we currently do not consider any power-up heuristics in the PMU hardware. Instead, we rely on a simple and more accurate power-up mechanism implemented in PAVM. Since many memory accesses occur immediately after a context switch due to cold cache misses, if PAVM can instruct the memory controller to power up the active ranks of the to-be-run process as early as possible, some re-synchronization penalties can be avoided.

3.4 Summary

Through the new PMU design and the cooperation from the system software, we can partition the memory traffic — both spatially (by rank) and temporally (by process) — so that the observed memory traffic can be translated more easily and accurately by the PMU into more power-efficient management decisions. This requires only small changes in the PMU hardware and a minimal collaboration from the system software. Additionally, we have also proposed the techniques where PAVM is able to pass information down to the PMU for the purpose of (i) amortizing the energy cost of the additional hardware in the PMU and (ii) reducing wake-up latency due to cold cache misses, thus allowing more efficient use of energy.

4 Evaluation

We now evaluate the effectiveness of the proposed SW–HW power management technique and compare it against the previously proposed techniques. Section 4.1 describes the simulation environment and the methodology that we used to collect and analyze results. Section 4.2 provides detailed simulation results using both synthetic and SPEC benchmarks (SPECjbb2000 and SPEC CPU2000).

4.1 Simulation Setup

To the best of our knowledge, the proposed PMU architecture is not available in any commercial systems to date. Therefore, the best one can do is to use a machine simulator; we choose to use Mambo [37] in this project. Mambo is a full-system simulator for PowerPC machine architectures, and it was originally developed as a derivative of the PowerPC extension to SimOS [36]. Currently, it is in active use by multiple research and development efforts at IBM. It emulates both 32-bit and 64-bit PowerPC processors and also supports many system architectures and components including VMX, complex cache hierarchies, SLBs, TLBs, disks, Ethernet controllers, UART devices, etc. The simulated system is easily configurable, and very different systems can be quickly set up and simulated by simply changing a few parameters. We used a modified 2.6.5-rc3 Linux kernel, running on top of a Mambo simulated machine (parameterized as shown in Table 1), to run all our benchmarks.

Component	Parameter
Processor	64-bit 1.6GHz PowerPC
DCache	64KB 2-way Set-Associative
ICache	32KB 4-way SA
L2-Cache	1.5MB 4-way SA
DTLB	512 entries 2-way SA
ITLB	512 entries 2-way SA
DERAT	128 entries 4-deep
IERAT	128 entries 4-deep
SLB	16 entries
Memory	400Mhz 768MB(64Mbx8) DDR
Linux Kernel	2.6.5-rc3 w/ PAVM patch

Table 1: System parameters used in Mambo. All cache lines are 128 Bytes long.

To evaluate various power management techniques, we could have modified the Mambo-simulated memory controller device to study how threshold affects a system's performance and energy consumption at runtime. However, as this is an exploratory study and there is a very large solution space to search, re-running the workload in Mambo for each solution point is too computational expensive to do. Instead, we first use Mambo to record all main memory traffic (i.e., filtered by the L1 and L2 caches) into a trace file, and then feed it into a trace-driven main memory simulator to simulate various power management decisions that could have been made by the memory controller at runtime. This memory simulator not only gives power and timing information at the DRAM device level, but it can also simulate detailed activities (e.g., contention and queuing) at the memory controller, at synchronous memory interfaces and on various buses. Using this approach allows us to more thoroughly explore a large solution space with much less computing time.

4.2 Simulation Results

In Section 4.3, we use a synthetic workload consists of two streaming processes, with the first process' memory accesses all going to the cache, and the second one's all missing in the cache and going to the main memory. In Section 4.4, we evaluate and compare these power management techniques when running more complex and realistic workloads — SPECjbb2000 and SPEC CPU2000.

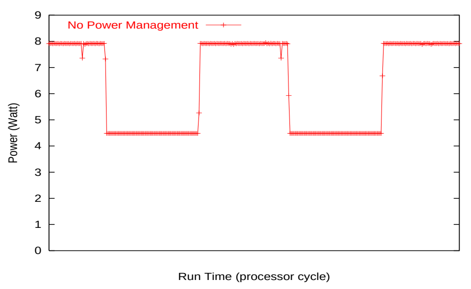
4.3 Synthetic Benchmark

4.3.1 Energy

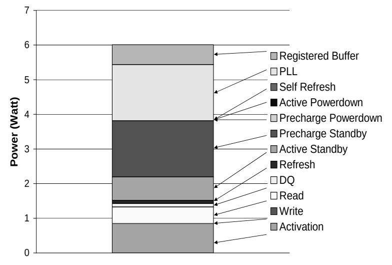
The machine configuration used for this benchmark is the same as that shown in Table 1, except that the memory capacity is reduced to only one 64MB rank. The two streaming processes are scheduled interleavingly by the Linux task scheduler. Without any power management, the instantaneous power dissipated by this rank is shown in Figure 8(a1). From this figure, one can clearly see when each process is scheduled. In Figure 8(a2), we break the average power dissipated down to various components. Power consumed by activation, read, write operations and data queues are due to DRAM devices doing useful works and cannot easily be reduced. There are previous works that studied open-page and close-page policies to reduce this type of power dissipation, however, it is not the focus of this paper. In this work, we look for ways and opportunities to reduce the idle power that is wasted when no work is done. Most of this idle power is dissipated in precharge standby mode, active standby mode, and by the PLL and the registered buffers as shown in Figure 8(a2).

First, we consider the simplest static hardware techniques, which tries to put the rank to either Power Down or Self Refresh mode immediately at the end of each memory request. Results are shown in Figures 8(b1,b2) and Figures 8(c1,c2), respectively. As we can see, power reduction opportunity arises when the low-memory referencing process starts to execute. Immediate Power Down (IPD) can significantly reduce power dissipated in Standby mode, whereas Immediate Self Refresh (ISR) can take additionally advantage of also powering down the PLL and the registered buffers. We will look at their performance implications shortly.

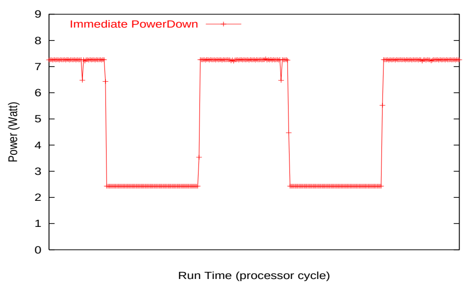
Next, Figure 8(d1) shows the results when power management decisions is purely made by the hardware (e.g., PMU in the memory controller). We assume IPD is implemented in the memory controller by default as it has a significant energy benefit and with only a very small performance impact (shown later). Additionally, past accesses



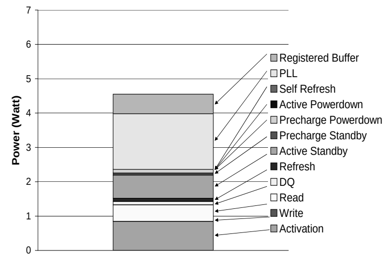
(a1)



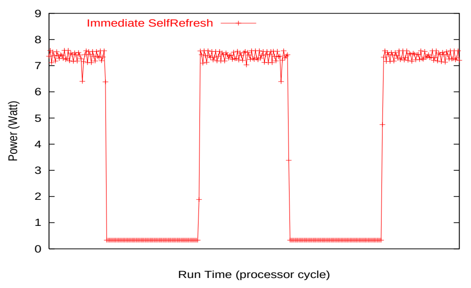
(a2)



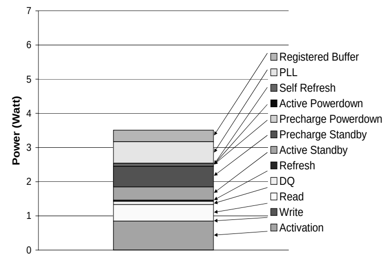
(b1)



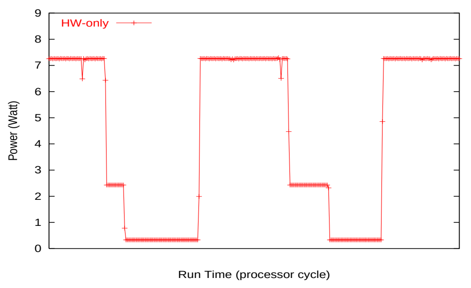
(b2)



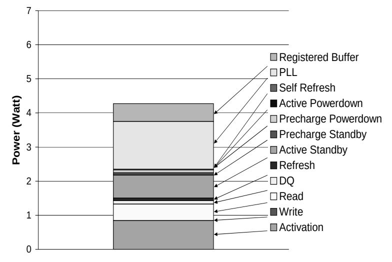
(c1)



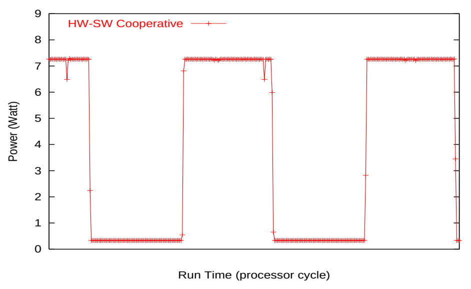
(c2)



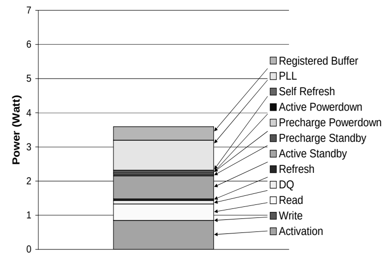
(d1)



(d2)



(e1)



(e2)

Figure 8: The fi rst column gives the instantaneous power for a zoomed-in portion of the synthetic workload under (a) no power management (b) Immediate Power Down (c) Immediate Self Refresh (d) HW-only and (d) HW-SW techniques. The second column shows the breakdown of the average power dissipated.

are kept track in the PMU's registers and then are used to dynamically predict threshold values, which are used to determine after how long of an idle period before Self Refresh mode should be entered. It uses a moving window size of 500 *µsec*, which we believe is a reasonable sized window that can both avoid over-compensation and also provide good adaptability in realistic workloads. However, the result shows that it only outperforms the IPD strategy by approximately 6% in power. The reason is that when the hardware tries to make power management decisions based upon its observation of the past memory access behavior, it gets confused when two processes with very different access behaviors interleavingly accessing the same rank. One can argue that if the window size is reduced to 50 *µsec* or even 1 *µsec*, we can adapt more quickly. However, shrinking the window size is a double-edged sword, having better adaptability runs at a higher chance to over-aggressively predict threshold values from observing transient behaviors at runtime. Shrinking the window size can benefit this synthetic workload, but for realistic workloads, it can cause more harms than benefits. Furthermore, as more and more systems are switching to smaller scheduling quantas (e.g., from 10 msec to 1 msec or even smaller) to increase responsiveness in the system, higher context switching rate will make the hardward predictor's job more difficult.

Finally, in Figure 8(e1) we show that if the system software can inform the PMU in the memory controller of which process is currently running, more aggressive and accurate power management decisions can be made. The PMU used here is exactly the same as that described above, but with additional capabilities to keep past access history for each process and to save/restore the history-keeping registers at each context switch. In this figure, we can see that immediately after the low memory intensive process starts to run, the PMU is able to instantaneously put the rank to Self Refresh, thus saving more energy. Unlike in the case of the HW-only technique, the cooperative technique will not be affected when scheduling quanta becomes smaller.

4.3.2 Performance

So far, we have only considered energy implications. Performance implication is more difficult to quantify, as it is limited by the trace-driven nature of this study. From a memory trace, we can identify exactly which memory reference is delayed and by how long due to power management. However, the memory dependency information was already lost when the memory trace was originally collected. Therefore, there is no way for us to know whether a delayed memory transaction will also delay the next memory reference. To measure performance implication, instead, we use the average response time (service time) for each memory reference. This is shown in Table 2. In this table, we also summarized the results that were presented in Figures 8(a–e).

From this table, we can see that using IPD is clearly beneficial. Compared to no power management, which has an average response time of 96.92 cycles per memory reference and consumes 10.34 J, IPD has an average response time

Total Simulated Cycles	3,442,155,784 cycles				
Number of Read	10,906,196				
Number of Writes	11,055				
	No Power Management	Immediate Power Down	Immediate Self Refresh	HW-only	HW-SW
Energy Consumption	10.34 J	7.83 J	6.04 J	7.35 J	6.18 J
Average Power	6.01 W	4.55 W	3.51 W	4.27 W	3.59 W
Average Response Time	96.92 cycles	105.04 cycles	894.01 cycles	107.20 cycles	106.81 cycles
Delayed Accesses Due to PD	0	10,486,433	0	10,391,535	10,531,756
Delayed Accesses Due to SR	0	0	603,389	16,340	8,044

Table 2: Summary of the synthetic benchmark. All cycles are in unit of processor cycles.

of 105.04 cycles (+8.4%) and consumes only 7.83 J (-24.3%). A few percent increase in the average response time not usually not a big problem for server-type workload as most are typically bandwidth-limited. On the other hand, using Immediate Self Refresh as shown in the ISR column gives additional energy benefit (6.04 J, -41.6%), but as expected it also comes with a prohibitively high response time (894.01 cycles, +822.42%). The HW-SW cooperative technique clearly shows energy benefits over the HW-only approach. Specifically, it consumes 15.9% less energy and has a slightly better average response time than the HW-only approach. In Table 2, we have also shown the number of delayed requests due to exiting Power Down (PD) and Self Refresh (SR). Exiting PD is only 5 nsec, whereas exiting SR is much more expensive — 1000 nsec. One of the reasons that the HW-SW technique consumes less energy and has lower response time than the HW-only approach is that it can more accurately predict threshold values to go to Self Refresh, and this is apparent from observing that HW-SW has much fewer number of delayed requests due to exiting from SR mode.

This synthetic benchmark is not meant to be realistic, but through this simple example, we can illustrate the potential benefit in making the memory controller context-aware. Furthermore, using this simple scenario, we can also see more clearly what are the energy and performance implications of various power management techniques. In the next section, we study these power management techniques in more detail with more realistic workloads by running some of the SPEC benchmarks.

4.4 SPEC Benchmarks

One of the benchmarks we used in our evaluation is the SPECjbb2000 [40] benchmark. It is implemented as a Java program emulating a 3-tier server system with emphasis on the middle tier. The tiers simulates a typical business application, where users in Tier 1 generate inputs that result in the execution of business logic in the middle tier (Tier 2), which calls to a database on the third tier. In a benchmark run, one can instantiate multiple warehouses, each with a 3-tier system. Each warehouse then executes as a separate Java thread within the JVM. However, since all warehouses are essentially running the same type of workload and they all share the same memory address space within the

Benchmarks	Total Runtime (processor cycles)	% of Total Runtime	Read Operations	% of All Reads	Write Operations	% of All Writes	Context Switches
Low memory intensive benchmark							
SPECjbb warehouse 1	470,662,157	4.5%	495,849	5.95%	148,964	4.67%	283
SPECjbb warehouse 2	430,865,647	4.1%	463,402	5.56%	150,847	4.73%	233
SPECjbb warehouse 3	614,658,695	5.9%	500,704	6.01%	151,581	4.75%	350
SPECjbb warehouse 4	389,326,169	3.7%	499,898	6.00%	146,077	4.58%	218
SPECjbb warehouse 5	544,571,120	5.2%	511,707	6.14%	141,688	4.44%	309
SPECjbb warehouse 6	330,170,302	3.2%	421,781	5.06%	110,106	3.45%	197
SPECjbb warehouse 7	1,694,958,880	16.3%	1,281,690	15.39%	212,097	6.65%	921
SPECjbb warehouse 8	396,145,352	3.8%	333,236	4.00%	100,222	3.14%	255
256.bzip2	2,591,125,601	24.9%	2,899,595	38.81%	1,467,012	46.00%	1,258
186.crafty	2,714,572,432	26.1%	692,731	8.32%	293,069	9.19%	1,259
Total (benchmarks)	10,177,056,355	97.7%	8,100,593	97.24%	2,921,633	91.61%	5,283
Total (all observed)	10,416,416,544	100.0%	8,330,756	100.00%	3,189,337	100%	10,148
High memory intensive benchmark							
SPECjbb warehouse 1	510,607,464	4.6%	704,477	1.29%	194,867	1.31%	734
SPECjbb warehouse 2	535,188,637	4.8%	772,954	1.41%	223,225	1.51%	478
SPECjbb warehouse 3	510,438,599	4.6%	581,688	1.06%	186,979	1.26%	465
SPECjbb warehouse 4	529,700,398	4.7%	768,019	1.40%	221,891	1.50%	420
SPECjbb warehouse 5	941,338,844	8.5%	1,167,305	2.13%	303,557	2.05%	550
SPECjbb warehouse 6	473,391,039	4.2%	776,669	1.42%	309,628	2.09%	715
SPECjbb warehouse 7	808,101,475	7.3%	1,041,908	1.90%	277,971	1.88%	508
SPECjbb warehouse 8	1,716,733,458	15.5%	2,092,407	3.82%	1,016,140	6.86%	1,379
181.mcf	2,853,500,163	25.8%	13,953,894	25.50%	7,004,631	47.26%	1,089
179.art	2,163,757,139	19.6%	32,453,738	59.31%	5,012,884	33.82%	1,089
Total (benchmarks)	11,042,757,216	99.8%	54,313,059	99.25%	14,751,773	99.53%	7,427
Total (all observed)	11,065,594,944	100%	54,721,075	100.00%	14,820,760	100.00%	12,342

Table 3: Summary of the low memory intensive and high memory intensive benchmarks. SPECjbb is ran with 8 warehouses, each spawned as a separate Java thread.

JVM, we will only observe a small amount of variation in how memory is accessed between context switches among these Java threads. In such systems, the benefit of using the HW-SW power management technique is fair limited. However, in real server systems, where the processor time is shared among multiple users, their applications, server processes, and various daemon processes, we can expect memory access behavior to change constantly when context switching between these processes at a fine granularity. To emulate such system, we decided to run a few SPEC CPU2000 benchmarks with well known execution behavior in parallel with the SPECjbb workload. We classified these benchmarks as either "high memory intensive" or "low memory intensive", based on L2 miss rates [9]. For the low memory intensive workload, we run SPECjbb having 8 warehouses in parallel with *256.bzip2* and *186.crafty*, and for the high memory intensive workload, we run SPECjbb in parallel with *181.mcf* and *179.art*.

4.4.1 Results

The runtime statistics of the two workloads are shown in Table 3. For each process in our benchmark, we kept track of the amount of CPU time it consumed, the number of read and write operations, and the number of times it was scheduled by the Linux task scheduler. We kept the system idle before we start each run. To verify that those non-benchmark processes in the system, e.g., shell, background daemons, etc, did not interfere with our runs and results, we compare the total CPU time, total number of read and write operations and total number of context switches incurred by all benchmark processes with the total number observed during the entire experimental run. For the low memory intensive workload, benchmark processes consumed 97.7% of the total CPU time, and are responsible for 97.2% of all read requests and 91.6% of all write requests in the system. For the high memory intensive workload, benchmark processes consumed 99.8% of the total CPU time, and are responsible for 99.2% of all read requests, 99.5% of all write requests in the system. The total number of context switches into the benchmark processes is significantly smaller than the total number for the entire run is because of the keyboard device driver that periodically wakes up and goes back to sleep. From these runtime statistics, we can see SPECjbb benchmark is more memory intensive than *bzip2* and *crafty*, but much less than *mcf* and *art*.

In Figure 4.4.1(a) and Figure 4.4.1(b), we show the instantaneous power dissipated during the entire run of the low memory intensive and high memory intensive workloads, respectively, for various power management techniques. Here, we do not show results for Immediate Power Down (IPD) and Immediate Self Refresh (ISR). ISR is not useful in practice due to obvious performance reasons. IPD is assumed to be implemented in the memory controller for all power management techniques that we will evaluate. Here, we compare three techniques against each other — SW-only (PAVM), HW-only, and HW-SW. The resulting power, energy, and response time are summarized in Table 4 and Table 5.

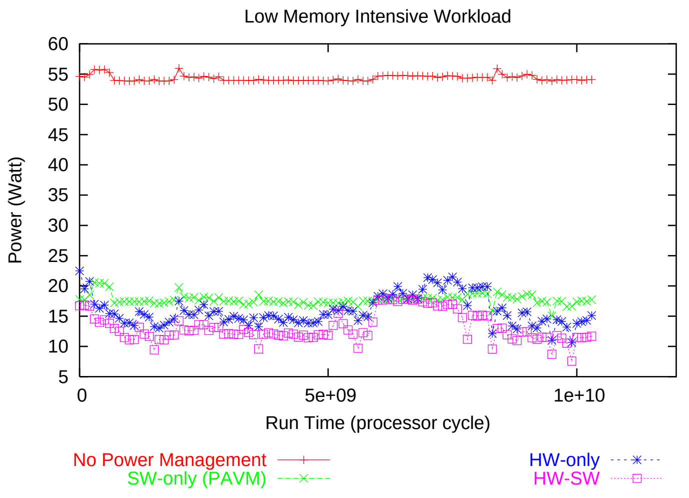
For the low memory intensive benchmark, HW-SW consumes 17.3% less energy than HW-only, and 25.8% less energy than SW-only. Furthermore, HW-SW has only a slightly higher response time (160.28 cycles) than SW-only (158.18 cycles), but it has a better response time than HW-only (161.93 cycles). For the high memory intensive benchmark, HW-SW consumes 14.2% less energy than HW-only, and 16.0% less energy than SW-only. It has a slightly higher response time (187.5 cycles) than both SW-only (179.98 cycles) and HW-only (183.17 cycles) approaches.

	No Power Management	SW-only (PAVM)	HW-only	HW-SW
Energy Consumption	282.96 J	92.407 J	82.91 J	68.61 J
Average Power	54.33 W	17.74 W	15.92 W	13.17 W
Average Response Time	136.26 cycles	158.13 cycles	161.93 cycles	160.28 cycles
Delayed Accesses Due to PD	0	5676163	6563589	5666469
Delayed Accesses Due to SR	0	1149	11842	6108

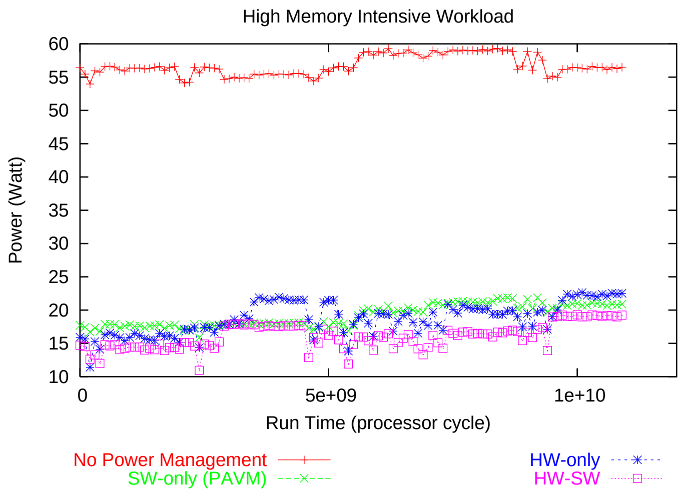
Table 4: Summary of low memory intensive benchmark.

	No Power Management	SW-only (PAVM)	HW-only	HW-SW
Energy Consumption	314.36 J	106.40 J	104.20 J	89.43 J
Average Power	56.82 W	19.23 W	18.83 W	16.16 W
Average Response Time	157.93 cycles	179.98 cycles	183.17 cycles	187.50 cycles
Delayed Accesses Due to PD	0	14,439,123	18,636,413	14,422,931
Delayed Accesses Due to SR	0	479	19,274	7,294

Table 5: Summary of high memory intensive benchmark.



(a)



(b)

Figure 9: (a) Instantaneous power for low memory intensive SPEC benchmarks. (b) Instantaneous power for high memory intensive SPEC benchmarks.

4.4.2 RDRAM

RDRAM is a new memory architecture that has emerged in the recent years. It has interesting power management features. A question one might ask is how would the result differ if RDRAM is used instead of DDR in this study. Actually, not much. It has been shown previously in [19] that the finer-grained level of control gives RDRAM a significant advantage over DDR and SDR in embedded and PC systems, where the number of power controllable memory units (i.e., DDR ranks or RDRAM devices) is small, but it has also been shown that as we increase the number of power controllable units in a system (as in the case of large server systems), there is an effect of diminishing return. Therefore, this work is directly applicable to RDRAM memory architecture, but with a slightly adjusted threshold predictor to suite RDRAM's power and performance characteristics.

5 Related Work

Recent research has demonstrated that a significant amount of energy can be saved by exploiting power management capabilities built in modern hardware components. In particular, a large body of the existing work has focused on reducing processor energy consumption. Weiser *et al.* [44] first demonstrated the effectiveness of using Dynamic Voltage Scaling (DVS) to reduce power dissipation in processors. Later work [32, 3, 15, 17, 18, 28, 33, 35, 34] further explored the effectiveness of DVS techniques in both real-time and general-purpose systems.

There is also a large body of work that focused on reducing power in other system components, including wireless communication [42, 20, 23, 14], disk drives [26, 6, 5, 24], flash devices [4, 30], cache [1, 22, 43], and main memory [8, 12, 13, 10, 11, 19], while some others [16, 46, 29, 39] explored system-level approaches to extend/target the battery lifetime of the system, as opposed to simply save energy for individual components.

Among power management techniques for DRAM, there are two main types of approaches — hardware and software-controlled. Among the hardware-controlled approaches, Lebeck *et al.* [8, 12] studied the effects of various static and dynamic memory controller policies to reduce power with extensive simulation in a single-process environment. In another paper [13], they used stochastic Petri Nets to explore more complex policies. Delaluz *et al.* took a similar approach in [10], where they studied various flavors of threshold predictors and evaluated their energy implications. The techniques proposed in this paper is orthogonal to the works described above and can be used to improve the accuracy in some of these previously proposed threshold prediction mechanisms. However, unlike previous works, the techniques proposed in this paper are especially designed and optimized for a multitasking environment, which most of today's systems are. Furthermore, we have also taken into account of various OS effects, which were shown to be also important in practice [19].

Among the software-controlled approaches, Delaluz *et al.*[11] demonstrated a simple scheduler-based power management policy. Huang *et al.* [19] later implemented Power-Aware Virtual Memory (PAVM) to improve upon this work. PAVM modifies the underlying physical page allocator to make it more energy-efficient by collaborating with the VM through a NUMA management layer so that the energy footprint of each process is reduced. To cope with various dynamics in real systems, PAVM leverages advanced techniques, such as library aggregation and page migration. Delaluz *et al.*[10] have also proposed a compiler-directed approach, where power management decisions are statically determined. Due to its static nature, this approach is not very appropriate for most complex systems, but may be applicable in some embedded systems where workloads are more deterministic.

There are advantages and disadvantages in the two types of approaches. The cooperative technique that we proposed in this paper offers the best features in both. With minimal help from the system software, we were able to show that the PMU in the memory controller can more accurately monitor memory traffic and thus more efficiently managing power. In other research contexts, using software and hardware collaboration [38, 2, 7, 27] has also been shown to be beneficial in terms of improving performance and security, and providing new functionalities. Additionally, we were able to show PAVM is able to provide other useful information to the PMU such that more energy can be conserved without affecting the effectiveness of the power management mechanism.

6 Conclusion

In this paper, we have proposed a novel power management technique that makes use of cooperation between the system software and the memory controller hardware so that the energy is more efficiently utilized by the main memory. By exporting a small amount of information from the system software to the memory controller, we are able to demonstrate a significant improvement in the accuracy of the PMU's threshold prediction logic, thus saving more energy. Using a full-system simulator, we have shown that the cooperative approach consumes 14.2–17.3% less energy than the hardware-only technique and 16.0–25.8% less energy than the software-only technique.

In this work, we have only explored software-assisted hardware power management techniques. Vice versa, we can imagine scenarios where hardware can provide feedbacks to the system software to create additional energy saving opportunities. For example, the hardware can inform the OS how "hot" each physical page is being accessed, and the OS can use this information to rearrange memory pages with each process' address space. This allows us to either (i) balance power dissipation on each memory rank, or (2) run hot ranks hotter and cold ranks colder to create more power saving opportunities on the cold ranks. Additionally, we would also like to explore direct cooperation between applications and the PMU. As applications themselves know more precisely about their future memory access behavior than the OS, such information can prove to be beneficial to the memory controller in its prediction logic, thus can be used to further enhance the proposed cooperative HW-SW system.

References

[1] R. Bahar, G. Albera, and S. Manne. Power and performance tradeoffs using various caching strategies. In *International Symposium on Low Power Electronic Design (ISLPED)*, pages 64–69, 1998.

[2] Edouard Bugnion and *et al.* Compiler-directed page coloring for multiprocessors. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1996.

[3] T. D. Burd and R. W. Brodersen. Energy effi cient CMOS microprocessor design. In *Proceedings of the 28th Annual Hawaii International Conference on System Sciences. Volume 1: Architecture*, pages 288–297. IEEE Computer Society Press, 1995.

[4] F. Douglis, R. Caceres, M. F. Kaashoek, K. Li, B. Marsh, and J. A. Tauber. Storage alternatives for mobile computers. In *Operating Systems Design and Implementation (OSDI)*, pages 25–37, 1994.

[5] F. Douglis, P. Krishnan, and B. Bershad. Adaptive disk spin-down policies for mobile computers. In *Proc. 2nd USENIX Symp. on Mobile and Location-Independent Computing*, 1995.

[6] F. Douglis, P. Krishnan, and B. Marsh. Thwarting the power-hungry disk. In *USENIX Winter*, pages 292–306, 1994.

[7] D. Engler, M. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *15th ACM Symposium on Operating Systems Principles (SOSP)*, 1995.

[8] A. R. Lebeck *et al.* Power aware page allocation. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 105–116, 2000.

[9] Karthikeyan Sankaralingam *et. al.* Exploiting ilp, tlp, and dlp with the polymorphous trips architecture. In *ISCA*, 2003.

[10] V. Delaluz *et al.* DRAM energy management using software and hardware directed power mode control. In *International Symposium on High-Performance Computer Architecture*, pages 159–170, 2001.

[11] V. Delaluz *et al.* Scheduler-based DRAM energy power management. In *Design Automation Conference 39*, pages 697–702, 2002.

[12] X. Fan, C. S. Ellis, and A. R. Lebeck. Memory controller policies for DRAM power management. In *International Symposium on Low Power Electronics and Design (ISLPED)*, pages 129–134, 2001.

[13] X. Fan, C. S. Ellis, and A. R. Lebeck. Modeling of DRAM power control policies using deterministic and stochastic petri nets. In *Workshop on Power-Aware Computer Systems*, 2002.

[14] L. M. Feeney and M. Nilsson. Investigating the energy consumption of a wireless network interface in an ad hoc networking environment. In *IEEE INFOCOM*, 2001.

[15] K. Flautner, S. Reinhardt, and T. Mudge. Automatic performance-setting for dynamic voltage scaling. In *Proceedings of the 7th Conference on Mobile Computing and Networking (MOBICOM)*, pages 260–271, 2001.

[16] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 48–63, 1999.

[17] K. Govil, E. Chan, and H. Wassermann. Comparing algorithms for dynamic speed-setting of a low-power CPU. In *Proceedings of the 1st Conference on Mobile Computing and Networking (MOBICOM)*, 1995.

[18] F. Gruian. Hard real-time scheduling for low energy using stochastic data and DVS processors. In *Proceedings of the International Symposium on Low-Power Electronics and Design (ISLPED)*, 2001.

[19] H. Huang, P. Pillai, and K. G. Shin. Design and implementation of power-aware virtual memory. In *USENIX Annual Technical Conference*, pages 57–70, 2003.

[20] C. E. Jones, K. M. Sivalingam, P. Agrawal, and J. Chen. A survey of energy effi cient network protocols for wireless networks. *Wireless Networks*, 7(4):343–358, 2001.

[21] Y. Joo and *el al*. Energy exploration and reduction of SDRAM memory systems. In *DAC*, pages 892–897, 2003.

[22] M. Kamble and K. Ghose. Energy-effi ciency of VLSI caches: A comparative study. In *Proc. of International Conference on VLSI Design*, 1997.

[23] R. Kravets and P. Krishnan. Power management techniques for mobile communications. In *Proceedings of the 4th Conference on Mobile Computing and Networking (MOBICOM)*, 1998.

[24] P. Krishnan, P. Long, and J. Vitter. Adaptive disk spin-down via optimal rent-to-buy in probabilistic environments. In *Proc. of International Conference on Machine Learning*, pages 322–330, 1995.

[25] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and Tom Keller. Energy management for commercial servers. In *IEEE Computer*, pages 39–48, Dec 2003.

[26] K. Li, R. Kumpf, P. Horton, and T. E. Anderson. A quantitative analysis of disk drive power management in portable computers. In *USENIX Winter*, pages 279–291, 1994.

[27] David Lie, Chandramohan A. Thekkath, and Mark Horowitz. Implementing an untrusted operating system on trusted hardware. In *19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[28] J. Lorch and A. J. Smith. Improving dynamic voltage scaling algorithms with PACE. In *Proceedings of the ACM SIGMETRICS 2001 Conference*, pages 50–61, 2001.

[29] Y. H. Lu, L. Benini, and G. De Micheli. Operating-system directed power reduction. In *International Symposium on Low Power Electronics and Design (ISLPED)*, pages 37–42, 2000.

[30] B. Marsh, F. Douglis, and P. Krishnan. Flash memory fi le caching for mobile computers. In *Proceedings of the 27th Hawaii Conference on Systems Science*, 1994.

[31] Micron. http://www.micron.com.

[32] D. Mosse, H. Aydin, B. Childers, and R. Melhem. Compiler-assisted dynamic power-aware scheduling for real-time applications. In *Workshop on Compilers and Operating Systems for Low-Power*, 2000.

[33] T. Pering, T. Burd, and R. Brodersen. Voltage scheduling in the lpARM microprocessor system. In *Proceedings of the International Symposium on Low-Power Electronics and Design (ISLPED)*, 2000.

[34] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 89–102, 2001.

[35] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic voltage scaling on a low-power microprocessor. In *Proceedings of the 7th Conference on Mobile Computing and Networking (MOBICOM)*, pages 251–259, 2001.

[36] M. Rosenblum, E. Bugnion, S. Devine, and S. Herrod. Using the SimOS machine simulator to study complex computer systems. In *ACM Transaction on Modeling and Computer Simulation*, volume 7, pages 78–103, 1997.

[37] H. Shafi , P. J. Bohrer, J. Phelan, C. A. Rusu, and J. L. Peterson. Design and validation of a performance and power simulator for PowerPC systems. In *IBM Journal on Research and Development*, volume 47, 2003.

[38] T. Sherwood, B. Calder, and J. Emer. Reducing cache misses using hardware and software page placement. In *ACM International Conference on Supercomputing*, pages 155–164, 1999.

[39] T. Simunic, L. Benini, P. Glynn, and G. De Micheli. Dynamic power management for portable systems. In *International Conference on Mobile Computing and Networking*, pages 11–19, 2000.

[40] Standard Performance Evaluation Corporation (SPEC). http://www.specbench.org/jbb2000/.

[41] Standard Performance Evaluation Corporation (SPEC). http://www.specbench.org/osg/cpu2000/.

[42] M. Stemm and R. H. Katz. Measuring and reducing energy consumption of network interfaces in hand-held devices. *IEICE Transactions on Communications, vol.E80-B, no.8, p. 1125-31*, E80-B(8):1125–31, 1997.

[43] C. Su and A. Despain. Cache design tradeoffs for power and performance optimization: A case study. In *International Symposium on Low Power Electronic Design (ISLPED)*, pages 63–68, 1995.

[44] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)*, pages 13–23, 1994.

[45] W. Wulf and S. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, 1995.

[46] H. Zeng, X. Fan, C. Ellis, A. Lebeck, and A. Vahdat. Ecosystem: Managing energy as a first class operating system resource. In *International Conference on Archtectural Support for Programming Languages and Operating Systems*, 2002.