Cooperative Software–Hardware Power Management for Main Memory *

Hai Huang, Kang G. Shin
The University of Michigan
{*haih,kgshin*}@*eecs.umich.edu*

Charles Lefurgy, Karthick Rajamani, Tom Keller, Eric V. Hensbergen, Freeman Rawson
IBM Austin Research Laboratory
{*lefurgy,karthick,tkeller,bergevan,frawson*}@*us.ibm.com*

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 servers to become more prone to failure. Main memory is one of the most energy-consuming components in many systems. In this paper, we propose and evaluate a novel power management technique, in which the system software provides the memory controller with a small amount of information about the current state of the system, which is used by the memory controller to significantly reduce power. Our technique enables the memory controller to more intelligently react to the changing state in the system, and therefore, be 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 [35] and various SPECcpu2000 [36] benchmarks. Using SPEC benchmarks, we are able to show that the cooperative technique consumes 14.2–17.3% less energy than the previously-proposed hardware-only technique, 16.0–25.8% less than the software-only technique, and 71.6–75.8% less than no power management.

Keywords

DDR, low power, memory system, software-hardware cooperation, system evaluation

1. INTRODUCTION

With semiconductor fabrication technology continuously improving and with workloads 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 for such findings is that many systems are designed to be capable of providing continuous service even when they are being stressed at their predetermined peak load. This is usually accomplished by over-allocating resources to these systems. However, when the system is operating at a *typical* load, some system resources will be under-utilized, thus creating opportunities to put certain components in low-power states, or even power them down. Subsequently, when the workload increases at a later time, any relevant system components will 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 the system performance and energy efficiency may degrade if these transitions are not controlled properly.

This paper focuses on reducing power dissipated by the main memory system (consists of DRAM). This is motivated by a continuous increase in the power budget allocated to the memory. For example, as much as 40% of the system energy is consumed by the memory system in a mid-range IBM eServer machine [22]. Power dissipated by the DRAM 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 cooperate with the memory controller hardware by providing it with critical system-state information which was previously unavailable at the hardware level. This allows the memory controller to more intelligently react to the changing state in the system, and therefore, significantly improve the energy-performance efficiency of main memory.
- Use of a full system simulator (Mambo [9]) 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. Using 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.
- Evaluation of registered DRAM (server-grade), which has been mostly under-explored in the past, but it is now becoming increasingly important as it is almost always used in today's server systems. Using registered DRAM, we demonstrate that our cooperative technique can save 14.2–17.3% more energy than previously-proposed hardware-only techniques, 16.0–25.8% more than software-only techniques, and 71.6–75.8% more than no power management.

The rest of the paper is organized as follows. Section 2 provides background information on the current state of DRAM technology and various DRAM architectures. Section 3 describes 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

*The work reported in this paper was supported in part by the US Air Force Office of Scientific Research under Grant AFOSR F49620-01-1-0120 and also by DARPA under Contract No. NBCH3039004..

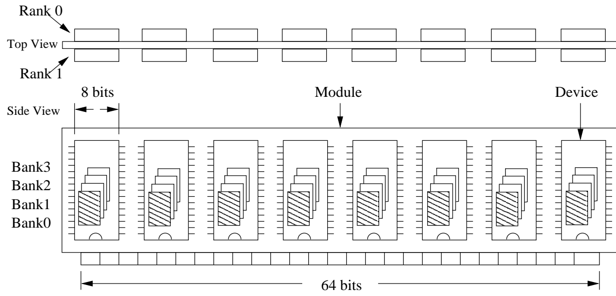

Figure 1: **A memory module, or a DIMM, that is composed of 2 ranks, 8 devices per rank, and each of which is quad-banked.**

evaluation are given in Section 4, demonstrating a 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 when power is managed for the main memory. Since 1980, the performance gap between the memory and the processor has been widening continuously — DRAM speed has been only improving at an annual rate of 7% while processor speed has been improving at an annual rate of 40% [39]. Furthermore, frequent interaction between memory and other I/O components makes it a crucial component in the overall performance of the system. Unfortunately, power reduction is only possible when memory is operating at lower performance states, and therefore, it is important to ensure that either this performance degradation can be hidden or that the energy saved in the memory justifies any performance degradation that it causes. Before illustrating the tradeoffs between performance and energy, we will first briefly describe the basics in DRAM technology.

2.1 DRAM

A DRAM core consists of a 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 state, making memory a continuous energy consumer. In reality, however, energy consumed by periodic refresh is actually very small, and 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 are disabled by switching a device to one of several pre-defined low-power states when it 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 the current low-power state to an active state before it can be accessed. This non-negligible delay 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 4.3.2.

2.2 Double-Data Rate DRAM Model

DDR memory is usually packaged as modules, or DIMMs, each of which usually contains either 1, 2 or 4 *ranks*, which are commonly composed of 4, 8 or 16 number of physical devices (shown in Figure 1). Each time a DIMM is accessed, 64 bits of data is read or written. Since each device, depending on design, can supply either 4, 8, or 16 bits at a time, multiple devices are needed to act simultaneously to satisfy a 64-bit DIMM access, and these devices constitute a rank. A rank is then divided into multiple banks (logical devices, usually 4 or 8), each of which may be accessed individually, but cannot be power managed separately. The smallest physical unit for which we can independently manage power is a single rank.

DDR architecture has many power states defined and even more possible transitions between them [30, 18]. For simplicity of presentation, we only consider four of these power states — Read/Write, Standby, Powerdown, and Self Refresh — listed in a decreasing order of power dissipation. The power dissipation in each state 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. For a 512MB registered DIMM consisting of 8 devices in a rank, the expected power draw values are 4.2 W, 2.2 W, 1.2 W, and 0.167 W, respectively, for the four power states considered here. The details of these power states are as follows:

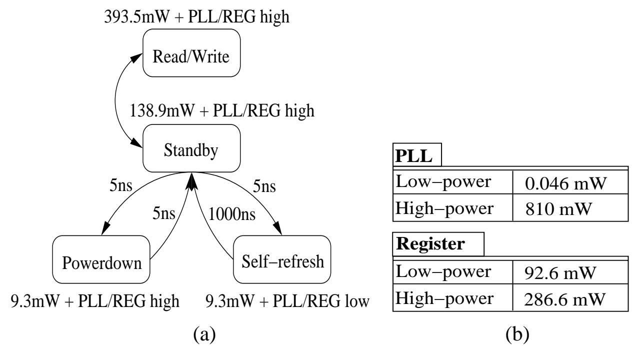

Figure 2: **(a) Power dissipated in each power state and the delays to transition between these states for a single 512-Mbit DDR device. (b) Power dissipation of a TI CDCVF857 PLL device (one per DIMM) and a TI SN74SSTV32867 register.**

- **Read/Write:** Dissipates the most power, and it 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 the components on a DIMM that are deactivated in Powerdown, the phase-lock loop (PLL) device and registers are also put to the low-power state to maximize energy savings as the PLL and the registers (Figure 2(b)) can consume a significant portion of the total energy on each DIMM. However, when exiting Self Refresh, a 1 μsec delay is needed to re-synchronize both the PLL and the registers.[1]

[1]Registered memory is almost always used in server systems to

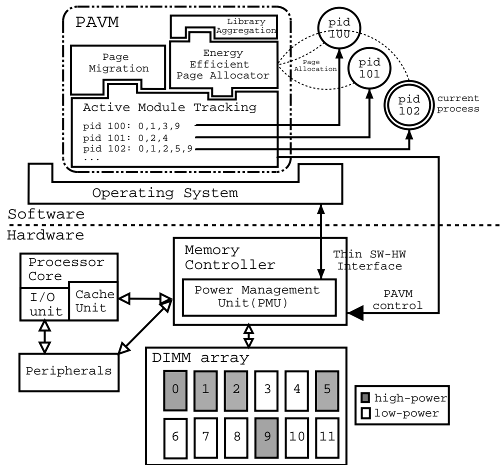

Figure 3: **Architectural overview of cooperative power management system.**

Due to having 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 also want to minimize performance degradation caused by accessing ranks that were previously put to one of the low-power states. Therefore, determining which ranks to power down, when to power down, and into which low-power state to transition are critically important to 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. We make the distinction between these two low-power states in Section 4 and illustrate how to best utilize each to maximize energy savings while minimizing performance impact.

3. DESIGN

This section details the design of the cooperative power management technique. 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

Proposals to manage power in the memory system have traditionally operated solely in the hardware domain [7, 10] or in the software domain [11, 16], but not in both. In our work, we discovered that a small amount of cooperation between 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 access patterns constantly change due to interleaved execution of many different processes. Any incorrect prediction 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 current running process in a high-power ready state while having all other ranks in low-power states. However, system software alone is not capable of achieving fine-grained power control, as the OS is not generally aware of which ranks a process is *accessing* at run-time, or how actively it is accessing a rank, or whether or not there are any memory access patterns that can be exploited. It only knows about the active ranks of the running process. However, since some of the active ranks are infrequently used, and due to its inability of exploiting such knowledge, many energy-saving opportunities will be lost in using this software-only approach.

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 available 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 recapture some of the missed energy-saving opportunities described earlier. 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 of this cooperative power management approach showing both the software and hardware components. In the next section, we first describe the architecture of the power management unit (PMU) in the memory controller. It is the hardware component responsible for monitoring memory traffic and controlling power in the DRAM. We then describe how to minimally modify this PMU so it can efficiently communicate with the system software to gain information about the current state of the system, and thus allowing it to more intelligently manage power. In Section 3.3, we describe what system and process state information are useful to the PMU and how does the system software convey this information to the PMU.

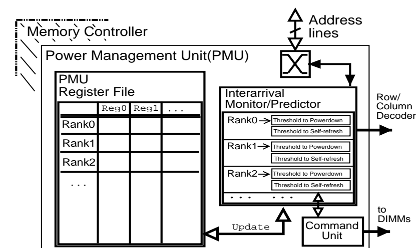

Figure 4: **Architecture of a per-rank PMU implemented in the memory controller.**

3.2 Context-Aware PMU

Memory-controller-based power management [7, 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.

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

better meet timing needs and provide higher data integrity, and the PLL and registers are critical components to take into account when evaluating registered memory in terms of performance and energy.

prediction logic to be. Based on the history, a threshold value is derived to determine how much idle time should elapse before putting a rank into 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.

Separate monitor/predictor logic is often kept for each of the ranks so the PMU can individually monitor memory accesses, keep history and control power state for each. The reason for keeping a separate set of logics is because each rank may be accessed very differently from all other ranks. To give an example, Figure 5 shows a histogram (in log scale) of inter-arrival times (in log scale) between consecutive memory accesses observed on two different ranks. It is apparent from this 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 heavier-tailed distribution) between memory accesses, suggesting that we have more energy-saving opportunities and also the fact that different thresholds should be used on these two ranks to maximize energy savings on each. However, this *per-rank* implementation in the PMU would require additional circuitry which not only adds manufacturing costs but also additional energy costs. Later, we will show how to use the process state information exported by the system software to reduce this additional cost.

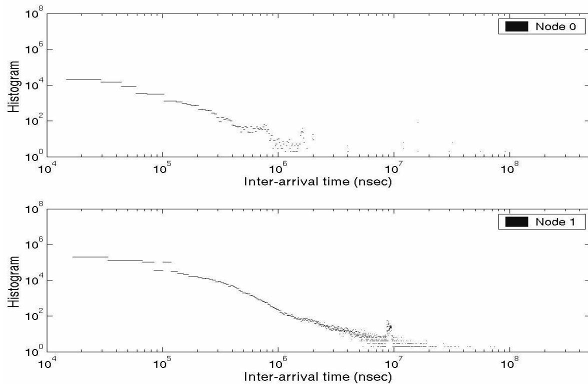

Figure 5: Inter-arrival time observed on two different ranks (or nodes).

3.2.1 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, we further partition the observed per-rank memory traffic on a *per-process* basis. The reason why this is important is that different processes can exhibit vastly different memory access behaviors. Even for processes with similar access behaviors, how they access each individual rank can be quite different (Figure 6(a)) given that the virtual-to-physical page mapping is controlled arbitrarily 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 in rapid successions (at a 10 msec or even an 1 msec quantum) as scheduled by the task scheduler. Therefore, the PMU will likely make inefficient power management decisions based on this "average" access behavior observed from all the concurrent processes. We illustrate this by an example shown in Figure 7. In this example (top portion of the graph), Process 1 rarely accesses rank 0, whereas Process 2 accesses this rank very frequently. 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", and thus, might make less-than-optimal power management decisions. However, by making the memory controller *context-aware* (bottom portion of the graph), 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 strongly impacts how the memory is being accessed and how it should be controlled.

The improvement to make the PMU context-aware can actually be very easily augmented with a small amount of hardware modifications in the PMU and some minor changes to the system software. On the software side, in addition to saving the processor context (i.e., CPU 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 at a later time, both the processor context and the PMU context associated with this process are restored. The PMU context saving/restoring operations can either be done synchronously by the processor, or asynchronously by the PMU itself 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 power for the main memory tailored to the memory access behavior specific to each process because the PMU can now make power management decisions solely based upon each process's past memory access behavior.

3.3 PAVM

Power Aware Virtual Memory (PAVM) was first proposed and implemented by Huang *et al.* in [16]. 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 make more accurate power management decisions and to determine which monitor/predictor circuits in the PMU are unnecessary so it can turn them off to further reduce power. The HW–SW interface is described in Section 3.3.2, but first we will give a brief overview of PAVM in the next section.

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

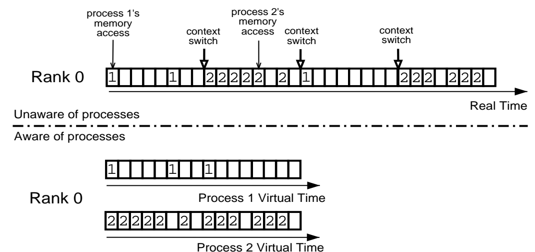

Figure 7: An example that gives some intuition on why it is beneficial to make the memory controller context-aware.

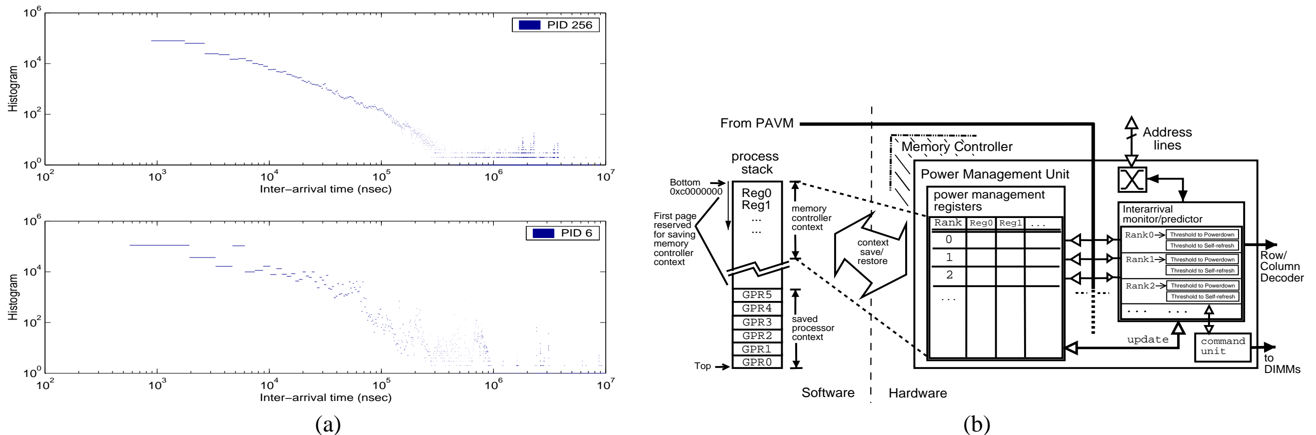

Figure 6: (a) Inter-arrival time incurred by two different processes observed on the same memory rank. (b) Architecture of the Process-Aware PMU in the memory controller.

the *active ranks* [16], for each individual process. An active rank of a process is defined to have at least one page mapped from this process's address space. Since the total number of ranks in a system is usually small, ranging from a few in small embedded systems to a couple hundred 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, with the availability of a full-system simulator, we are able to run real workloads under PAVM-enabled Linux kernel and observe the memory access behavior at run-time. This allows us to find several problems with the original PAVM implementation. We found that a small but a non-negligible number of memory accesses did not go to the active ranks of the current running process. These were later found to be memory accesses incurred by the kernel (i.e., through system call, interrupt, exception) while in user process's 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 keeping this rank in the most-ready state to reduce performance impact. In our experiment, a single 64MB memory rank seems to have more than enough capacity for such purpose.

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 a non-negligible additional power is dissipated. To amortize this cost, PAVM can inform the PMU which ranks are 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` line shown in Figure 6(b).

Cooperations with PAVM also have certain performance benefits. 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 hardware. Instead, we rely on a simple but 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 with the cooperation from the system software, we can partition the observed 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 also proposed techniques that allow PAVM 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 the energy.

4. EVALUATION

We now evaluate the effectiveness of the proposed cooperative HW–SW power management technique and compare it against some previously-proposed techniques. Section 4.1 describes the simulation environment and the methodology that we have used to collect and analyze results. Section 4.2 and Section 4.3 provide detailed simulation results using synthetic and SPEC benchmarks (SPECjbb2000 and SPECcpu2000), respectively.

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 [32] in this project. Mambo is a full-system simulator for PowerPC® machine architectures and 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 a multi-tiered cache hierarchy, SLBs, TLBs, disks, Ethernet controllers, UART devices, etc. The simulated system is easily configurable, and very different sys-

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

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

tems 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 workloads.

To evaluate various power-management techniques, we first use Mambo to record all the 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 is written using CSIM [28] library, and it can simulate detailed activities in memory devices, controllers, synchronous memory interfaces (SMIs) and on various buses. Instantaneous power is calculated using the method described in [29]. We keep track of state information for each bank on a per-cycle basis, which gives us power and performance information.

4.2 Synthetic Benchmark

We first use a synthetic benchmark consisting of two streaming processes. The first process's memory accesses all miss in the cache and go to the main memory, and the second process's all hit in the cache. 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 in using various power management techniques. In the following section, we evaluate and compare these power management techniques with more realistic workloads — SPECjbb2000 and SPECcpu2000.

4.2.1 Power Management Techniques

The machine configuration used for this benchmark is the same as that shown in Table 1, except that the memory capacity is reduced to a single 64MB rank. The two streaming processes are scheduled in an interleaved-manner by the Linux task scheduler. Without any power management, the instantaneous power dissipated by the memory is shown in Figure 8(a1), where one can clearly see when each process is scheduled. In Figure 8(a2), we break the average power dissipated for this benchmark down to various components. Power used by activation, read, write operations and data queues are due to DRAM devices doing useful work and cannot be reduced by using power management. Here, 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 the Precharge Standby mode, Active Standby mode, and by the PLL and the registers.

First, we consider the simplest static hardware techniques, which try to put the rank to either Power Down or Self Refresh mode immediately at the end of each memory request. We call them Immediate Power Down and Immediate Self Refresh, respectively, and the results are shown in Figures 8(b1-b2) and Figures 8(c1-c2). As

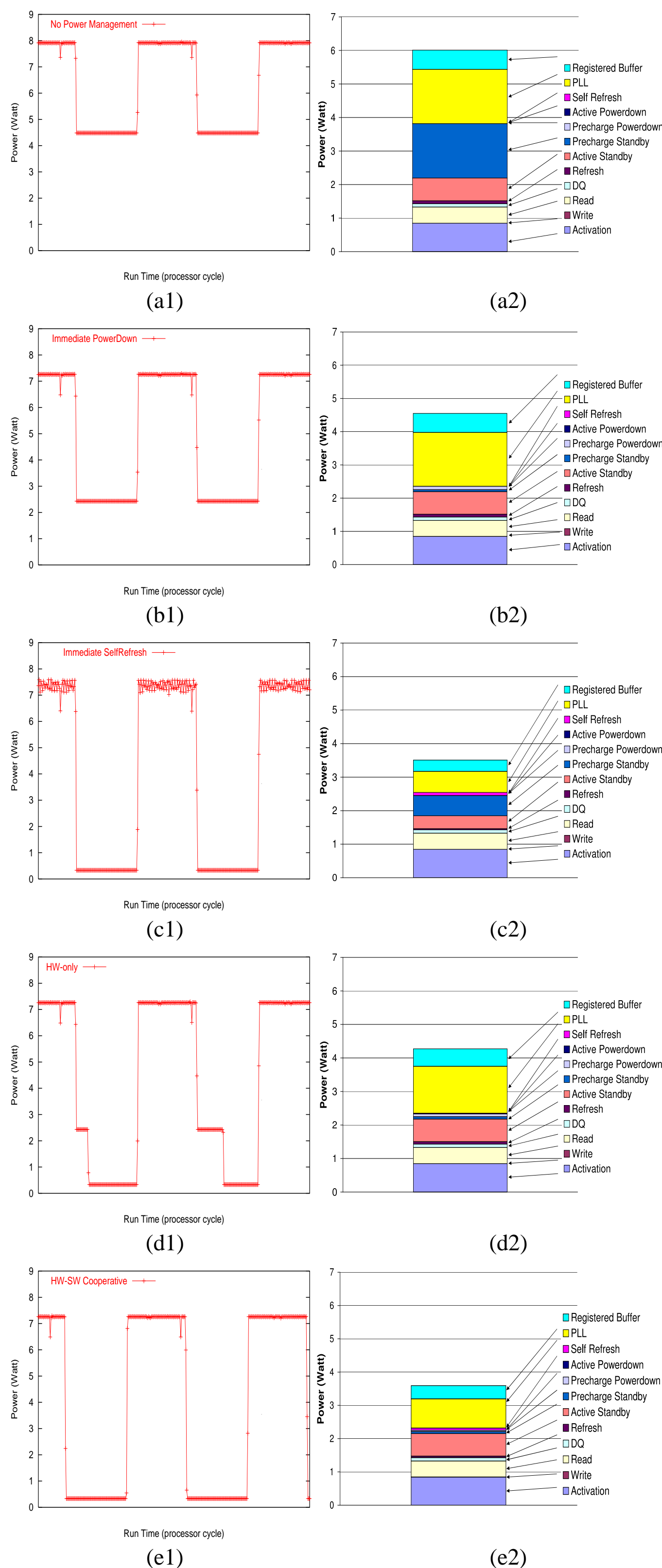

Figure 8: The first column gives the instantaneous power for a zoomed-in portion of the synthetic workload under (a) no power management (b) Immediate Power Down (c) Immediate Self Refresh (d) HW-only and (e) HW–SW techniques. The second column shows the breakdown of the average power dissipated.

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.

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 achieve additional energy benefit by also powering down the PLL and the registers, although at a severe performance penalty. We will look at their performance implications in detail shortly.

Next, Figure 8(d1) shows the results when power management decisions is dynamically 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). The PMU keeps history information on past accesses in its internal registers which are used to dynamically predict threshold values 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 is reasonable because it can avoid over-compensation and provide good adaptability to realistic workloads. However, the result shows that it only outperforms the IPD strategy by approximately 6% in power because when the hardware tries to make power management decisions based on its observation on the past memory access behavior, it gets confused when two processes with very different access behaviors are accessing the same rank in an interleaved-manner. One can argue that if the window size is reduced to 100 μsec or even 10 μsec, we can adapt more quickly. However, shrinking the window size is a double-edged sword, having this better adaptability runs at a higher chance to over-aggressively predict threshold values from observing transient behaviors at run-time. Shrinking the window size can benefit this synthetic workload, but for realistic workloads, it can cause more harm than benefits. As we will show in the next section, mistakingly entering Self Refresh mode can be very expensive. Furthermore, as more and more systems are switching to smaller scheduling quanta (e.g., from 10 msec to 1 msec or even smaller) to increase responsiveness in the system, higher context switching rate will make the hardware predictor's job even 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 the past access history specific to 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 referencing process starts to run, the PMU is able to instantaneously put the rank to Self Refresh, thus saving more energy. Additionally, unlike in the case of the HW-only technique, the cooperative technique will not be affected when the task scheduling quantum becomes increasingly smaller over time.

4.2.2 *Results*

The effect on energy can be easily obtained in our simulator. Performance implication is more difficult to quantify though, 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 dependency information among memory requests is not retained in a trace-based approach. Therefore, there is no way for us to know whether a delayed memory transaction will also delay a memory request that goes to an independent rank. 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 all other results for the synthetic workload.

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 of 105.04 cycles (+8.4%) and consumes only 7.83 J (-24.3%). A few percent increase in the average response time is usually not a big problem for server-type workloads as most are typically bandwidth-limited. On the other hand, when using Immediate Self Refresh, even though we can get an additional energy benefit (6.04 J, -41.6%), but it comes at a prohibitively high average response time (894.01 cycles, +822.42%). Compared to these static techniques, the dynamic ones perform much better. They consume almost as little energy as ISR but without ISR's hefty performance penalty, and they consume much less energy than IPD but pays almost as little performance penalty as IPD. Among the dynamic techniques, the HW–SW cooperative technique shows clear energy benefits over the HW-only approach. Specifically, it consumes 15.9% less energy than HW-only and also has a slightly better average response time. In Table 2, we also show the number of delayed requests due to exiting Power Down (PD) and Self Refresh (SR). Exiting PD is only 1 memory clock cycle, whereas exiting SR is much more expensive — 200 memory clock cycles. One of the reasons why HW–SW consumes less energy and has lower response time than the HW-only approach is that it can more accurately predict threshold for entering Self Refresh, and this is apparent from observing that HW–SW has far fewer number of delayed requests due to exiting from SR.

4.3 SPEC Benchmarks

One of the benchmarks we used in our evaluation is SPECjbb2000 [35]. It is implemented as a Java program emulating a 3-tier server system with an emphasis on the middle tier. The tiers simulate 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 a database on the third tier. In a benchmark run, one can instantiate multiple warehouses, each with a 3-tier system. Each warehouse executes as a separate Java thread within the JVM, and is mapped to a different Linux process. However, since all warehouses are essentially running the same type of workload and they all share the same memory address space within the JVM, we will only observe a small amount of variation in how memory is accessed between context switches among these SPECjbb processes. In such systems, the benefit of using the HW–SW power

Benchmarks	Total Runtime (processor cycles)	% of Total Runtime	Read Operations	% of All Reads	Write Operations	% of All Writes	Context Switches
Low memory-intensive workload							
SPECjbb process 1	470,662,157	4.5%	495,849	5.95%	148,964	4.67%	283
SPECjbb process 2	430,865,647	4.1%	463,402	5.56%	150,847	4.73%	233
SPECjbb process 3	614,658,695	5.9%	500,704	6.01%	151,581	4.75%	350
SPECjbb process 4	389,326,169	3.7%	499,898	6.00%	146,077	4.58%	218
SPECjbb process 5	544,571,120	5.2%	511,707	6.14%	141,688	4.44%	309
SPECjbb process 6	330,170,302	3.2%	421,781	5.06%	110,106	3.45%	197
SPECjbb process 7	1,694,958,880	16.3%	1,281,690	15.39%	212,097	6.65%	921
SPECjbb process 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 workload							
SPECjbb process 1	510,607,464	4.6%	704,477	1.29%	194,867	1.31%	734
SPECjbb process 2	535,188,637	4.8%	772,954	1.41%	223,225	1.51%	478
SPECjbb process 3	510,438,599	4.6%	581,688	1.06%	186,979	1.26%	465
SPECjbb process 4	529,700,398	4.7%	768,019	1.40%	221,891	1.50%	420
SPECjbb process 5	941,338,844	8.5%	1,167,305	2.13%	303,557	2.05%	550
SPECjbb process 6	473,391,039	4.2%	776,669	1.42%	309,628	2.09%	715
SPECjbb process 7	808,101,475	7.3%	1,041,908	1.90%	277,971	1.88%	508
SPECjbb process 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 workloads. SPECjbb is ran with 8 warehouses, each spawned as a separate Java thread.

management technique is limited. However, in real server systems, where the processor time is shared among multiple users and their applications, multiple 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 a system, we decided to run a few SPECcpu2000 benchmarks with well known execution behavior in parallel with the SPECjbb workload. We classified these workloads as either "*high memory-intensive*" or "*low memory-intensive*", based on L2 miss rates [8]. 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*. *Reference* input sets are used for these SPECcpu2000 benchmarks.

4.3.1 Results

The run-time statistics of the two workloads are shown in Table 3. For each process in our benchmark, we keep 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. In our experiment, we keep the system idle at the start of each run. To verify that the non-benchmark processes in the system, e.g., shell, background daemons and interrupt service routines, did not interfere with our runs and results, we compare the total CPU time, the total number of read and write operations and the total number of context switches incurred by our benchmark with the total number that was observed during the entire experimental run. For the low memory-intensive workload, benchmark processes used 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. Moreover, the reasons that the total number of context switches into the benchmark processes is significantly smaller than the total number that was observed in the entire run is due to a polling keyboard device driver, which periodically wakes up and then goes back to sleep. The run-time information gives us more confidence in our results as these benchmark processes are minimally affected by the system's background noise. From these run-time statistics, we can also see that SPECjbb benchmark is more memory intensive than *bzip2* and *crafty*, but much less than *mcf* and *art*.

In Figures 9(a) and 9(b), we show the instantaneous power dissipation throughout the entire run of the low memory-intensive and high memory-intensive workloads, respectively, for various power management techniques. IPD is assumed to be implemented in the memory controller to complement all other power management techniques (except for ISR) that we will evaluate. Here, we compare five techniques against each other — IPD, ISR, SW-only (PAVM), HW-only, and HW–SW. The resulting power, energy, and average response time are summarized in Table 4 and Table 5.

First we look at the static techniques. IPD by itself uses much more power than the other techniques, and it has only a slightly better average response time than SW-only, HW-only, and HW–SW approaches, and therefore, is not useful by itself. ISR's prohibitively-high average responsive time makes it not practical to use either by itself. Dynamic techniques perform much better than these static techniques. Among the three dynamic techniques, PAVM performs the worst, and HW–SW performs the best in terms of power savings. For the low memory-intensive workload, HW–SW consumes 16.7% less energy than HW-only, and 23.6% less energy than SW-only. It also has a comparable average response time (130.47 cycles) to SW-only (128.74 cycles) and HW-only (128.96 cycles). For the high memory-intensive workload, HW–SW consumes 14.1% less energy than HW-only, and 14.7% less energy than SW-only, and it has only a slightly higher response time (148.64 cycles) than both SW-only (144.45 cycles) and HW-only (145.94 cycles) approaches.

4.3.2 RDRAM

RDRAM is a new memory architecture that has emerged in the recent years. It has some 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. We believe that similar results can be achieved, as RDRAM has a similar set of power states with varying transition times, and is organized with multiple entities that

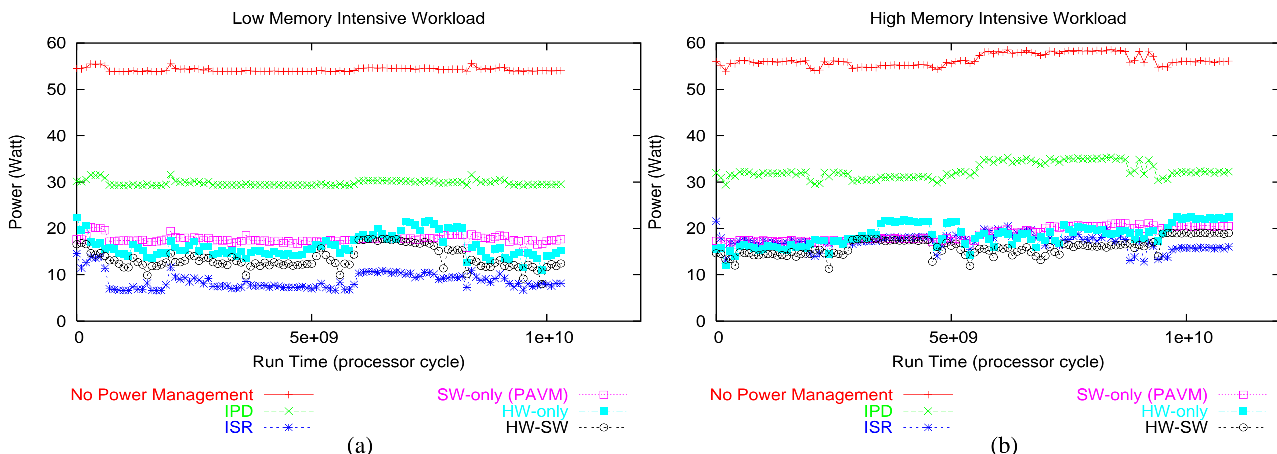

Figure 9: (a) Instantaneous power for the low memory-intensive workload. (b) Instantaneous power for the high memory-intensive workload.

	No Power Management	IPD	ISR	SW-only (PAVM)	HW-only	HW–SW
Energy Consumption	353.11 J	194.04 J	56.73 J	114.92 J	105.33 J	87.79 J
Average Power	53.24 W	29.81 W	8.71 W	17.65 W	16.18 W	13.48 W
Average Response Time	114.77 cycles	126.06 cycles	1121.73 cycles	128.74 cycles	128.96 cycles	130.47 cycles
Delayed Accesses Due to PD	0	6,790,058	0	5,871,024	6,771,680	5,863,568
Delayed Accesses Due to SR	0	0	2,704,257	1,155	10,111	4,925

Table 4: Summary of low memory-intensive workload.

may be independently power managed. In fact, RDRAM power states can be controlled at the device-level, rather than rank, providing a finer level of control than DDR. This 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 as we increase the number of power-controllable memory entities in a system (as in the case of large server systems), there is a diminishing return. The proposed techniques are 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 in computing systems by exploiting power management capabilities built into modern hardware components. In particular, a large body of the existing work has focused on reducing processor energy consumption. Weiser *et al.* [38] first demonstrated the effectiveness of using Dynamic Voltage Scaling (DVS) to reduce power dissipation in processors. Later work [3, 14, 25, 31] 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 [37, 17, 20], disk drives [23, 5, 21], flash devices [4, 27], caches [1, 19], and main memory [7, 12, 13, 10, 11, 16], while others [15, 40, 26, 34] 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 main memory, there are two main types of approaches — hardware and software-controlled. Among the hardware-controlled approaches, Lebeck *et al.* [7, 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 are orthogonal to the works described above and can be used to improve the prediction accuracy in some of these previously-proposed threshold prediction mechanisms. However, unlike these previous works, the techniques proposed in this paper are specifically designed and optimized for a multitasking environment, as are most of today's systems. Furthermore, we have also taken into account of various OS effects, which were shown to be also important in practice [16].

Among the software-controlled approaches, Delaluz *et al.*[11] demonstrated a simple scheduler-based power management policy. Huang *et al.* [16] 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 virtual memory 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 are 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 [33, 2, 6, 24] has also been shown to be beneficial in terms of improving performance and security, and providing new functionalities.

6. CONCLUSION

In this paper, we proposed a novel power management technique that makes use of cooperation between the system software and the

	No Power Management	IPD	ISR	SW-only (PAVM)	HW-only	HW–SW
Energy Consumption	390.18 J	225.21 J	105.31 J	130.25 J	129.40 J	111.11 J
Average Power	56.42 W	32.56 W	15.23 W	18.83 W	18.71 W	16.07 W
Average Response Time	134.08 cycles	144.11 cycles	909.73 cycles	144.45 cycles	145.94 cycles	148.64 cycles
Delayed Accesses Due to PD	0	20,688,949	0	16,242.587	20,641,111	16,228,517
Delayed Accesses Due to SR	0	0	4,944,750	476	17,357	5,999

Table 5: Summary of high memory-intensive workload.

memory controller hardware. It is shown to make a significant improvement in the accuracy of the PMU's threshold prediction logic. Using a full-system simulator, our HW–SW cooperative approach is shown to consume 14.2–17.3% less energy than the HW-only technique and 16.0–25.8% less energy than the SW-only technique. We used a uni-processor system to explore the feasibility of using this technique and quantified its benefits. We are planning to extend this work to multi-processor systems, where a combination of running processes, instead of a single running process, must be considered.

Furthermore, alternative to this software-assisted hardware power management technique proposed here, we can also imagine scenarios where the hardware can also provide feedback to the system software to create additional energy saving opportunities. For example, the hardware can inform the OS how "hot" each of the physical pages are being accessed, and the OS can use this information to re-arrange memory pages within each process's address space. This allows us to either (1) run hot ranks hotter and cold ranks colder to create more energy saving opportunities in the cold ranks, or (2) balance power dissipation on each rank and remove hot spots. Additionally, we would also like to explore direct cooperation between applications and the PMU. As applications themselves know more about their future memory access behavior than the OS, such information can prove to be beneficial to the memory controller in its prediction logic, and thus, can be used to further enhance the proposed power management system.

7. 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. Marsh. Thwarting the power-hungry disk. In *USENIX Winter*, pages 292–306, 1994.

[6] 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.

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

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

[9] P. Bohrer *et al.* Mambo —a full system simulator for the powerpc archtecture. In *ACM SIGMETRICS Performance Evaluation Review*, volume 31, 2004.

[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] 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.

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

[16] 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.

[17] 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.

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

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

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] Mesquite Software. http://www.mesquite.com.

[29] Micron. http://download.micron.com/pdf/technotes/tn4603.pdf.

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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

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

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

[37] 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.

[38] 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.

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

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