ABSTRACT

Exploiting Unused Storage Resources to Enhance Systems' Energy Efficiency, Performance, and Fault-Tolerance

by

Hai Huang

Chair: Kang G. Shin

The invention of better fabrication materials and processes in solid-state devices has led to unprecedented technological breakthroughs in computer hardware. Today's system software, however, often cannot take full advantage of the hardware's rapidly improving capabilities, thus resulting in idling resources, e.g., unoccupied memory and disk space. To make hardware operate more efficiently and to reduce the amount of idle resources, this thesis proposes several techniques that can harness such resources to the benefit of users.

Although there are many different types of hardware resources, this thesis focuses on the reclamation of idle resources in the storage hierarchy. First, we implemented a pure software technique to reduce the power dissipation of main memory. By aggregating unmapped and unused memory pages and powering down unused memory ranks, a significant amount of energy can be saved with little or no performance degradation. Next, we explored several architectural-level solutions that can more aggressively reduce energy, but at the expense of performance.

We also developed techniques to exploit unused disk capacity to improve disks' performance and energy-efficiency. These techniques are realized in our implementation of the Free Space File System (FS2). Unlike traditional file systems where extra disk space is not used, FS2 actively makes use of it to hold replicas of temporally-related data blocks that were poorly placed by the underlying file system. Using contiguous regions of free disk space to place related data blocks closer to one another enables disk heads to work more efficiently.

Free disk space may also be used to enhance the fault-tolerance of disks. The placement of replicas is shown to be critical to both the fault-tolerance and performance of file systems that make use of replicas. However, without a thorough understanding of how disks fail and how data become corrupted when failures occur, good data placement strategies are difficult to devise. We studied a large number of failed disks and analyzed their failure characteristics. This characterization study will help design more fault-tolerant file systems that can take advantage of today's large capacity hard drives.

Exploiting Unused Storage Resources to Enhance Systems' Energy Efficiency, Performance, and Fault-Tolerance

by

Hai Huang

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2006

To my parents.

ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Professor Kang G. Shin, for his guidance and support during my six years at U of M. This work would not be possible without his enormous amount of patience and understanding, for which, I am deeply grateful. I would also like to thank Professors Peter M. Chen, Jason Flinn, and Dawn Tilbury for serving on my thesis committee.

I am most indebted to my parents, my mother Mei L. Jin and my father Yong K. Huang, who have stood by me from the beginning and supported me with unconditional love. During my six years of graduate school, in spite of their long drives across states and some bad weather conditions, they would come and visit me very often, just to encourage me to keep going. This thesis is dedicated to them.

I would also like to express my sincere thanks to all my officemates—those who are still here and those who have already left, for their friendship. I would like to especially thank Babu Pillai for his help during my early years in graduate school and collaboration on several projects, John Reumann for being my technical sounding board, Hani Jamjoom, Cheng Jin, Haining Wang, Cong Yu, Zhiheng Wang, Songkuk Kim, Howard Tsai, Mohamed El Gendy, Jian Wu, Wei Sun, Zhigang Chen, and Jisoo Yang, just for being good friends.

Finally, I would like to thank my fiancée, Xiaoyu Jia, for her patience, tolerance, and constant encouragement. Having her with me has made this journey much less painful and a lot more fun.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

In 1981, the first IBM Personal Computer was introduced and ran on a 4.77 MHz Intel 8088 microprocessor with 16 KB of memory and one floppy drive. It cost $1,565, which would be nearly $4,000 today after inflation. We have come a long way since then. Continual innovation in computer hardware, at a rate closely following Moore's Law, has made high-performance / low-cost computers widely available. As a result, computers of all forms—ranging from small cellphones and PDAs to large server farms and mainframes—have integrated into every aspect of our daily lives.

Hardware's progress generally exceeds that of software, but its capabilities are often far from being fully utilized. As a result, the additional resources made available in modern hardware are sometimes wasted. This thesis will present the theory and practice of managing idle resources in systems. Before I describe how the excessive hardware resources can be utilized, I will first recap the significant progress that is made in computer hardware during the past two decades and the amount of additional resources that are made available as a result.

1.1 Resource-Rich Hardware

Leading the way is the microprocessor. In the past two decades, microprocessor core frequency has increased by three orders-of-magnitude. Important architectural innovations such as pipelining, superscalar, speculative executive, branch prediction, hyper-threading,

Year	1982	1985	1989	1993	1997	2001	2005
CPU	16-bit addr/bus	32-bit addr/bus	5-stage, FPU, on-chip I&D	2-way superscalar 64-bit bus	3-way superscalar Out-of-Order	Superpipelined on-chip L2	SMT, multi-core 64-bit addr/bus
Freq	12.5 MHz	16 MHz	25 MHz	66 MHz	200 MHz	1500 MHz	3200 MHz

Table 1.1: Processor trend [95].

and multi-core are the driving forces behind putting high-performance and low-cost processors in modern systems. However, these architectural innovations would not be possible without having much smaller transistors (half a billion transistors on a single die). In Table 1.1, we summarize various generations of microprocessors.

During the time when the microprocessor was progressing by leaps and bounds, storage devices, e.g., cache memory, dynamic RAM, magnetic devices, and flash memory, have also made large strides. Dynamic random access memory (DRAM) is one of the most active and volatile areas in the IT industry. Hundreds of companies came and went, which has made this technology sector highly competitive. This created a natural spawning ground for innovation, shown in Table 1.2. With the recent price drop in DRAM devices, today's consumer-grade PC machines are commonly installed with 1 or 2 GB of main memory. Such large amount of memory is usually more than enough for typical users, but any unused portion is often let idle and cannot benefit users in any way.

Year	1980	1983	1986	1993	1997	2000	2005
Memory	DRAM	Page Mode	Fast Page 32b	Fast Page 64b	SDRAM	DDR	DDR2
Mbits/chip	0.06	0.25	1	16	64	256	1024
Bandwidth	13 MB/s	40 MB/s	160 MB/s	267 MB/s	640 MB/s	1600 MB/s	3200 MB/s

Table 1.2: DRAM trend [95].

It is impressive how densely today's memory devices are packed, but compared to disks, their density still trails behind. We show the trend of various generations of magnetic disks in Table 1.3. In terms of capacity, the first IBM PC had only 10 MB of storage space, but nowadays, a single disk drive is capable of storing close to 1 TB of data. Transitioning from horizontal recording to perpendicular recording, the disk industry is expecting at least another two orders-of-magnitude increase in recording density. Constant decrease in price-per-gigabyte, at a rate of 60% per year, for the last 25 years, has made mass-storage in small form-factor widely available to common users. Compared to 10 years ago, we are much less constrained by the available disk capacity and often leave a significant amount of disk capacity idle. As we will see later in this chapter, unused disk capacity is actually a valuable resource that can be easily exploited and translated into a significant amount of benefits perceivable to users.

Year	1983	1990	1994	1994	2003	2006
Hard Disk	3600 RPM	5400 RPM	7200 RPM	10000 RPM	15000 RPM	7200 RPM
Capacity	0.03 GB	1.4 GB	4.3 GB	9.1 GB	73.4 GB	750.0 GB
Access Time	48.3 ms	17.1 ms	12.7 ms	8.8 ms	5.7 ms	12.3 ms

Table 1.3: Magnetic disk trend [95].

Introduced in the early 90's, flash memory was offered as an alternative to magnetic disks in persistent storage. Having higher density than DRAM devices and better shock resistance than magnetic disks has created a unique market for flash memory devices to thrive. Additionally, as information can be stored without dissipating any power, flash memory devices are especially attractive to small battery-powered devices. However, it is unlikely for flash memory to replace magnetic disks in the near future as there is still a significant gap in capacity (more than two orders-of-magnitude) between the technologies. The same argument also applies when comparing to DRAM devices as there is a five orders-of-magnitude difference between their access time. Despite the flash memory's small capacity, it is still rare for users to consume all the disk space. However, with its unique I/O access characteristics and low-power property, the unused portion of flash memory's capacity can potentially be exploited for interesting applications.

Year	1978	1995	1999	2003
Local Area Network	Ethernet	Fast Ethernet	Gigabit Ethernet	10 Gigabit Ethernet
Bandwidth	10 Mbps	100 Mbps	1 Gbps	10 Gbps
Delay	3000 ms	500 ms	340 ms	190 ms

Table 1.4: Local Area Network trend [95].

The Internet boom has brought over a billion people online, where most of them are from countries in Asia, Europe, and North America [117]. Internet users can access any information imaginable and chat with friends from thousands of miles away in real-time, all from the convenience of their home computer. As a result of the Internet boom, home networking equipments, e.g., Ethernet and wireless 802.11 devices, quickly became low-price commodity items. The upward trend of Ethernet devices' bandwidth and downward trend of their latency are shown in Table 1.4. Many homes with Internet access are switched from

Plain Old Telephone Switches to Cable, ISDN, and DSL services with plenty of network bandwidth. It makes us wonder how much of this bandwidth is actually getting utilized? 25%? 10%? 5%? Or less than 1%? Can network bandwidth be better utilized while not interfering with other users on the Internet?

In addition to the previously mentioned hardware, there are many other types of under-utilized hardware resources, and the most noticeable of them is the graphics processing unit (GPU). GPUs are used for texture mapping and polygon rendering in 3D applications. With their core frequency doubling roughly every 6 months [33] (even faster than the CPU's rate of improvement), these operations can be done at a blazing speed. However, as GPUs are utilized by only a small number of 3-D applications, they mostly sit idle in the background. Various interest groups are pushing for GPU manufacturers toward more general-purpose GPU architectures with an easier programming model. This will ideally allow GPUs to be used for general-purpose computation, but up-to-date, only a very specific set of scientific applications is able to take advantage of these GPUs [26, 120]. This is still a research area with many open questions and plenty of low-hanging fruits. In addition to the GPU, an ample amount of video memory is often found in today's video cards. Half of a gigabyte of video memory is common, which is comparable to the size of the main memory. It is not difficult to imagine having system software that can opportunistically take advantage of the idle video memory as a part of the main memory when running memory-intensive applications.

It is the aim of this thesis to reclaim unused resources for the benefit of users with a minimal amount of their effort and interaction. Today's systems are built from powerful hardware components operated by fine-tuned system software that can efficiently allocate resources to running tasks. But if the running tasks are highly volatile and require a unpredictable amount of resources, as in mobile devices and desktop workstations, inefficient use of hardware resources may result. Today's system software does not cope well with many unused (inactive) resources. We will show in this thesis that it is critically important for system software to manage both the *active resources* and the *inactive resource*, and that the potential benefit of utilizing the inactive resources can be quite high with low overheads.

There are some previous works done in this area, and the most widely known work is the SETI@Home project [111]. SETI stands for Search for Extra-Terrestrial Intelligence, which, as its name suggests, is used to detect life outside of Earth by analyzing radio signals collected by the Arecibo radio telescope. In this project, a massive amount of data is sent to 5.2 million participants around the world connected by the Internet and then processed by their idle CPU cycles. Besides harnessing the CPU cycles, projects such as Farsite [28], Pastiche [16], and Samsara [17] exploit free disk space from participating clients for distributing data across geographically different locations to implement a highly available, reliable, and secure data storage system.

Although there are many different types of hardware resources, this thesis focuses on the utilization of wasted resources in the storage hierarchy, and namely, memory and disk. Using unused memory pages and disk blocks, we will show how they can be used to improve systems' energy-efficiency, performance, and fault-tolerance, with energy management being our main focus.

1.2 Primary Research Contributions

In Section 1.2.1, we will first introduce energy management and its importance. Here, we will briefly discuss how unused memory pages and disk blocks can be used for reducing hardware's energy consumption. Next, in Section 1.2.2, we will discuss how the unused disk space can also be used for improving performance. Finally, in Section 1.2.3, fault-tolerance techniques designed to opportunistically use the unused disk space are described.

1.2.1 Energy Management

Faster, denser, and highly integrated hardware components are readily available, but unfortunately, they also dissipate a good amount of power. This causes problems in a wide variety of systems ranging from small battery-powered embedded devices to large-scale servers. Some of the most pressing problems in this area are discussed in the following paragraphs.

The need to be mobile in today's society has created a huge market for battery-powered devices, e.g., cellphones, PDAs, and laptops, but this mobility is severely constrained by the amount of battery energy these devices can carry. Despite having more power-efficient hardware, today's mobile devices do not necessarily have a longer operating time than some of the older devices. Driven to meet consumers' ever increasing needs and having to run bloated software, vendors are packing more, and more powerful hardware into smaller and smaller devices. Designed often for peak loads, these mobile devices are provisioned with much more hardware resources than needed for typical usage. And the fact that battery technology has always trailed behind hardware will only make the gap between energy supply and demand increasingly larger.

On large-scale server systems, we are faced with a different set of problems. These systems, including data centers, Internet service providers, and telecommunication switches, can have a power density ranging from $100\mathrm{W}/\mathrm{ft}^2$ – $300\mathrm{W}/\mathrm{ft}^2$[6], which is ten to forty times more than the power density of typical commercial office buildings. Data from several sources [4, 59] have indicated that the cost of electricity represents a significant portion of the Total Cost of Ownership (TCO) in such systems. In one of the studies [4], it was estimated that as much as 30–40% of the TCO in a particular data center is due to the cost of electricity used for powering computational and cooling equipments. Furthermore, as power dissipation becomes heat, hardware will tend to become less reliable and cause systems to become more prone to failures.

To reduce power dissipation and improve energy-efficiency of a system—whether for the purpose of elongating battery lifetime or for alleviating heat-related problems—we must first have a thorough understanding of each of its subsystems so that effective power reducing mechanisms can be deployed to target the most energy consuming parts. The storage subsystem is by far one of the most power-hungry subsystems, and reducing its energy consumption is one of our focuses in this thesis.

Storage Hierarchy Overview

A storage system is organized hierarchically as shown in Figure 1.1. Within this hierarchy, magnetic disk and main memory are often responsible for most of its power drain. Using custom-made circuits, it was reported in [8] that disks alone can consume more

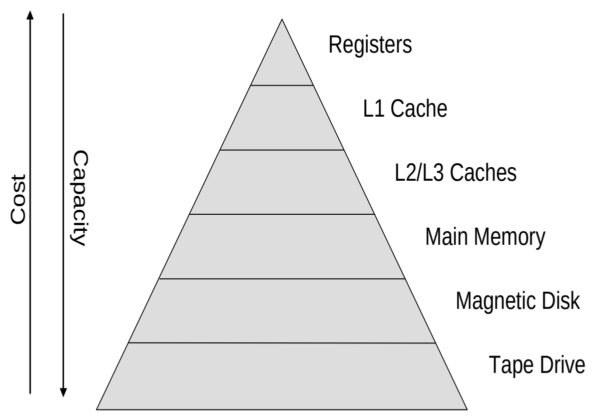


Figure 1.1: This figure shows a hierarchy of storage components. Components at the top are fast to access and small in size. Towards the bottom, components have lower performance, however with larger capacity and lower cost.

than 50% of the total system power. This translates to as much as 27% [100] of the total electricity cost in a data center.

Main memory also dissipates a significant amount of power. As reported in [75], 40% of the total system power (excluding the power drawn by disks) in a mid-range IBM eServer system is dissipated by the main memory. Even on small battery-powered devices, memory and disk can also consume a significant portion of the total power—up to 20% as reported by [84].

Main Memory

To reduce the power dissipated by main memory, we first implemented a purely software-controlled technique, called *Power-Aware Virtual Memory* (Chapter 2), where all the power-control mechanisms are implemented in the operating system. Leveraging the rich amount of information available in an operating system to manage power, we have shown that power can be reduced with little or no effect on performance. This approach has the benefit of requiring no additional hardware support beyond what is already implemented in typical PCs, and therefore, can be easily implemented in today's systems. However, due to some inherent limitations of the operating system, only a coarse-grained level of monitor and power-control can be achieved if the power is managed solely by the system software. As a result, many energy saving opportunities are lost. To re-capture some of the missed opportunities, we next implemented a software-directed hardware power management technique (Chapter 3) in which the system software is able to provide the memory controller with a small amount of information about the current state of the system. This enables the hardware to more accurately react to the changing state of the system for the purpose of power management. Fine-grained monitor and power-control mechanisms available only at the hardware level are exploited. However, using only the hardware to manage power is not sufficient as hardware has only a bottom-up and narrow view of the entire system. In our design, we have the system software provide it with some additional information to prevent it from making bad power management decisions when they can be avoided and to allow it to make more aggressive decisions when they are safe to do so. Lastly, we implemented a memory-traffic reshaping technique (Chapter 4) to aggressively expose idle time in main memory with little impact on performance. This technique can complement some existing power management techniques that rely on having long idle periods in memory devices.

Magnetic Disk

Reducing power for magnetic disks requires a very different strategy from that of the main memory. Memory has a constant access time regardless of the data's location, whereas a disk has access times that can vary by a few orders-of-magnitude. Additionally, memory devices can transition to a ready state from a low-power state in a few nanoseconds, whereas disks would take seconds or tens of seconds. Due to these differences, we must take a different approach when designing power management techniques for disks.

It has been shown in [57, 132] that idle periods in disks are often short and scarce, especially in server systems, thus rendering many traditional power management techniques [20, 21] not applicable sometimes, as they rely on the existence of long idle periods. We took a different approach—an active approach to improve disks' energy-efficiency. By reducing the number and the travel distance of head-seek operations, which dissipate a large amount of power, we can lower disks' energy consumption. To do so, we selectively replicate disk blocks that were observed to have frequently incurred long access times and place their replicas to strategic disk locations. As replicas can be accessed in place of their original data, head movements are minimized, in both their number and distance.

1.2.2 Performance Improvement

As mentioned in the last section, maintaining performance is critically important when power management techniques are applied. In addition to *maintaining* performance, we will show unused hardware resources can also be used for *improving* performance. Specifically, we studied how the free disk space can be used to improve disk performance. As disks are often identified as one of the slowest hardware components in many systems—with access times that can take up to tens of milliseconds—improving their performance will significantly increase the overall performance of a system by shortening critical I/O paths.

The same replication technique we introduced in Section 1.2.1 can also be used to improve disks' access time. By ways of reducing seek time and rotation delay, we can significantly cut down on the average access time. Experimental results using real-world workloads have shown that disk drives can be much better utilized when system software can dynamically monitor and utilize the unused disk space during runtime.

1.2.3 Fault-Tolerance Enhancement

Besides energy consumption and performance, fault-tolerance is also an important area to improve as losing data can be extremely costly. In mission-critical applications, unprotected hardware failures can even lead to more serious consequences.

In this thesis, we are mainly concerned with disk hardware failures and any data loss that may result. It is one of the most commonly occurring hardware failures in today's computer systems. Disk failures come in four different flavors: firmware failure, electronic failure, mechanical failure, and logical failure. Firmware and electronic failures are usually completely recoverable as data are often kept intact in the aftermath. Logical failures are usually caused by human errors or file system bugs, which are not commonly found in production-grade operating systems. The type of disk failures that is the most destructive and the most difficult to protect against is a mechanical failure, which often lead to a partial and sometimes a complete loss of data on disk.

In systems with multiple disks, there are some existing fault-tolerance solutions, such as Redundant Array of Inexpensive Disks (RAID) [94], which can tolerate a certain number of disk failures without losing data by keeping additional parity information and mirroring techniques. However, for some systems, e.g., laptops and PDAs, having multiple disks is usually not an option due to form-factor constraints. We believe replicating important user data and file system's meta-data and placing the replicas on the disk itself (using its unused disk space) is a possible method to avert unexpected data loss. However, determining which data blocks to replicate, where to place these replicas, and how many replicas we should maintain are all difficult decisions to make, especially when disk failures are poorly understood.

We characterized 60 failed disks. From our observation, we believe replication should be implemented as an integral part of file systems. This is true only when performance and space overheads of maintaining the replicas are tolerable. Our study has shown some interesting results that can do exactly that. Similar techniques may also be extended to systems with multiple disks to provide additional fault-tolerance orthogonal to existing fault-tolerance techniques.

1.3 Thesis Overview and Organization

This thesis focuses on the development of software techniques to dynamically monitor and utilize unused storage resources for benefiting users. The advantage of this approach is that users can more efficiently use their hardware resources. We will show benefits, such as energy savings, performance improvements, and fault-tolerance enhancements, can be easily attained with our approach.

In Chapter 2, we consider how to identify and aggregate unused and unmapped memory pages in an operating system for power savings. Power-Aware Virtual Memory [62] is a software technique realized in a Linux kernel, and it is platform-independent. Only a small amount of code revision is needed to have it ported to another operating system, machine architecture, or memory architecture. This is the first work that we know to consider the effects of multi-tasking, virtual memory, and low-level operating system behaviors on power management.

There are many benefits associated with this software technique, but its coarse-grained power-control and monitor mechanisms do not save as much energy as some of the proposed hardware solutions (although with their own set of drawbacks). Chapter 3 discusses a cooperative hardware-software power management technique [63] for main memory. This cooperative technique can leverage the best of both worlds—fine-grained power-control and monitor mechanisms available only in the memory controller and the high-level state information about processes from the system software.

The power management techniques that we have proposed in the two previous chapters are all passive-monitoring techniques, which act upon the monitored state of main memory. In Chapter 4, we describe an active technique [61] to actively modify memory request streams so memory devices can be allowed to go to deeper sleeping states for a longer period of time to achieve additional energy savings.

In Chapter 5, we propose and implement the Free Space File System [60] that can actively make use of the unused portions of disks for storing replicated data blocks. This technique allows disks to be utilized much more efficiently in terms of energy consumption and I/O performance by reducing seek distance and rotational delay.

We believe intra-disk replication can improve the fault-tolerance of disks in systems with a single disk, but the lack of information on disk failures may result in an inefficient use of disk space for replication. We believe the characterization of disk failures described in Chapter 6 can help design more fault-tolerant data and replica placement strategies and allow file systems to make better use of today's large-capacity disks.

CHAPTER 2

Power-Aware Virtual Memory

2.1 Motivation

Current hardware technologies allow various system components (e.g., microprocessor, memory, hard disk) to operate at different power levels (and corresponding performance levels). 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. Such findings are based on the fact that many systems are designed for providing continuous service even when they are 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 some components to low-power states, or even power them off. Subsequently, when the load increases, the relevant system components are then restored to higher performance / power levels. Effectively, this provides performance on-demand while conserving energy during the 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.

A large body of previous research concentrates on reducing the power dissipation of processing elements in computer systems due to their high peak-power. However, using existing techniques, a Mobile Pentium 4 processor dissipates only 1–2 W on average when running typical office applications despite having a high 30 W peak-power [66]. From a software perspective, further effort to reduce power in microprocessors is likely to yield only a diminishing marginal return. On the other hand, there has been relatively little work done on reducing power used by the memory. As applications are becoming more data-centric, more power is needed to sustain a higher-capacity/performance memory system. Unlike microprocessors, a fairly substantial amount of power is continuously dissipated by the memory in the background independent of the current workload. Therefore, the energy consumed by the memory is usually as much as, and sometimes more than, that of the microprocessor. In this chapter, we propose a purely software-controlled power management technique, called Power-Aware Virtual Memory (PAVM). Implementing Power-Aware Virtual Memory (PAVM) allows us to significantly reduce power dissipated by the memory but with almost no impact on performance. Additionally, this technique requires no additional hardware support beyond what existing hardware already provides.

The rest of this chapter is organized as follows. Section 2.2 provides some background information on various memory technologies. Here we also give the memory system model that we use in the rest of the thesis. Section 2.3 describes our initial design of PAVM, while Section 2.4 describes the limitations of this prototype design and the necessary modifications needed to handle the complexity of memory management and task interactions in a real working implementation. Section 2.5 presents detailed simulation results. In Section 2.6, we discuss limitations of the proposed technique and ways to get around them. Finally, we conclude this chapter in Sections 2.7.

2.2 Memory System Model

Since the 1980's, the performance gap between memory (DRAM) and processor has been widening continuously—DRAM speed has been improving at an annual rate of only 7% while processor speed has been leaping at an annual rate of 40% [129]. Frequent interactions between memory and other I/O components, e.g., disks, networking hardware, and video devices (shown in Figure 2.1), are making it a crucial component in the overall performance of the system. However, since memory's power can only be reduced when it is

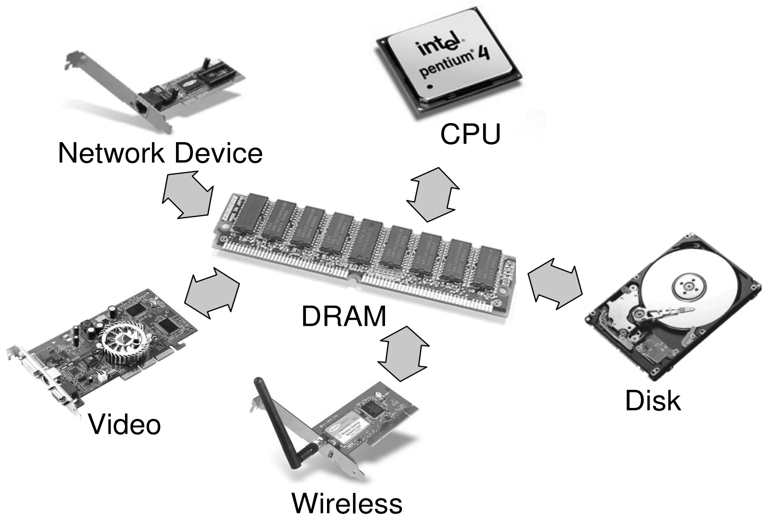


Figure 2.1: Interaction of memory with the rest of the system.

operating at lower performance states, it is important to ensure that either this performance degradation can be hidden or that the energy saved in the memory justifies the performance degradation that it had caused. Before illustrating the tradeoffs between performance and energy, we will first briefly describe the basics in DRAM technology.

2.2.1 Dynamic Random Access Memory

Dynamic Random Access Memory (DRAM) core consists large arrays of cells, each of which is a transistor-capacitor pair as shown in Figure 2.2. To prevent electric 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 very small as refresh rate is fairly low. Instead, most of the energy is actually consumed by row and column decoders, sense amplifiers, and external bus drivers, as well as the internal drivers needed to propagate signals across large arrays of cells over very long and high capacitance internal bit lines. To reduce power, one or more of these subcomponents are disabled by switching a device to a low-power state when it is not being actively accessed. To keep things simple, memory controller implements a simple interface that we can use to transition memory devices to various power states, and it will take care of all the power-up and power-down operations, and their timing constraints, thus controlling memory's power

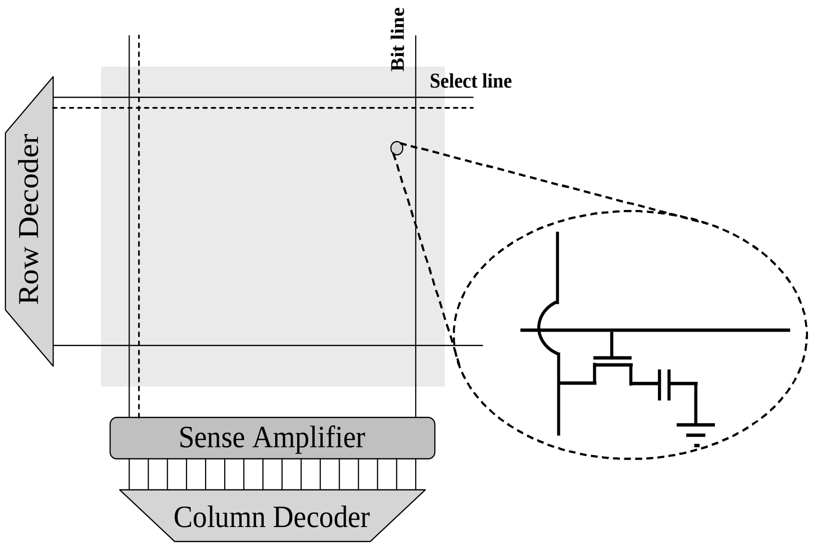


Figure 2.2: An overview of DRAM architecture with a magnified view of a single DRAM cell composed of a transistor-capacitor pair.

states in the software layer is trivial. However, if a device is accessed while it is still in a low-power state, a certain performance penalty, called a *resynchronization cost*, is incurred to transition the device to an active-ready state before it can be accessed again. This non-negligible delay is the cause of performance degradation when power management is applied to main memory, and masking it is vitally important as performance is still often the primary metric for many systems.

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

2.2.2 Double-Data Rate DRAM

Double-Data Rate (DDR) memory is usually packaged as modules, or DIMMs, each of which usually contains either 1, 2, or 4 *memory ranks*, which are commonly composed of 4, 8 or 16 physical devices (shown in Figure 2.3). Each time a DIMM is accessed, 64 bits of data are read or written. Since each device, depending on design, can only supply 4, 8, or 16 bits at a time, multiple devices are grouped together and operate in unison to satisfy a

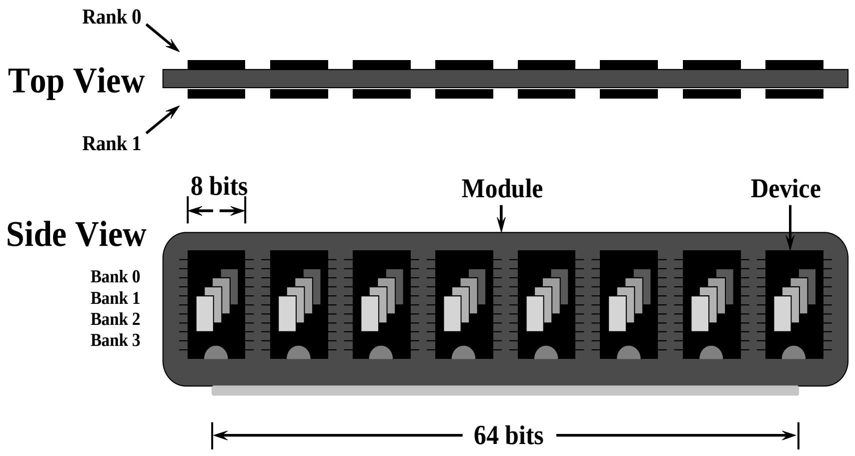


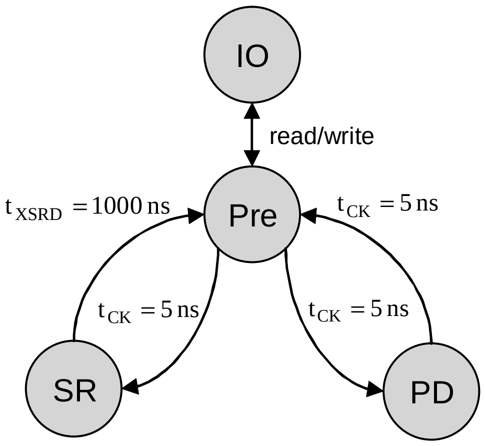
Figure 2.3: A memory module, or a DIMM, composed of 2 ranks, 8 devices per rank, and each of which is quad-banked.

64-bit memory access. This group of simultaneously accessed devices constitute a memory 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. Therefore, 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 these states [69, 88]. These states and transitions are fully simulated in our memory simulator, which we will discuss in Section 2.5. However, for simplicity of presentation, we only show four of these power states here—Read/Write, Standby, Powerdown, and Self Refresh—listed in a decreasing order of power dissipation. The power dissipated in each state and the transitional delays between them are shown in Figure 2.4(a). Note that the power numbers shown here are per single device. Therefore, to calculate the total power dissipated by a rank, we need to multiply this power by the number of device parts used for each rank (for ECC DIMMs, one additional device is used per rank to hold parity bits). For example, in 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. Details of these power states are as follows.

- **Read/Write:** Dissipates the most amount of power, and this state 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	Power
IO	Read/Write	1200 mW
PRE	Precharge	137.5 mW
PD	Powerdown	12.5 mW
SR	Self Refresh	10 mW

(a)

PLL	Power
Low-power	0.05 mW
High-power	750 mW

Register	Power
Low-power	0.025 mW
High-power	117.5 mW

(b)

Figure 2.4: (a) Power dissipation at each power state and the delay 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.

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 periodic refresh signal. I/O buffers, sense amplifiers and row/column decoders are all deactivated in this state.

- **Self refresh:** In addition to all the subcomponents on a DIMM that are deactivated in Powerdown, phase-lock loop (PLL) device and registers are also put to their corresponding low-power state to maximize energy savings as PLL and registers (Figure 2.4(b)) can consume a significant portion of the total energy on each DIMM. However, when exiting Self Refresh, a 1 μs delay is needed to resynchronize both the PLL and the registers.[1]

[1]Registered memory is almost always used in server systems to better meet timing needs and provide higher data integrity, and the PLL and registers are all critical components to take into account when evaluat-

As there is a large power difference 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 them down, and into which low-power state to transition will significantly impact both energy and performance. For the time-being, we refer to Standby as the high-power state, and both Powerdown and Self Refresh as the low-power state. We make the distinction between these two low-power states in Section 2.5 and illustrate how to best utilize each to maximize energy savings while minimizing performance impact.

2.3 Power-Aware Virtual Memory

Prior research on reduction of power dissipation in the main memory primarily focused on power management at a very low hardware level, where the memory controller is responsible for monitoring activity on each of the ranks and putting them to lower power states based on certain power-management policy. This has the benefit of being completely transparent to processes, but because the controller is totally unaware of the running processes, which actually are the entities responsible for all the memory accesses in the system, performing power management at such a low level can often lead to poor decisions which, in turn, degrade performance. In the design of Power-Aware Virtual Memory (PAVM), we elevate this decision making to the operating system level, where more information is readily available to make better state transitioning decisions to minimize performance degradation and reap greater energy savings. By elevating this decision making to the software level, this approach also has the advantage of not relying on any hardware support beyond what most existing systems already provide, and therefore, can be easily ported to other systems. Since a rank is the smallest unit of power management in memory, we now describe how to manage these ranks from the system software layer.

ing registered memory in terms of performance and energy.

2.3.1 Tracking Active Ranks

Since each rank's power state can be individually controlled, power can be reduced by selectively having ranks operating at lower power states. However, selecting which ranks to put into a low-power state is critical to both a system's performance and its memory subsystem's power dissipation since accessing a rank that is in a low-power state will incur resynchronization cost and stall execution, and, as a result, may increase energy consumption and offset any prior savings.

To avoid such costs, we need to ensure that all the ranks a process may access (i.e., its *active ranks*) in near future are kept in a high-power ready state during its execution. More specifically, we define a rank to be an active rank of process i, denoted as P_i, if and only if there is at least one page within this rank that is mapped to P_i's memory address space, and we denote the set of active ranks for P_i as α_i. By promoting all ranks in α_i to the highest ready state and demoting all other ranks (i.e., those in $\overline{\alpha}_i$) to a low-power state when P_i is running, we not only can reduce power but also can ensure that P_i does not suffer any performance degradation since none of the low-power ranks will be accessed while P_i is executing. In this work, we assume uniprocessor systems, but this technique can also be applied to multiprocessor systems. In such systems, instead of using the active ranks of the current running process, we need to use the union of the active ranks of all running processes.

Of course, this assumes that α_i can be easily tracked to accurately reflect the active ranks for each process P_i. Delaluz *et al.* [42] used repeated page faults and page table scans to keep track of the active set, but this involves very expensive, high overhead operations. Instead, we take a different approach: we maintain an array of counters for each process, and have each counter associated with each of the memory ranks in the system. The kernel is then modified such that on all possible execution paths in which a page is mapped into P_i's address space, the counter associated with the rank that contains this page is incremented. Similarly, when a page is unmapped, this counter is decremented. From these counters, the active set, α_i, is trivially derived: a rank is in α_i if and only if P_i's counter for this rank is greater than zero. The overhead of maintaining α is only one extra instruction per each mapping/unmapping operation, and is therefore negligible.

2.3.2 Reducing Active Set Size

By performing power management based on active sets, we can ensure that none of processes will suffer any performance loss during their execution. However, this approach does not necessarily guarantee any energy savings. In particular, if the number of ranks in (i.e., size of) the active set, $|\alpha|$, of each process is close to the total number of ranks in the system, power cannot be significantly reduced. So, to further reduce power, we need to minimize the total number of active ranks used by each process. This can be formally expressed as a minimization problem. Specifically, we want to minimize the summation, $(\sum \omega_i|\alpha_i| : i \in \text{all processes})$, where the number of active ranks, $|\alpha_i|$, of P_i is weighted by its CPU utilization (fraction of processing capacity/time spent executing the process), denoted by ω_i. However, allocating pages for all processes among all the ranks to minimize this sum is a very difficult and high-overhead problem to solve at runtime, even with a static set of tasks, let alone in a dynamic system.

For simplicity, we assume that an approximate solution can be obtained by minimizing the number of active ranks for each process. To this end, a simple heuristic can be applied by using the concept of the *preferred rank* and by maintaining a set of preferred ranks, ρ_i, for each P_i. A preferred rank is defined as follows. All processes start with an empty set of preferred ranks. When P_i allocates its first page, this page is taken from the rank with the most free memory space available, which is then added to ρ_i. Future memory allocations requested by this process are first tried on ranks in ρ_i. If all ranks in ρ_i are full (which happens very rarely), the allocation is made from the rank which currently has the most free memory space available, and this rank is then added to ρ_i. By using this worst-fit algorithm to generate ρ, each process's memory footprint is packed onto a small number of ranks, thereby decreasing each process's energy footprint.

2.3.3 A NUMA Management Layer

Implementing PAVM based on the above approach is not easy on modern operating systems, where virtual memory (VM) is extensively used. Under the VM abstraction, all processes and many parts of the OS only need to be aware of their own *virtual* address space, and can be totally oblivious to the actual physical pages used. Effectively, the VM decouples page allocation requests from the underlying physical page allocator, hiding much of the complexities of memory management from the higher layers. Similarly, the decoupling of layers works in the other direction as well—the physical page allocator does not distinguish from which process a page request originates, and simply returns a random physical page that is free, treating all memory uniformly. However, when performing power management for the memory, we cannot treat all ranks equally since accessing a rank that was previously put to a low-power state will incur higher latencies and consume more energy. Therefore, we need to eliminate this decoupling and make the page allocator aware of which process has made a page allocation request, so it can non-uniformly allocate pages based on ρ_i to minimize $|\alpha_i|$ for each P_i.

This unequal treatment of multiple sections of memory due to different access latencies and overheads is not limited only to power-managed memory. Rather, it is a distinguishing characteristic of the Non-Uniform Memory Access (NUMA) memory architecture, where there is a distinction between low-latency local memory and high-latency remote memory. In a traditional NUMA system, the physical location of a page used by a process is critical to its performance since local and remote memory access times can differ by a few orders-of-magnitude. Therefore, a strong emphasis has been placed on allocating and keeping the working set of a process localized.

In this work, by treating each rank as if it were physically located on a unique node in a large multi-node computer, we can employ a NUMA management layer to simplify the non-uniform treatment of the physical memory. This creates an illusion that each process is running under a different NUMA system, even though these NUMA systems are consisted of the same set of memory ranks. With a NUMA layer placed below the Virtual Memory (VM) system, the physical memory is essentially partitioned by ranks. Each rank has a separate physical page allocator, to which page allocation requests are redirected by the NUMA layer. The VM is modified such that, when it requests a page on behalf of P_i, it passes a hint (i.e., ρ_i) to the NUMA layer indicating the preferred ranks from which the physical page should be allocated. If this optional hint is given, the NUMA layer simply invokes the physical page allocator that corresponds to the hinted ranks. If the allocation fails, ρ_i must be expanded as discussed previously. By using a NUMA layer, we can implement PAVM with preferential rank allocation without having to re-implement the complex low-level physical page allocator.

2.3.4 Hiding Latency

The techniques described in the previous sections ensure that processes will not experience any performance loss during their execution. However, at each context switch time, a resynchronization delay is incurred when there is at least one rank that needs to be transitioned from a low-power state to a high-power state. This happens fairly commonly as different processes usually have different active memory ranks. As mentioned earlier, the transition from Self Refresh to Standby takes 1000 nanoseconds, which, although not a long time, is non-negligible if incurred at every context switch. This problem will only become worse as more operating systems are transitioning from 10 ms scheduling quanta to 1 ms quanta to increase system responsiveness. If this delay is not properly handled and masked, it could, as a result of increased runtime, erode prior energy savings and undermine the effectiveness of our technique.

One possible solution is that at every scheduling point, we find not only the best process (P_i) to run, but also the next best process (P_j). Before making a context switch to P_i, we transition the ranks in the union set of α_i and α_j to Standby mode. The idea here is that with a high probability, at the next scheduling point, we will either continue execution of P_i or switch to P_j. Effectively, the execution time of the current executing process will mask the resynchronization latency for the next process. Of course, the cost here is that we will need to have more ranks idling in Standby mode than needed for the current running process, thus consuming more energy, but, with a high probability, performance degradation (i.e., long state transition latencies) can be completely eliminated.

We also proposed an alterative solution, which is easier to implement, has lower computational and energy overhead, but might not be able to hide resynchronization latency as well as the first approach. The intuition behind this solution is that context switching takes time, e.g., load new page tables, flush TLB and cache, and modify internal kernel data structures to get ready for executing the next scheduled process, and these operations are all done even before the next process's memory pages are accessed. We instrumented a Linux kernel to measure the context switching time in the scheduler function—from the time immediately after the scheduler has decided which process is to be executed next until the time when this process actually starts running. The cumulative distribution of the context switching time on a 1.6 GHz Intel Pentium® 4 processor is shown in Figure 2.5. Given that DDR takes 1000 nanoseconds to transition from Self Refresh to Standby, we can see that at every context switch, we will pay a resynchronization penalty equal to the time difference between 1000 nanoseconds and the actual context switching time. This already allows us to mask hundreds of nanoseconds in resynchronization penalty. Moreover, because there might be overlaps in the active ranks of consecutively executing processes, memory accesses (made by the newly scheduled process) that are within the range of any overlapped ranks will not incur any performance penalty as these ranks are already in a high-power ready state (previously used by the last scheduled process). Not only these memory accesses do not incur any performance penalty, they also help us further mask the perceived resynchronization latency until the time when the first memory access lands in a non-overlapping active rank that has not been completely resynchronized to full power. However, as we have mentioned before, its effectiveness is probabilistic as there is no way to guarantee a high degree of overlapping between active ranks of consecutively-scheduled processes.

With faster processors in the future, the cumulative distribution function of the context switching time shown in Figure 2.5 may shift toward left, making the second solution less attractive as latencies will less likely be masked. The first solution proposed is more general and may be applied without any hardware constraints, but at a higher energy overhead. In practice, however, even as processor frequencies have been increasing rapidly, context

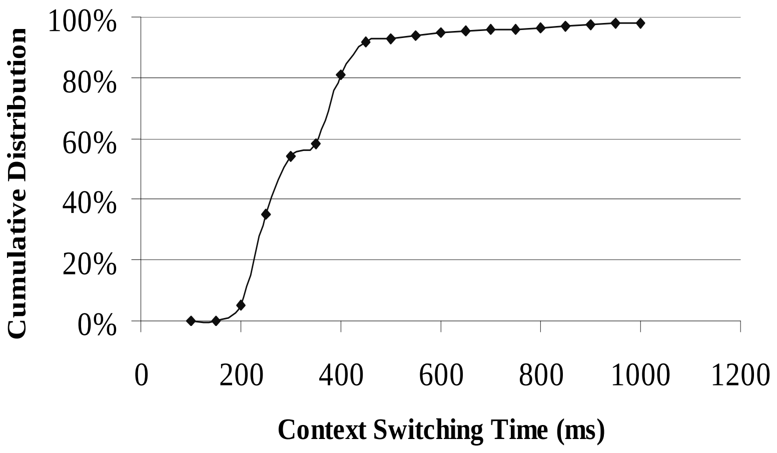


Figure 2.5: Cumulative distribution of context switch times.

switching time improves rather slowly, so the second solution should remain viable and more energy efficient under most circumstances.

2.4 PAVM Implementation

In this section, we describe our experience in implementing and deploying PAVM in a real system. Due to numerous complexities in real systems, a direct realization of the PAVM design described in the previous sections did not perform up to our original expectation. Further investigation into how memory is used and managed in the Linux operating system revealed insights that led us to further refinement of our original system, and we finally succeeded in conserving a substantial amount of energy in memory running complex real-world workloads.

2.4.1 Initial Implementation

Our first attempt to reduce memory's power dissipation is a direct implementation of the PAVM design (described in Section 2.3) in a Linux kernel. We extend the memory management data structures in the kernel to include various counters that are needed to keep track of the active ranks, α, per process. Furthermore, in the task scheduler, as soon as the next process is scheduled to execute, we transition all ranks in its active set to Standby, and all others to Self Refresh. This way, power is reduced, resynchronization time is masked, and the performance loss is minimized.

We also modify the page allocation code to use the preferred set, ρ, to reduce the size of the active sets for each process. Linux relies on the Buddy system [72] to handle the underlying physical page allocations. Like most other page allocators, it treats all memory equally, and is only responsible for returning a free page if one is available, so the physical location of the returned page is generally nondeterministic. For our purpose, the physical location of the returned page is not only critical to the performance but also to the energy footprint of the requesting process. Instead of adding more complexity to an already-complicated Buddy system, a NUMA management layer (described in Section 2.3.3) is placed between the Buddy system and the VM, to handle the preferential treatment of ranks.

The NUMA management layer logically partitions the physical memory and permits independent management and allocation to each segment. The Linux kernel already has data structures defined to accommodate architectures with NUMA support. To ensure the NUMA layer partitions memory according to memory ranks and permits rank-granular control of allocation, we populate these data structures with the memory geometry, which includes the number of ranks in the system as well as the size of each rank. As this information is needed before the physical page allocator (i.e., the Buddy system) is instantiated, determining the memory geometry is one of the first things we do at the system initialization time. On almost all architectures, memory geometry can be obtained by probing a set of internal registers on the memory controller.

Unfortunately, at the time when we implemented PAVM, NUMA support for the x86 architecture in Linux was not complete. In particular, since the x86 architecture is strictly non-NUMA, some architecture-dependent kernel code was written with the underlying assumption of having only uniformly accessed memory. We remove these hard-coded assumptions and add NUMA support for x86. With this modification, page allocation is now a two-step process: (i) determine from which rank to allocate a page, and (ii) perform the actual page allocation steps within that rank. Rank selection is implemented trivially by using a hint, passed from the VM layer, indicating the preferred ranks. If no hint is given, the behavior defaults to a sequential allocation. The second step is handled simply by instantiating a separate Buddy system on each rank.

With the NUMA layer in place, the VM is modified such that on all page allocation requests, it passes ρ of the requesting process down to the NUMA layer as a hint. This ensures that allocations tend to be localized to a minimal number of ranks for each process. In addition, on all possible execution paths, we ensure that the VM updates the appropriate counters to accurately bookkeep α and ρ for each process with a minimal overhead, as discussed in Section 2.3.

2.4.2 Shared Memory Issues

Having debugged the the initial implementation and ensured that the system is stable with the new page allocation method in place, we evaluate PAVM's effectiveness in reducing the energy footprint of processes. We expect that the set of active ranks, α, of each task will tend to localize to the task's preferred set, ρ. However, this is far from what we have observed.

Table 2.1 shows a partial snapshot of the processes in a running system, and, for each process P_i, indicates the ranks in sets ρ_i and α_i, as well as the number of pages allocated on each rank.[2] It is clear from this snapshot that each P_i has a large number of ranks in its active set, and $|\alpha_i|$ is much larger than the corresponding $|\rho_i|$. This causes a significantly larger energy footprint for each process than what we have originally anticipated. Nevertheless, since most pages are allocated on the preferred ranks, and none of the processes use all the ranks in the system, we still have opportunities to put some ranks into low-power states and conserve energy. However, it is not as effective as we would like, due to the fact that for each process, there is a significant number of pages that are *scattered* across a large number of ranks.

To understand this "*scattering*" effect, we need to investigate how memory is used in a Linux system. As in most systems, a majority of the system memory is used by user processes. Most of these pages are, in turn, used to hold memory-mapped files, which include binary images of processes, dynamically-loaded libraries (DLL), as well as memory-

[2]We only show a partial list of processes running in the system, but other processes behave similarly.

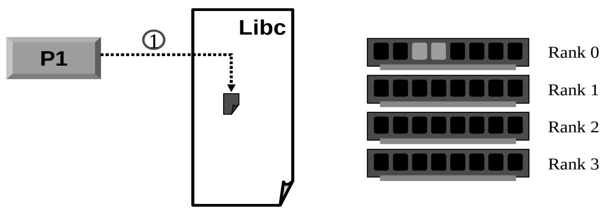
Process	ρ	α									
syslog	**14**	**0**(3)	**8**(5)	**9**(51)	**10**(1)	**11**(1)	**13**(3)	**14**(76)			
login	**11**	**0**(12)	**8**(7)	**9**(112)	**11**(102)	**12**(5)	**14**(20)	**15**(1)			
startx	**13**	**0**(21)	**7**(12)	**8**(3)	**9**(7)	**10**(12)	**11**(25)	**13**(131)	**14**(43)		
X	**12**	**0**(125)	**7**(23)	**8**(47)	**9**(76)	**10**(223)	**11**(19)	**12**(1928)	**13**(82)	**14**(77)	**15**(182)
sawfish	**10**	**0**(180)	**7**(5)	**8**(12)	**9**(1)	**10**(278)	**13**(25)	**14**(5)	**15**(233)		
vim	**10,15**	**0**(12)	**9**(218)	**10**(5322)	**14**(22)	**15**(4322)					
...	...	...									

Table 2.1: A snapshot of processes' rank usage pattern using the primitive implementation of PAVM. The notation **a**(b) indicates a process has b pages allocated on an active rank **a**.

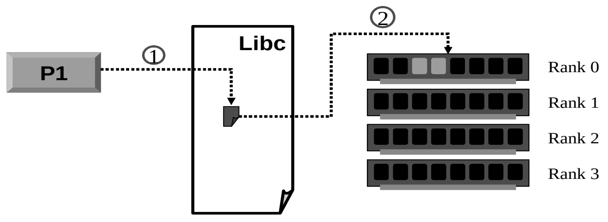
mapped data files. To reduce the size of the executable binaries on disk and the processes' cores in memory, DLLs are extensively used in Linux and most other modern operating systems. The scattering effect of pages that we observed is a result of this extensive use of DLLs combined with the behavior of the kernel-controlled page cache.

The page cache is used to buffer blocks previously read from the disk, so on subsequent accesses, they can be served without going to the disk, thus effectively reducing file access latencies. This causes complications for our technique. For example, when a process requests a block that is already in the page cache, the kernel simply maps that page to the requesting process's address space without allocating a new page. However, since this block may have been previously requested by any arbitrary process, it can be on any arbitrary rank, resulting in an increased memory footprint for the process. Unfortunately, this is not limited to shared data files, but also to DLLs, as these are basically treated as memory-mapped, read-only files. The pages used for DLLs are lazily loaded, through demand paging. So, when two processes with disjoint preferred ranks access the same shared library, demand-loaded pages in this shared library will scatter across the union of the two preferred sets, depending on the access pattern of the two processes and which process first incurred the page-fault to load a particular portion of the library into the page cache. An example is shown in Figure 2.6.

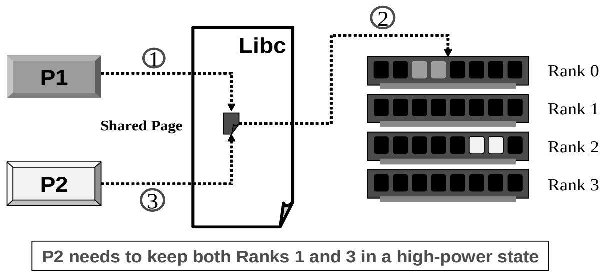
In what follows, we describe how to reduce the memory/energy footprint for each process by using DLL aggregation and page-migration techniques. We then discuss how to reduce the overhead of these new techniques.



(a)



(b)



(c)

Figure 2.6: (a) Process P1 tries to access a page in libc, which is not currently in memory. (b) Demand paging in P1's context brings the accessed page into memory and into P1's active rank (Rank 0). (c) When process P2 tries to access the same page, the page is simply mapped to P2's address space without going through the page allocator. As a result, P2's active set now will need to also include Rank 0.

2.4.3 Aggregation of Shared Libraries

Due to the many benefits of using shared libraries, or dynamically loaded libraries (e.g., libc), most, if not all, processes make use of them, either explicitly or implicitly. Therefore, a substantial number of pages within each process's address space may be shared through the use of shared libraries. As discussed above, this sharing inevitably causes pages to be littered across the entire memory range, resulting in a drastic size-increase of α for every process.

The cause of this scattering effect is that we are trying to load shared library pages onto

Process	ρ	α						
syslog	**14**	**0**(108)	**1**(2)	**11**(13)	**14**(17)			
login	**11**	**0**(148)	**1**(4)	**11**(98)	**15**(9)			
startx	**13**	**0**(217)	**1**(12)	**13**(25)				
X	**12**	**0**(125)	**1**(417)	**9**(76)	**11**(793)	**12**(928)	**13**(169)	**14**(15)
sawfish	**10**	**0**(193)	**1**(281)	**10**(179)	**13**(25)	**14**(11)	**15**(50)	
vim	**10,15**	**0**(12)	**1**(240)	**10**(5322)	**15**(4322)			
...	...	...						

Table 2.2: Effect of aggregating pages used by DLLs.

the preferred ranks of processes which first initiated the read-in from disk, as if these were private pages. To alleviate the scattering effect on these shared library pages, we need to treat them differently in the NUMA management layer. We implement this by ignoring the hint (i.e., ρ) that is passed down from the VM layer when allocating a page for shared libraries, and instead, resorting to a sequential first-touch policy, where we try to allocate pages linearly starting with rank 0, and fill up each rank before moving onto the next rank. This ensures that all shared pages are aggregated together, rather than scattered across a large number of ranks. Table 2.2 shows a snapshot of the same set of processes under the same workload as in Table 2.1, but with DLL aggregation employed.

As expected, aggregating DLL pages reduces the number of active ranks used per process. However, a new problem is introduced. Due to the extensive use of DLLs, by grouping pages used for libraries onto the low-address memory ranks, we allocate a large number of pages onto these ranks and quickly fill them. As a result, many processes will now need to have several of these low-address ranks in their active sets to access all the needed libraries without performance penalties. In the two snapshots shown, this is apparent: after aggregating all the shared library pages (shown in Table 2.2), both ranks 0 and 1 are mapped in all processes' active sets, whereas only rank 0 was needed without aggregation (shown in Table 2.1). With many libraries loaded, we would use up these earlier ranks fairly quickly, and may unnecessarily increase the memory footprint of some processes. We will give a detailed account of this in the next section and also describe how to relieve the extra pressure on these earlier ranks.

2.4.4 Page Migration

Even after aggregating these shared library pages, there is still some scattering of pages across ranks outside of ρ. Some of these are due to actual sharing of pages, but the rest are due to previous sharing and residual effects of past file accesses in the page cache. Furthermore, even though aggregating all the shared library pages ensures shared pages are kept on a few ranks, not all libraries are shared, or remain shared as the system execution progresses. It is better to keep these pages in the preferred ranks of the processes that are actively using them, rather than polluting ranks that are used for library aggregation and unnecessarily increasing the energy footprints of processes. We can address all of these issues by migrating pages dynamically.

In a NUMA system, page migration is used to keep the working set of a process local to the execution node in order to reduce average access latency and improve performance, particularly when the running processes are migrated to remote nodes for load-balancing purposes. In the context of PAVM, there is no concept of process migration, or remote and local nodes, but we can use page migration to localize the working set of a process down to a fewer number of ranks and overcome the scattering effect of shared pages and other objects in the page cache during runtime. This will allow us to have more ranks in low-power states, thereby conserving more energy.

In our implementation, page migration is handled by a kernel thread called *kmigrated* running in the background. As with other Linux kernel threads, it wakes up periodically (every 3 seconds). Every time it wakes up, it first checks to see if the system is busy, and if so, it goes back to sleep to avoid causing performance degradation to the foreground processes. Otherwise, it scans the pages used by each process and starts migrating pages that meet certain conditions. We further limit performance impact due to migration by setting a limit on the number of pages that may be migrated at each invocation of *kmigrated* to avoid spikes in memory traffic. Effectively, by avoiding performance overheads, the only penalty we pay from using this technique is a fixed energy cost for each page migrated.

A page is migrated if any of the following conditions holds:

- If a page is *private* (i.e., used only by one process, which is determined by the page's

Process	ρ	α			
syslog	**14**	**0**(15)	**14**(125)		
login	**11**	**0**(76)	**11**(183)		
startx	**13**	**0**(172)	**13**(82)		
X	**12**	**0**(225)	**1**(2)	**12**(2220)	
sawfish	**10**	**0**(207)	**1**(56)	**10**(436)	
vim	**10,15**	**0**(12)	**1**(240)	**10**(5322)	**15**(4322)
...	...	...			

Table 2.3: Effect of DLL aggregation with page migration.

reference count), and it is not located on a rank that is in this process's preferred set, ρ, then the page is migrated to any rank in ρ. This will not affect the size of the active set of any other processes.

- If a page is *shared* between multiple processes, and the rank that it resides on is outside of at least one of these processes' preferred sets, then the page is migrated to an earlier rank so it can be aggregated with the other shared pages, if and only if this migration does not cause the size of α to increase for any of the processes sharing the page.

Migrating a process's private page is straightforward. We simply allocate a new page from any active ranks of that process, copy the content from the old page to the new page, redirect the corresponding page table entry in that process's page table to point to the new page, and finally free the old page.

Migrating a shared page is more difficult. First, from the physical address of the page alone, we need to quickly determine which processes are sharing this page so we can check if it meets the migration criterion given above. Second, after copying the page, we need a quick way to find the page table entry for each of the sharing processes, so we can remap these entries to point to the new page. If any of the above two conditions cannot be met, an expensive full-range scan of the page tables of all processes is needed for migrating a single shared page.

To meet both conditions, we use the *rmap* [102] kernel patch, a reverse page mapping facility. It is included in the default kernel of the RedHat 7.3 Linux distribution and the

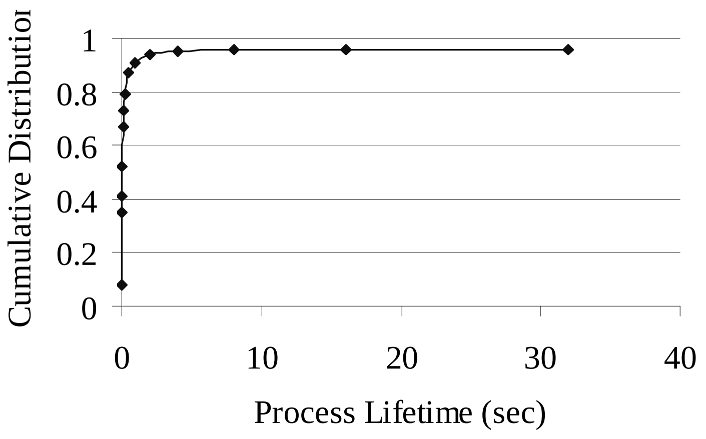


Figure 2.7: Cumulative distribution function of process life times.

recent stable Linux kernel releases. With *rmap*, if a page is used by at least one process, it will have a chain of back pointers (*pte_chain*) that indicates the page table entries among all processes that point to this page (meets the second requirement). In turn, for each page table containing the above page table entries, there is a back pointer indicating the process that uses this page mapping (satisfying the first requirement). So, when trying to migrate a shared page, we first allocate a new page, and find all the processes sharing this page to determine whether migrating this page will cause memory footprint to increase for any of the processes. If not, we copy the page content from the old page to the new page, replace all page entries that point to the old page with ones that point to the new page, update the reverse mappings in the new page, and finally free the old page.

With *kmigrated* running, processes use much fewer ranks (as shown in Table 2.3) than in the initial implementation of PAVM. As a result, memory power is significantly reduced. However, for each page migrated, we still incur a fixed energy cost for performing the memory-to-memory copy.

2.4.5 Reducing Migration Overhead

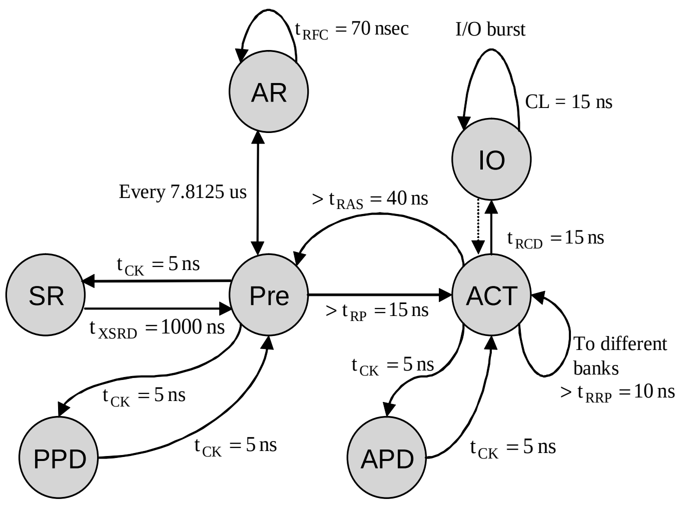
Although page migration greatly reduces the energy footprints of processes, it triggers additional memory activity, which may undermine energy savings. Thus, we must consider ways to limit the number of migration operations to have its benefits without incurring too much energy cost. We propose two solutions to reduce the number of page migrations.

Solution 1: The DLL aggregation technique described previously assumes libraries tend to be shared. Any library that is not shared will later be migrated to the process's preferred ranks. This is not efficient for those applications that use proprietary dynamic libraries. We can keep track of the processes that cause a large number of page migrations, and then classify them either as *private-page dominated* or *shared-page dominated*. A process is *private-page dominated* when the number of private pages migrated is much larger than the number of shared pages migrated. It indicates that the pages this process uses are not likely to be shared, meaning that we should allocate pages on this process's preferred ranks from the beginning and not automatically aggregate the library pages that it uses.
On the other hand, if a process is *shared-page dominated*, it means that many shared pages were initially wrongly migrated and then later moved back. For these processes, we want to inhibit the number of migrations for shared pages to prevent future migrations to reverse the initial migration decision.

Solution 2: It is well-known that a majority of processes are short-lived. Process lifetime is similar to what is shown in Figure 2.7 [81], where only 2% of all processes live more than 30 seconds. Instead of performing page migration for all processes, we only migrate pages of long-lived processes, since the energy spent on migrating pages for short-lived processes does not justify the resulting energy savings. Note that the current implementation of *kmigrated* implicitly avoids migrating pages for many short-lived processes, as it checks the system only once every 3 seconds and only when the system is not busy.

2.5 Evaluation

We now evaluate PAVM and compare it against some of the previously-proposed techniques. Section 2.5.1 describes the simulation environment and the methodology that we used to collect and analyze data. Section 2.5.2 describes the benchmarks used in our evaluation—SPECjbb2000 and SPECcpu2000. In Section 2.5.3, we briefly describe each of the power management techniques we evaluate here. Section 2.5.4 provides detailed simulation results.



	State	Power
PRE	Precharge	137.5 mW
PPD	Precharge-Powerdown	12.5 mW
ACT	Active	150 mW
APD	Active-Powerdown	112.5 mW
SR	Self-refresh	10 mW
AR	Auto-refresh	27.5 mW
IO	Read/Write	1200 mW

Figure 2.8: Detailed DDR state machine that we simulate in our memory simulator. Some minor states and transitions are omitted from this graph for better viewing.

2.5.1 Simulation Setup

PAVM was initially implemented on a real machine running Linux 2.4.18 kernel. However, to compare it with other power management techniques, we port it to run on a simulated machine as some of the other power management techniques cannot yet be implemented on real machines due to lack of hardware support. We used Mambo [36] as our simulated machine. It is a full-system simulator for PowerPC® machine architectures and is currently in active use by multiple research and development groups in IBM. It emulates both 32-bit and 64-bit PowerPC® processors and also supports system configurations and hardware components, including a multi-tiered cache hierarchy, SLBs, TLBs, disks, Ethernet controllers, UART devices, etc. The simulated machine is easily configurable, and very different system configurations 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 a Mambo-simulated machine (parameterized as shown in Table 2.4) in this work.

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

To evaluate various power-management techniques, we first use Mambo to record the main memory traffic (i.e., filtered by the L1 and L2 caches), save it to a trace file, and then feed it to 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 [86] library. It can accurately model performance and power dissipation of the memory by simulating a detailed DDR state machine (shown in Figure 2.8). Furthermore, it can also simulate the effect of queuing and contention occurring at the memory controller, at synchronous memory interfaces (SMIs), and on various buses. Power dissipation of memory devices is calculated by keeping track of the state information for each bank on a per-cycle basis, as was described in [87].

2.5.2 SPEC Benchmarks

One of the benchmarks we used in our evaluation is the SPECjbb2000 [115] benchmark. 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 Java Virtual Machine (JVM), and each is mapped to a different Linux process thread. 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. Such setup does not reflect how processes on real systems would use memory as they do not extensively share their address spaces. Additionally, in real systems, where the processor time is often shared among multiple users and their applications, system processes, and various daemon processes, we can expect memory access behavior in a typical system to change constantly when context switching between different processes at a fine granularity. All of these behaviors are not found in running the SPECjbb benchmark alone. To simulate these behaviors, we decided to run a few SPECcpu2000 [116] benchmarks that have well-known execution behavior in parallel with the SPECjbb workload. We classify these workloads as either "*high memory-intensive*" or "*low memory-intensive*," based on the L2 miss rates [106]. 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 all the SPECcpu2000 benchmarks.

The runtime statistics of the two previously described workloads are shown in Table 2.5. For each process in our workload, we keep track of the amount of CPU time it consumed, the number of memory read and write operations, and the number of times it was scheduled by the 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 benchmarks and their 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 run. For the low memory-intensive workload, the 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, the 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. However, in both workloads, the total number of context switches caused by benchmark

Benchmarks	Total Runtime (CPU 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 2.5: Summary of the low memory-intensive and high memory-intensive workloads. SPECjbb is ran with 8 warehouses, each spawned as a separate Java thread.

processes is significantly smaller than the total number that was observed in the entire run. This behavior is caused by a polling keyboard device driver (made specifically for running Linux in Mambo), which periodically wakes up and then goes back to sleep. This does not happen in real systems, but having it in our system does not interfere with the fidelity of our benchmarks and our results. The runtime information gives us more confidence in our results as these benchmark processes are minimally affected by the system's background noise. It is interesting to note that from the runtime statistics, SPECjbb is shown to be more memory-intensive than *bzip2* and *crafty*, but much less than *mcf* and *art*.

2.5.3 Power Management Techniques

In our evaluation, we compare 5 different power management techniques in terms of performance and power. They are listed as follows.

- **No Power Management (NOPM):** Here, no power management technique is used, and ranks are put to the highest ready state when idling. We use this as the baseline in our study.

- **Immediate Powerdown (IPD):** This is the simplest form of hardware-controlled power management. It is a static power-management technique where the memory controller immediately transitions a rank to Powerdown when all memory requests on this rank have completed.

- **Immediate Self Refresh (ISR):** Same as IPD, but transitions to Self Refresh instead of to Powerdown.

- **Dynamic Hardware Technique (HW):** This is a dynamic, epoch-based, hardware-controlled power management technique. It monitors past memory accesses, and based on which, makes predictions on after how long of an idle period it should transition a rank to Self Refresh (threshold prediction algorithm is shown in Figure 2.9). Transition to Powerdown has a zero threshold, which was previously shown to be the most efficient [25].

- **Power-Aware Virtual Memory (PAVM):** This is the software technique that we have proposed in Section 2.3 and Section 2.4.

2.5.4 Results

Figures 2.10(a) shows the instantaneous power dissipation for the entire duration of the low memory-intensive workload for different power-management techniques. These results are then summarized in Table 2.6. In this table, we show average power, average response time, normalized runtime and the number of delayed memory requests due to ranks being put to Self Refresh or Powerdown. We use two metrics to evaluate performance—average response time and normalized runtime. Average response time is useful to measure exactly how much impact power management has on the main memory, and normalized runtime shows impact on overall system's performance due to slowdown in the memory. The more important metric of the two is the normalized runtime as it represents the user-perceived system performance. For the less memory-intensive workloads, we can be more aggressive

t_{REFI}	= Auto refresh interval (7.8 usec)
${}_{i}T_{i+1}$	= Time interval between ith and (i + 1)th refresh interval
${}_{i}N_{i+1}[x]$	= Number of memory accesses within ${}_{i}T_{i+1}$ on rank x
$Th[x]$	= Threshold to enter Self Refresh on rank x
Th_{Max}	= Maximum threshold value

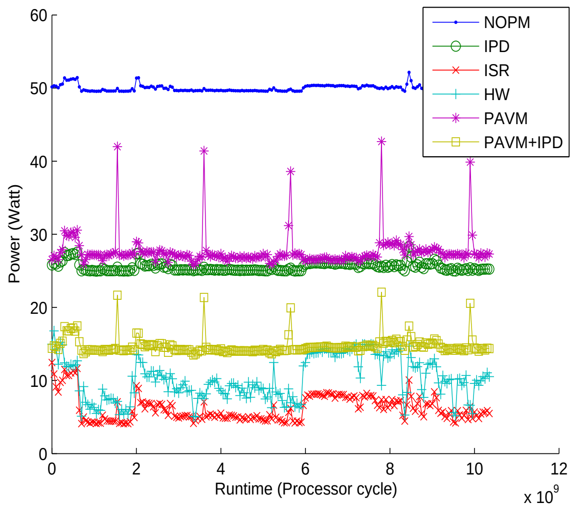
```
Threshold Prediction Algorithm:

At the end of the i^th refresh interval
    For each rank x
        if ($_{i}N_{i+1}[x]$ == 0) then
            if ($Th[x] < t_{REFI}$) then
                $Th[x] = Th[x]/2$
            else $Th[x] = Th[x] - t_{REFI}$
        else
            $Th[x] = Th[x] + {}_{i}N_{i+1}[x] * t_{REFI}$
            if ($Th[x] > Th_{Max}$)
                $Th[x] = Th_{Max}$
        if (rank x is not currently in Self Refresh)
            enter(Powerdown)
            if (idle time has exceeded Th[x]) then
                enter(Self Refresh)
```

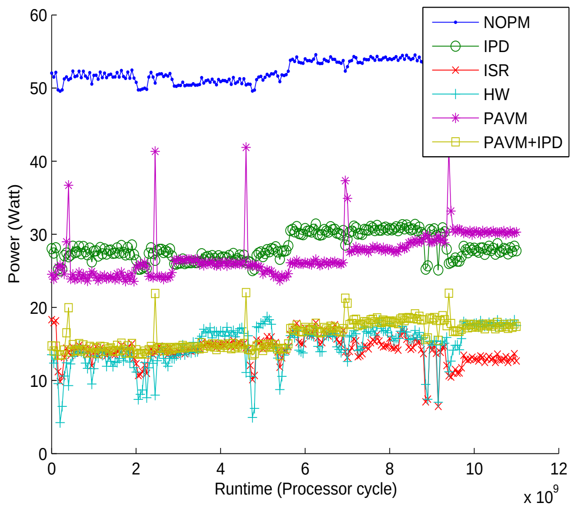
Figure 2.9: Description of the threshold prediction algorithm. Threshold value of each rank is determined periodically based on previously observed memory access patterns. For convenience, we choose this periodicity to be 7.8 μ*sec*, i.e., memory refresh interval for the memory devices we are simulating in this study. $\mathrm{Th}_{\mathrm{Max}}$ is currently set to 64 times $\mathrm{t}_{\mathrm{REFI}}$.

in making power management decisions. Even though this might result in higher average response time, this slowdown in the memory will likely to have only a small impact on the system performance. On the other hand, when memory is more frequently accessed, higher average response time will result in more significant degradation in the overall system performance. We will show these effects in our experiments shortly.

For the low memory-intensive workload, from Table 2.6, we can see that without any power management, memory dissipates 49.97 W of power, and we use this and its runtime as the baseline in our evaluation. We first look at the simplest hardware power management technique, Immediate Powerdown (IPD), where power is reduced to 25.54 W—a 48.89% power saving. Even though many memory requests are delayed due to ranks being put to



(a)



(b)

Figure 2.10: (a) Instantaneous power dissipation for the low memory-intensive SPEC workload. (b) Instantaneous power dissipation for the high memory-intensive SPEC workload.

Powerdown state, the total runtime has only increased by 1.80% because the resynchronization latency of exiting from Powerdown is so small. With Immediate Self Refresh (ISR), power is further reduced since both PLL and registers can be turned off. ISR dissipates only 6.23 W of power—a 87.54% power saving—but it is achieved only with a heavy hit on performance. It runs 110.70% slower than the baseline case, which makes it very impractical to use in real systems. Using the HW (dynamic hardware-controlled) technique, we can achieve a similar amount of power savings compared to ISR but with much less performance degradation. It dissipates only 10.24 W in power—a 79.51% saving (comparable to ISR)—and it runs only 3.21% slower than the baseline case. In the case of PAVM, even though it dissipates more power than the other power management techniques, nevertheless, it has achieved what we had in mind originally. First of all, it is completely software controlled and requires no additional hardware support. Second, power can be saved (44.70%) without any noticeable sacrifice in performance (in this case, runtime increased by less than 0.1%), thus making PAVM one of a kind. These are all desirable features to have in power management of real systems. Furthermore, we are over-estimating the amount of power saved in the hardware techniques as we do not take into account of the additional power dissipated to support the extra hardware components that are needed to implement these techniques, thus making software-controlled techniques even more favorable.

In term of average response time, IPD has a 14.1% higher average response time, ISR 868.74%, HW 25.18%, and PAVM only 0.4%. Even though IPD's 14.1% and HW's 25.18% increase in average response time did not impact system performance significantly (1.80% and 3.21%, respectively), we will see in the high memory-intensive workload that its effect is more severe. However, with PAVM, system performance can still be preserved.

Despite its performance advantage, PAVM does not save nearly as much power as some of the hardware techniques. This can be improved, however, at a price of performance. The reason that PAVM does not save as much power as the other techniques is because it only manages power for the inactive ranks and leaves active ranks in the highest power state, which is somewhat wasteful. To limit performance degradation when managing power for the active ranks, we decided to use IPD. As we have seen previously, IPD can save a substantial amount of power without significantly impacting performance and only requires a minimal amount of hardware support. We label this technique as PAVM+IPD in Figure 2.10. It dissipates 14.72 W in power and increases runtime by only 1.86%. Which technique is more appropriate to deploy will vary from system to system. For example, on a system where performance is of the uttermost importance but with energy conservation as another important metric, the generic PAVM technique will be more favorable as it can save power without disturbing performance.

The instantaneous power dissipated for the high memory-intensive workload is shown in Figure 2.10(b) and is summarized in Table 2.7. Here, similar conclusion can be drawn, but there are a couple of things worth noticing. Because memory is much more intensively accessed, managing power results in more performance degradation—IPD imposes a 8.10% slowdown in runtime, ISR 555.82%, and HW 7.16%. On the other hand, PAVM's performance impact is still under 0.5%. This allows PAVM to be deployed in situations where other power management techniques are not appropriate due to their performance impact.

	No Power Management	IPD	ISR	HW	PAVM	PAVM+IPD
Energy Consumption	275.02 J	143.15 J	68.26 J	58.35 J	152.27 J	81.09
Average Power	49.97 W	25.54 W	6.23 W	10.24 W	27.64 W	14.72 W
Average Response Time	72.01 ns	82.17 ns	697.60 ns	90.14 ns	72.29 ns	82.52 ns
Normalized Runtime	1.000	1.018	2.107	1.032	1.001	1.019
Delayed Accesses Due to PD	0	6,781,614	0	6,693,095	0	6,720,072
Delayed Accesses Due to SR	0	0	2,701,345	74,094	70	70

Table 2.6: Summary of the low memory-intensive workload.

	No Power Management	IPD	ISR	HW	PAVM	PAVM+IPD
Energy Consumption	482.11 J	282.79 J	743.42 J	149.21 J	252.24 J	150.42 J
Average Power	52.15 W	28.30 W	14.57 W	14.79 W	27.15 W	16.19 W
Average Response Time	84.13 ns	92.19 ns	537.45 ns	93.21 ns	84.59 ns	92.83 ns
Normalized Runtime	1.000	1.081	5.558	1.091	1.005	1.087
Delayed Accesses Due to PD	0	20,648,953	0	20,603,033	0	20,280,402
Delayed Accesses Due to SR	0	0	4,954,357	27,559	181	182

Table 2.7: Summary of the high memory-intensive workload.

RDRAM

RDRAM is a new memory architecture that has recently emerged. 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 and is organized with multiple entities that may be independently power managed. In fact, RDRAM power states can be controlled at the device level, rather than at the rank level, providing a finer grained 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. Very few number of things needs to be changed and the proposed techniques can be directly applied to the RDRAM memory architecture.

2.6 Discussion: DMA and Kernel Threads

In the current implementation, there are three limitations that we do not fully address. First, we do not consider direct memory access (DMA). Using DMA, hardware components can more efficiently transfer data between themselves and the main memory without

CPU intervention. However, without knowing whether a rank was put to a low state, DMA operations may suffer performance degradation. Fortunately, this problem can be easily mitigated by ensuring that DMA operations use only pages within a pre-defined physical memory range (e.g., the first rank), which, due to the use of DLL Aggregation, is almost always in the Standby ready mode, which will not result in any loss in performance.[3]

Second, kernel threads that run in the background may touch random pages belonging to any process in the system. Since these maintenance threads are invoked fairly infrequently, a simple solution is to treat these as special processes and turn on all ranks when they are invoked to avoid performance degradation.

Third, PAVM is very inefficient in managing power for the active ranks. A rank is active simply means that there is at least one process that has pages mapped onto it and may potentially access these pages in near future. However, how frequently or how infrequently an active rank is being accessed is unknown to the OS. Without such valuable information, the best one can do at the OS level is to either put active ranks to high power state or to use IPD, and both ways are wasteful in terms of both power and performance. This problem is more pronounced on dedicated server systems where almost all the memory in the system is used by a single process, e.g., database server, file server, DNS server, etc. Most likely, the active set of this process will contain all the ranks in the system, and thus PAVM will not be able to save much power. A much finer-grained level of control is needed to effectively manage power. In the next chapter, we propose a software-directed hardware technique that will allow us to do just that.

2.7 Conclusions

Due to better processing technology and a highly competitive market, systems are equipped with bigger-capacity and higher-performance main memory as workloads are becoming more data-centric. As a result, power dissipated by the memory is becoming increasingly significant. In this work, we have presented the design, implementation, and

[3]Note that due to the address space limitation in older ISA devices, Linux for x86 already limits DMA operations to use only the first 16 MB of memory (i.e., within the first rank).

analysis of Power-Aware Virtual Memory (PAVM) to reduce total memory energy expenditure by managing power states of individual memory ranks. We have also shown a working implementation of PAVM in the Linux kernel, and described how it was later evolved to handle complex memory sharing among multiple processes and between the processes and the kernel in a modern operating system.

Using PAVM, we show that from software alone, 44.70–47.94% of the power can be reduced. This technique can be directly applied to many of today's systems without additional hardware support and has negligible performance impact (well less than 1%)—thus this power saving comes almost for free. Combining PAVM with Immediate Powerdown (IPD), we show that 68.96–70.54% of the power can be reduced with only 1.9–8.7% performance penalty. It would only require as much additional hardware as IPD alone, but it saves much more power than IPD (45.74–48.89%) can achieve by itself. Compared to a more complex dynamic hardware technique, where it saves 74.66–82.71% in power and causes 3.2–9.1% performance degradation, PAVM and its derivative certainly have many advantages and can be more generally deployed.

CHAPTER 3

Software-Directed Memory Power Management

3.1 Motivation

In the previous chapter, we proposed a purely software-controlled power management technique called PAVM. It is different from previously proposed techniques because it can save energy without impacting performance. However, despite its performance advantage, it is not as power efficient as some of the hardware techniques. It always puts the active ranks to the highest power state as it can neither differentiate a frequently accessed rank from an infrequently accessed rank, nor detect any memory access patterns that may be exploited. The software technique treats all active ranks the same, and due to which, many energy saving opportunities are lost. To re-capture these missed opportunities, we will propose a software-directed power-management technique. We will show that with a minimal amount of cooperation between the system software and memory controller hardware, power can be more aggressively reduced.

The rest of this chapter is organized as follows. We begin with a brief design overview in Section 3.2. Hardware and software control mechanisms are described in Section 3.3. Results from detailed evaluation using both synthetic and SPEC workloads are given in Section 3.4, and finally we conclude in Section 3.5.

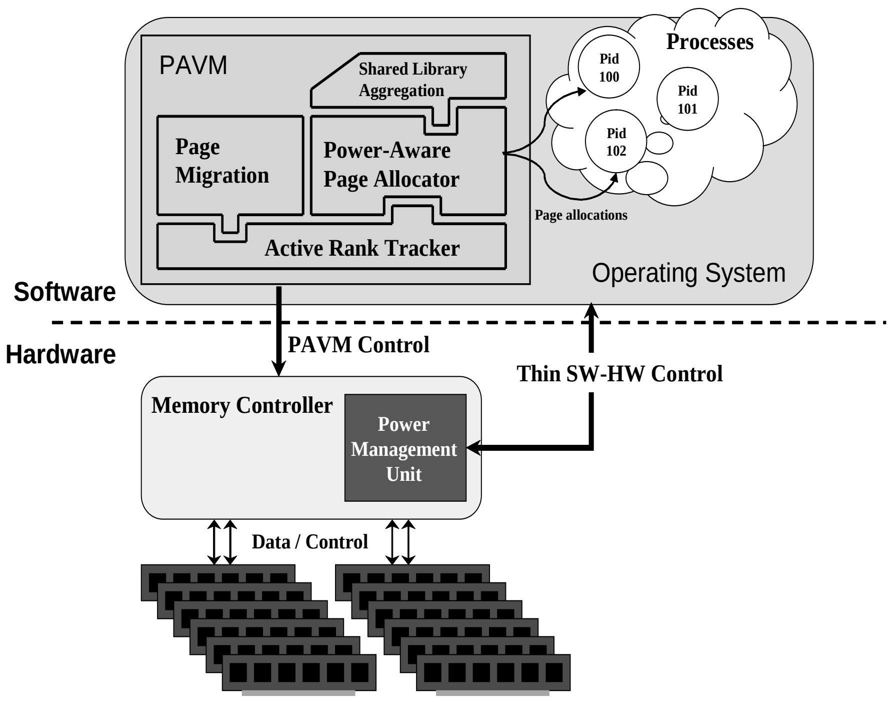


Figure 3.1: Architectural overview of the software-directed power-management system.

3.2 Overview

Power management implemented in hardware (i.e., memory controller) has some obvious advantages over software techniques, but it has its own set of deficiencies. On one hand, as the memory controller is able to observe all memory traffic in real-time, it allows for implementation of a very fine-grained and highly-adaptive control mechanism that can be used to ideally glean all possible energy-saving opportunities. On the other hand, the effectiveness of using hardware to manage power is heavily dependent on how accurately the memory controller is able to predict future memory references from its past observations. Unfortunately, accurate prediction is often difficult to accomplish by the hardware alone, especially in systems with multiple active processes running, each with its own memory access behavior.

As the memory controller has no understanding of neither processes nor the operating system, it can get easily confused from its observation of memory traffic. This is especially true during context switching time when memory access pattern can suddenly change. To avoid making bad power management decisions during such time, hardware will need to adapt to the newly invoked process's memory access pattern quickly, but in reality, it takes a non-negligible amount of time to do so. Even after the memory controller has adapted itself to the access pattern of the newly invoked process after a while, it will need to re-adapt itself again at the next context switch, which will happen shortly afterwards. This problem will only get worse as systems are making use of smaller and smaller scheduling quanta (higher context switching frequency) to increase their responsiveness. 5 to 10 years ago, most Linux systems were using 10 ms scheduling quantum. Today, 1 ms quantum is becoming the standard scheduling interval. However, as scheduling quantum keeps getting smaller, hardware will spend more time re-adapting and less time making the right power management decisions.

As hardware-controlled and software-controlled techniques both have their own set of problems, we will propose a software-directed hardware power-management technique. In this approach, memory controller will respond to low-level memory access patterns not visible to the system software, while the system software can provide the hardware with high-level information about the current system state, permitting aggressive but accurate power management decisions to be made at the hardware level. Figure 3.1 depicts the system architecture used in this software-assisted power-management approach. We will elaborate on this design shortly.

In the next section, we first describe the architecture of a generic power management unit (PMU) implemented in the memory controller. The PMU is responsible for monitoring memory traffic and controlling power for DRAM devices. In Section 3.3.1 we describe how to minimally modify this PMU so it can communicate with system software to gain additional information about the current state of the system. Then, in Section 3.3.2, we describe what system and process state information are useful to the modified PMU and how system software can convey this information down to the hardware.

3.3 PMU Architecture

Hardware power management performed at the memory controller has been previously proposed in [25, 45, 46]. Fine-grained monitoring and power-control mechanisms are ac-

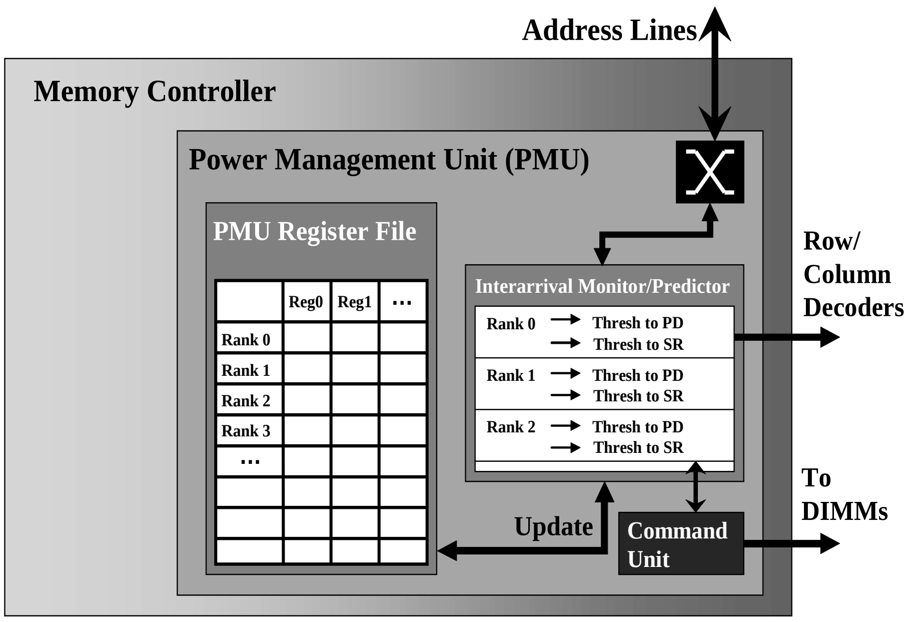

Address Lines
Memory Controller
Power Management Unit (PMU)
PMU Register File
Reg0
Reg1
...
Rank 0
Rank 1
Rank 2
Rank 3
...
Interarrival Monitor/Predictor
Rank 0 Thresh to PD Thresh to SR
Rank 1 Thresh to PD Thresh to SR
Rank 2 Thresh to PD Thresh to SR
Row/ Column Decoders
To DIMMs
Update
Command Unit

Figure 3.2: Architecture of a generic PMU.

complished by a separate power-management unit (PMU) within the memory controller. The PMU is typically implemented as a set of simple logic devices that (i) monitor memory accesses, (ii) predict threshold values to determine when to power down, (iii) determine which low-power state to transition, and (iv) instruct the memory controller to perform the actual power-down operations when certain conditions are met.

We show the schematic diagram of a generic PMU in a memory controller in Figure 3.2. It monitors memory accesses by snooping the address lines and keeps track of past memory access behavior in an internal register file, where the number of registers depends on how accurate we need the prediction logic to be and also the available chip area on the memory controller for laying out the PMU. Based on the observed memory traffic, a threshold value can then be derived to determine how much idle time should elapse from the time of the last memory access before putting a rank to a lower power state. When multiple energy-saving states are defined, one can derive multiple thresholds, each used to transition a rank to a different low-power state. In Figure 3.3, we show an example of how threshold value is used to transition a rank to a low-power state.

For each memory rank, the PMU will individually monitor memory accesses, keep access history and manage power. The reason for keeping a separate set of logics for

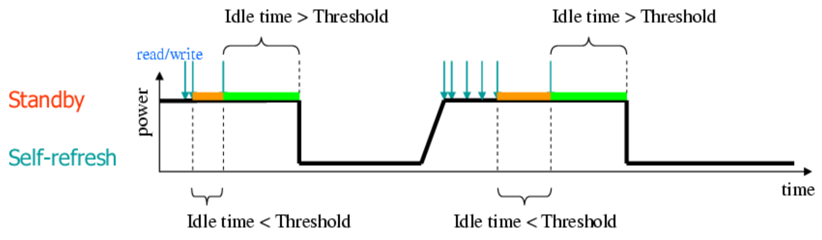


Figure 3.3: An example showing how threshold is used to manage power for the memory. Arrows indicate read or write memory accesses.

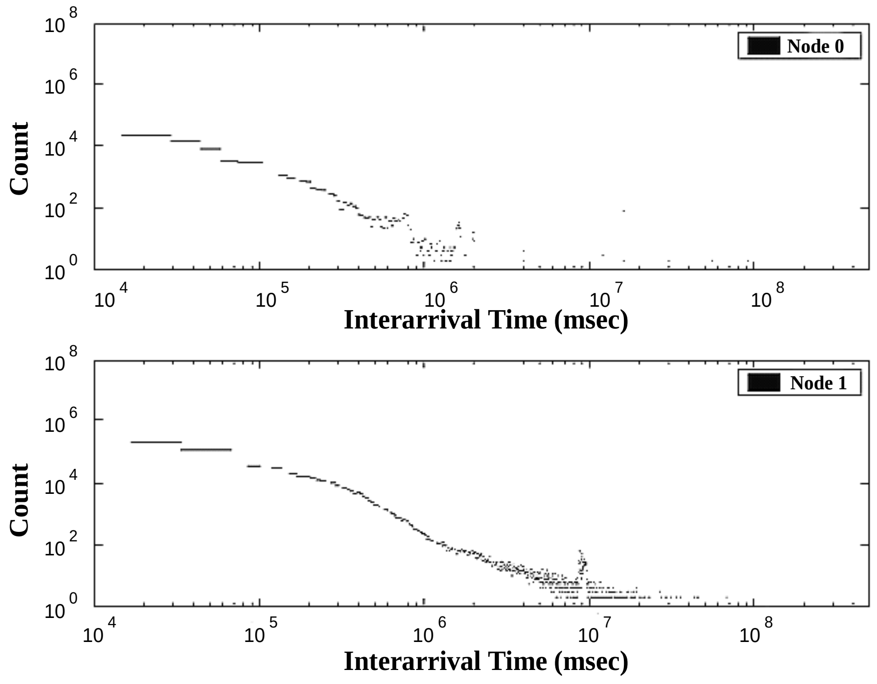


Figure 3.4: Inter-arrival time observed on two different ranks (or nodes).

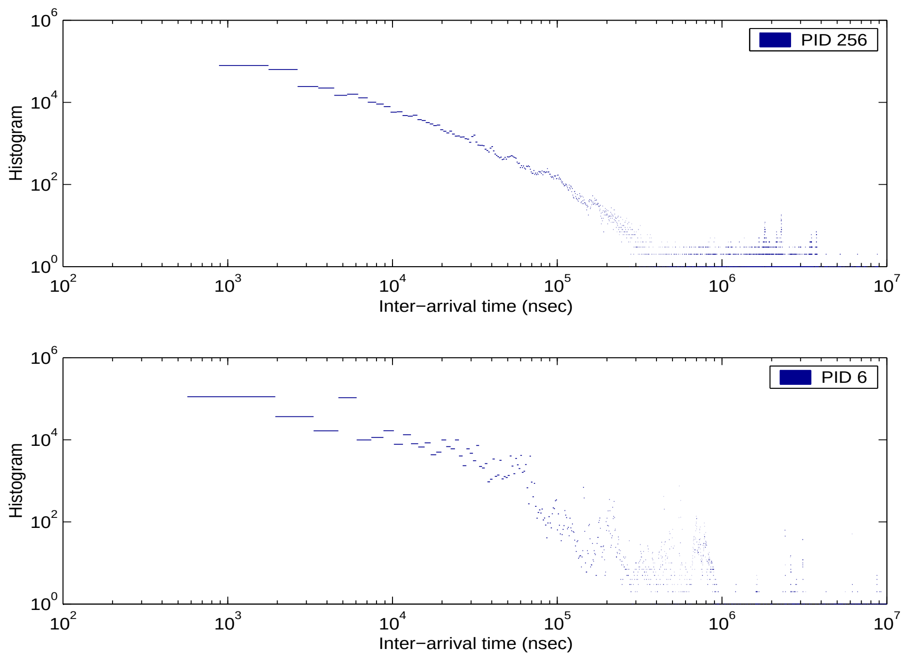
each rank is that one may be accessed very differently from others. To give an example, Figure 3.4 shows a histogram (in log scale) of interarrival times (in log scale) between consecutive memory accesses observed on two different ranks. It is clear from this figure that the access characteristics observed on these two ranks are distinct. On rank 0, we can observe that with most interarrival times being very short, nearly every memory access comes within 1 second after the previous one. On rank 1, however, there are many larger gaps (indicated by the heavier-tailed distribution) between consecutive memory accesses, suggesting that on this rank we have more energy-saving opportunities and also the fact that different power-down thresholds should be used for these two ranks to maximize energy savings on each. However, this *per-rank* implementation in the PMU would require additional circuitry which not only would increase manufacturing costs but also consume more energy. Later, we will show how to use the process-state information exported by the system software to reduce these additional costs.

3.3.1 Design of a Context-Aware PMU

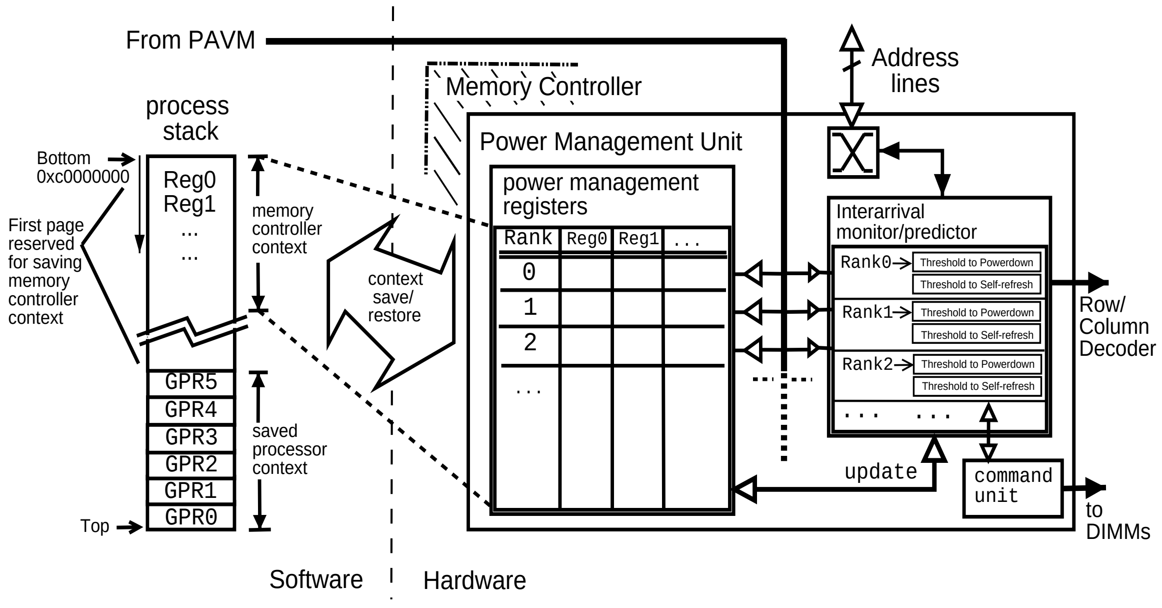
In the previous section, we illustrated the traditional PMU architecture. However, by monitoring memory traffic and performing power management at such a low level, we often lose sight on entities that are responsible for actually utilizing the memory—running processes and the operating system. This loss of information results in hardware frequently making short-sighted power management decisions, which can have significant impact on performance and energy consumption.

Our solution to this problem is to export process-state information down to the PMU layer so we can make it *context-aware*. This allows the PMU to monitor memory accesses on a per-process basis, and using which, it can customize power management policies tailored specifically to the currently executing process. This is very important as different processes may exhibit vastly different memory access behaviors, and therefore, customizing power management decisions for each process can significantly increase the decisions' effectiveness in conserving power. Furthermore, even if we have processes with similar memory access behavior, the way they access each of the memory ranks can be quite different (shown in Figure 3.5(a)). This is especially true for modern systems where virtual memory is extensively used and virtual pages can be mapped to arbitrary physical pages by the operating system.

Without having any understanding of the running processes, the observed memory traffic by the PMU is essentially "polluted", by all the processes that have accessed this rank in rapid successions (at a 10 ms or even an 1 ms quantum) as scheduled by the task scheduler. Therefore, the PMU will likely make inefficient power-management decisions based on an "average" access behavior observed from all the concurrent processes. We illustrate this by an example shown in Figure 3.6. In this example, Process 1 rarely accesses rank 0, whereas



(a)



(b)

Figure 3.5: (a) Inter-arrival time between consecutive memory requests observed in two different processes on the same memory rank. (b) Architecture of the process-aware PMU.

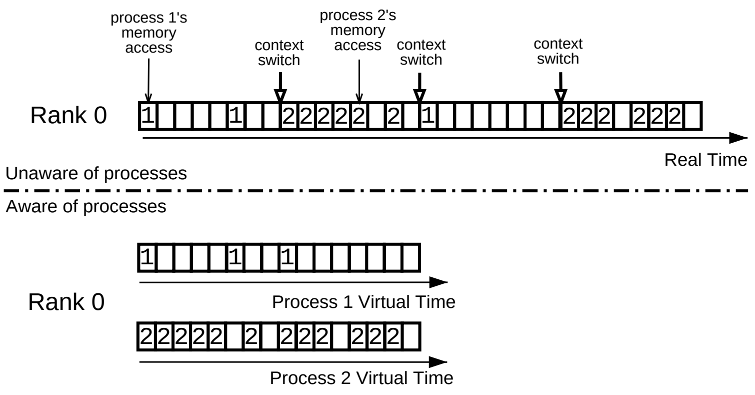


Figure 3.6: An example that gives some intuitions about why it is beneficial to make the memory controller context-aware.

Process 2 accesses this rank very frequently. If the PMU 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, 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 the per-rank power management, the memory controller is totally oblivious to the concept of a process, which ironically, has a strong impact on how the memory is being accessed and how it should be power managed.

The improvement to make the PMU context-aware can be very easily augmented with a small amount of hardware modifications and with some minor changes to the system software. On the software side, when context switching, in addition to saving the processor context (i.e., CPU registers) onto the stack of the switched-out process, we must, in parallel, also save the value of the history-keeping registers used by the PMU as shown in Figure 3.5(b) (Ignore the `PAVM line` for now). When this process is context switched back at a later time, we will need to restore both the processor and the PMU context associated with this process. The PMU context saving and 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 design will allow the memory controller to manage power tailored specifically to each process because the PMU is now capable of making power management decisions based purely on each process's own memory access behavior.

3.3.2 Cooperation with PAVM

As indicated in Section 3.3, even though only a small amount of modifications is needed to implement the aforementioned energy-conserving mechanisms in hardware, the additional hardware does not come for free—a non-negligible amount of additional power is dissipated as a result. To amortize this cost, PAVM can inform the PMU which ranks are actively used by the running process so the PMU can completely gate off all the monitor/predictor circuits and history-keeping registers for those inactive ranks without affecting its effectiveness. This information is passed down from the `PAVM control` line shown in Figure 3.5(b). Since it is the characteristic of PAVM to pack the memory footprint of each process onto as fewer ranks as possible, the software technique will help to minimize the number of monitor/predictor circuits that are needed in the full power mode.

This cooperation with PAVM also has 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 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, we can direct PAVM to instruct the memory controller to power up the active ranks of the to-be-run process as early as possible so some resynchronization penalties could be minimized or even avoided.

3.3.3 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 easily and accurately by the PMU to effective power management decisions. This requires only a small amount of changes in the PMU hardware and the system software. Additionally, we also proposed techniques that allow PAVM to assist the PMU in (i) amortizing the energy cost of the additional hardware in the PMU and (ii) reducing wake-up latency due to cold cache misses, and thus allowing more efficient use of the energy.

3.4 Evaluation

We will now evaluate the software-directed technique (labeled as HW–SW for short) and compare it against some of the previously-proposed techniques. We use the same simulation environment and evaluation methodology as that was described in Section 2.5.1. Section 3.4.1 and Section 3.4.2 provide detailed simulation results from running a synthetic and some SPEC benchmarks (SPECjbb2000 and SPECcpu2000), respectively.

3.4.1 Synthetic Workload

We first use a synthetic workload consisted of two streaming processes. Assume all the memory accesses of the first process miss in the cache and are fetched from the main memory, and the second process's all hit in the cache. This synthetic workload is not meant to be realistic, but through this simple example, we can illustrate the potential benefit of making the memory controller context-aware. Furthermore, we can also see more clearly the energy and performance implications of various power management techniques by breaking the total power dissipated by the main memory into various components.

Synthetic Workload Results

The machine configuration we used to run the synthetic workload is shown in Table 3.1. It is the same as that was shown in Table 2.4, except that the memory subsystem is reduced to a single 64 MB rank. This allows us to interpret the results of the synthetic workload much more easily. In this workload, we evaluated 5 power management techniques: No Power Management, Immediate Powerdown, Immediate Self Refresh, Dynamic Hardware

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 64MB (64Mbx1)
Linux Kernel	2.6.5-rc3 w/ PAVM patch

Table 3.1: System parameters used in Mambo for running the synthetic workload. All cache lines are 128 Bytes long.

Technique, and Software-Directed Hardware Technique, and they are described in the following section. Note, we omitted PAVM in this synthetic workload because it does not make much sense to run PAVM on a system with only one memory rank as it would not yield any power savings. As many small embedded systems are often equipped with only one rank, it makes PAVM unsuitable to use for these types of systems and hardware-based techniques more applicable.

- **No Power Management (NOPM):** Here, no power management technique is used, and ranks are put to the highest ready state when idling. We use this as the baseline for our comparison.

- **Immediate Powerdown (IPD):** The simplest form of hardware-controlled power management. This is a static technique where the memory controller automatically transitions a rank to Powerdown state immediately after all the memory requests have completed.

- **Immediate Self Refresh (ISR):** Same as IPD, but transitions to Self Refresh instead of Powerdown.

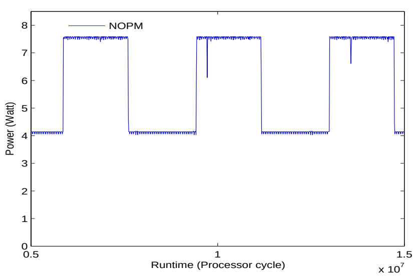
- **Dynamic Hardware Technique (HW):** This is a purely hardware-controlled power management technique. It monitors past memory accesses, and based on which, dynamically makes predictions on after how long of an idle period, it should transition a rank to Self Refresh (threshold prediction algorithm is shown in Figure 2.9). Transition to Powerdown has a zero threshold, which was previously shown to be most efficient [25].

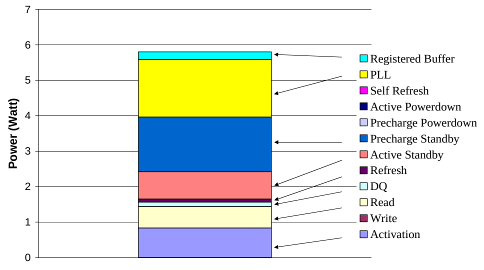
- **Software-Directed Hardware Technique (HW–SW):** This is the software-directed hardware technique that we have proposed in Section 3.3.

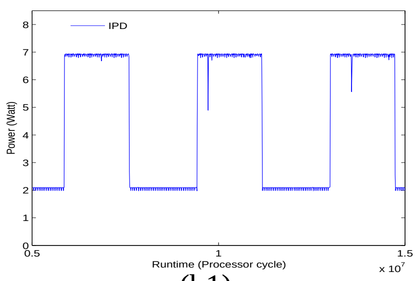
The two streaming processes are scheduled in an interleaved-manner by the Linux task scheduler. Without any power management, a sample of the instantaneous power dissipated by the memory is shown in Figure 3.7(a1), where one can clearly see when each process is scheduled. Furthermore, we break the average power down to its individual components, which is shown in Figure 3.7(a2). Power used by activation, read, write operations and data queues are due to DRAM device doing useful work and cannot be further reduced. Although there are means to reduce this power, e.g., open/close-page policy, rank interleaving, sub-banking, etc., they are beyond the scope of our work. Instead, we focus on finding ways and opportunities to reduce the amount of idle power wasted when no work is being done, thus having less impact on performance. From the previous figure, we can see that most of this idle power is dissipated in the Precharge Standby mode, Active Standby mode, and by the PLL and the registers.

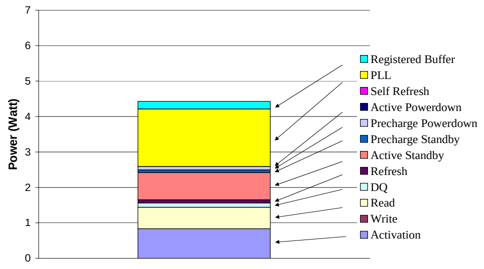
First, we consider the simplest static hardware techniques, which will put memory ranks to either Powerdown or Self Refresh mode immediately at the end of each memory request, if and only if there are no outstanding memory accesses on any of the banks. We call these techniques Immediate Powerdown (IPD) and Immediate Self Refresh (ISR), respectively, and the results are shown in Figures 3.7(b1–b2) and Figures 3.7(c1–c2). As we can see from these figures, power reduction opportunities arise when the low memory-intensive process starts to execute. IPD can significantly reduce power dissipated in Standby mode, whereas ISR can achieve additional energy savings by also powering down the PLL and the registers, although at a severe performance penalty. We will look at the performance implications in more details shortly.

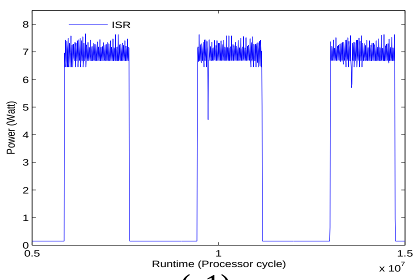
Next, Figure 3.7(d1) shows the effect on power when power-management decisions are dynamically made by the hardware (e.g., PMU in the memory controller). The PMU keeps

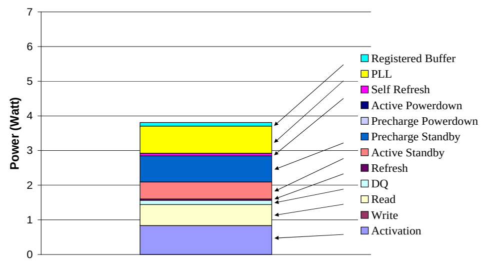

(a1)

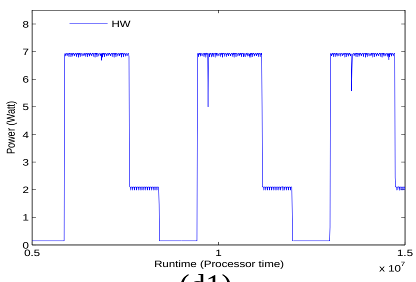

(a2)

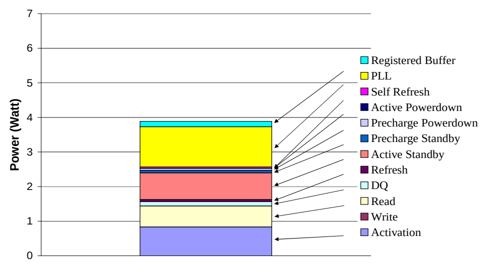

(b1)

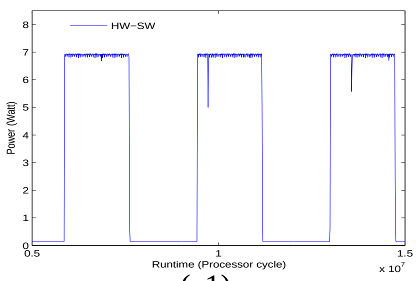

(b2)


(c1)


(c2)


(d1)


(d2)


(e1)

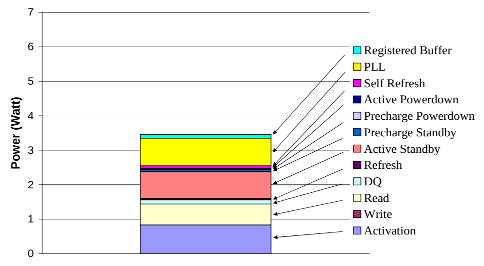

(e2)

Figure 3.7: The first column gives the instantaneous power for a zoomed-in portion of the synthetic workload when using (a) NOPM, (b) IPD, (c) ISR, (d) HW, and (e) HW–SW. The second column shows the breakdown of the average power dissipated.

Total Simulated Cycles	1.89e9 cycles				
Number of Read	7,429,188				
Number of Writes	333				
	No Power Management	Immediate Powerdown	Immediate Self Refresh	HW	HW–SW
Average Power	5.80 W	4.43 W	3.81 W	3.88 W	3.45 W
Average Response Time	60.53 ns	65.59 ns	555.15 ns	66.03 ns	66.08 ns
Normalized Runtime	1.000	1.032	4.110	1.035	1.035
Delayed Accesses Due to PD	0	7,296,600	0	7,289,002	7,288,920
Delayed Accesses Due to SR	0	0	395,341	538	646

Table 3.2: Summary of the synthetic benchmark.

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, our result shows that it can only save 12.29% more energy than IPD because when the hardware tries to make power-management decisions based only on its observation of the past memory access behavior, it will get confused when two processes with very different access behaviors are accessing the same rank in an interleaved-manner.

One can argue that if $\mathrm{Th}_{\mathrm{Max}}$ is reduced, we can adapt more quickly. However, shrinking this parameter is a double-edged sword: having better adaptability minimizes the problems caused by constant context-switching but runs at a risk of over-aggressively predicting threshold values when there are transient behaviors at runtime. Adjusting this parameter will have a positive effect for this synthetic workload, but for realistic workloads, it can cause more harm than good. Furthermore, as more and more systems are switching to smaller scheduling quanta (e.g., from 10 ms to 1 ms or even smaller) to increase responsiveness in the system, it will make the hardware predictor's job even more difficult.

Finally, Figure 3.7(e1) shows that if the system software can inform the PMU 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 of keeping past access history for each of the processes and saving/restoring history-keeping registers at each context switch. From 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. Moreover, unlike the hardware-only technique (HW), the HW–SW technique will not be affected when the scheduling quantum becomes smaller over time.

In Table 3.2, we summarize power and performance results of the different power management techniques for the synthetic workload. Even though ISR saves the most amount of power (34.31%), its performance impact (310.97% slowdown) makes it prohibitive to use in practice, therefore, we do not consider this technique further. All other techniques have similar performance degradations (IPD 3.19% slowdown, HW 3.46%, and HW–SW 3.49%), but HW–SW is the most energy efficient followed by HW technique. Specifically, in this case, HW–SW saves 11.03% more energy than HW. The reason why HW–SW consumes less energy than the HW approach is because it can more accurately predict threshold values to go to Self Refresh due to its awareness of processes. Using this simple example, we have shown the benefits of exporting process information from system software to the memory controller hardware. More accurate and aggressive power management decisions can be made as a result.

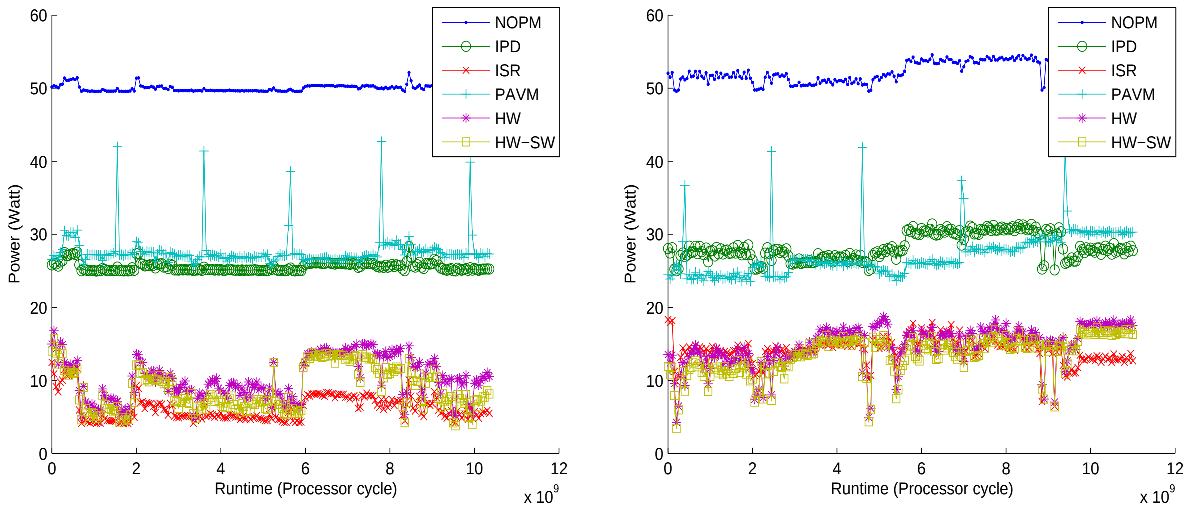
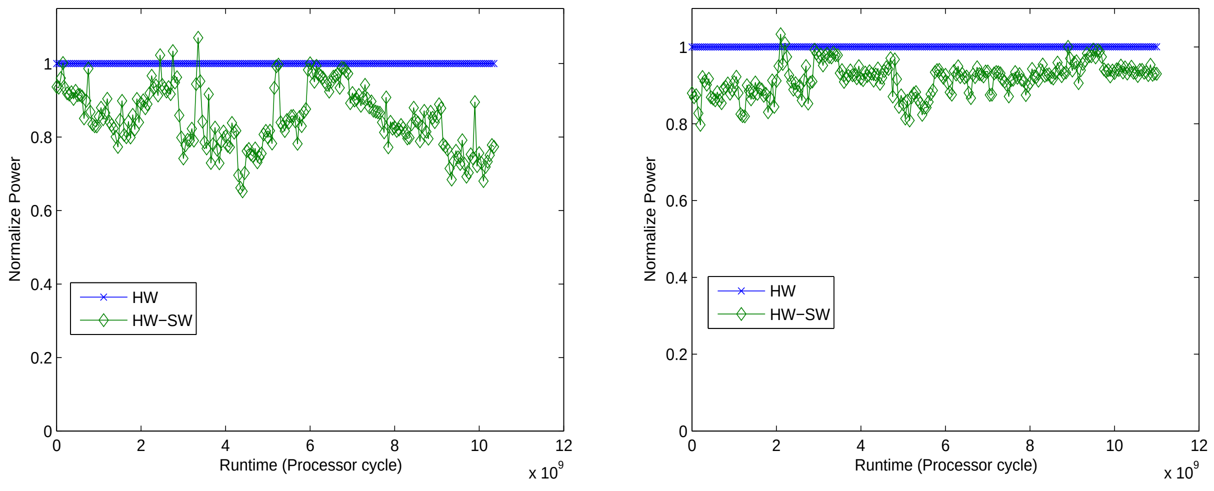


Figure 3.8: (a) Instantaneous power for the low memory-intensive SPEC workload. (b) Instantaneous power for the high memory-intensive SPEC workload.



(a) (b)

Figure 3.9: Comparing power dissipated by HW and HW–SW techniques. Power is normalized to HW's power dissipation for (a) low memory-intensive workload and (b) high memory-intensive workload.

3.4.2 SPEC Benchmarks

In the previous section, we used a synthetic benchmark to give some basic intuitions on why software-directed technique is superior. In this section, we use more realistic benchmarks to evaluate how it would perform and measure up to the other techniques. We use the same benchmarks we used for our evaluation from the last chapter (Section 2.5.2).

	No Power Management	IPD	ISR	PAVM	HW-only	HW–SW
Energy Consumption	275.02 J	143.15 J	68.26 J	152.27 J	58.35 J	50.38 J
Average Power	49.97 W	25.54 W	6.23 W	27.64 W	10.24 W	8.81 W
Average Response Time	72.01 ns	82.17 ns	697.60 ns	72.29 ns	90.14 ns	94.38 ns
Normalized Runtime	1.000	1.018	2.107	1.001	1.032	1.039
Delayed Accesses Due to PD	0	6,781,614	0	0	6,669,095	6,642,938
Delayed Accesses Due to SR	0	0	2,701,345	70	74,094	115,577

Table 3.3: Summary of the low memory-intensive workload.

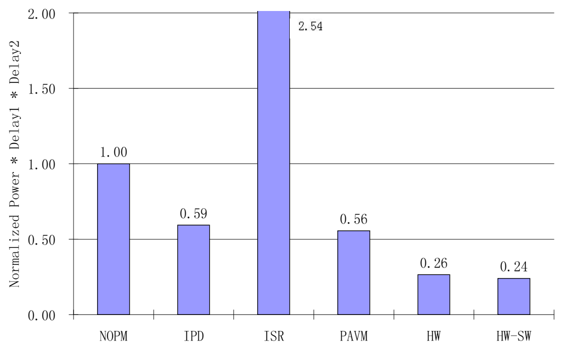
SPEC Benchmarks Results

Figure 3.8(a) and Figure 3.8(b) show the instantaneous power dissipated throughout the entire run of the low memory-intensive and high memory-intensive workloads, respectively, using (i) no power management, (ii) IPD, (iii) ISR, (iv) PAVM, (v) HW, and (vi) HW–

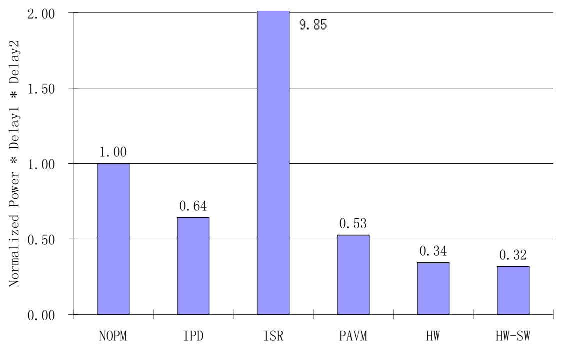
	No Power Management	IPD	ISR	PAVM	HW-only	HW–SW
Energy Consumption	482.11 J	282.79 J	743.42 J	252.24 J	149.21 J	137.72 J
Average Power	52.15 W	28.30 W	14.47 W	27.15 W	14.79 W	13.58 W
Average Response Time	84.13 ns	92.19 ns	537.45 ns	84.59 ns	93.21 ns	93.75 ns
Normalized Runtime	1.000	1.081	5.558	1.005	1.091	1.097
Delayed Accesses Due to PD	0	20,648,953	0	0	20,603,033	20,560,901
Delayed Accesses Due to SR	0	0	4,954,357	181	27,559	56,335

Table 3.4: Summary of the high memory-intensive workload.

SW. In both workloads, ISR, HW, and HW–SW saved much more energy than the rest. However, due to ISR's prohibitively high performance penalty, it is not very interesting to us. A more clear comparison between HW and HW–SW techniques is shown in Figure 3.9. In this figure, we show the normalized power dissipation with respect to HW. It shows a clear power reduction from using the additional system state information to make power management decisions in the memory controller hardware—8.18–14.02% more power can be saved as a result. In terms of runtime, HW–SW is within 1% of HW.



(a)



(b)

Figure 3.10: Comparing power management techniques using the $\mathrm{Power} \times \mathrm{Delay1} \times \mathrm{Delay2}$ metric for (a) the low memory-intensive workload and (b) the high memory-intensive workload.

Several metrics have been proposed previously to evaluate the overall goodness of a power management technique by taking into account of both energy and performance impacts. $\mathrm{Power} \times \mathrm{Delay}$ and $\mathrm{Power} \times \mathrm{Delay}^2$ are the most popular metrics that have been used. The reason for the squared term in the latter metric is to put more weight on performance as in many systems performance is still the dominant metric to optimize. In this work, we use a variant of the second metric—$\mathrm{Power} \times \mathrm{Delay1} \times \mathrm{Delay2}$, where Delay1 is the normalized runtime and Delay2 is the normalized average response time. The results are shown in Figures 3.10. Again, we see HW, HW–SW are better than the other techniques.

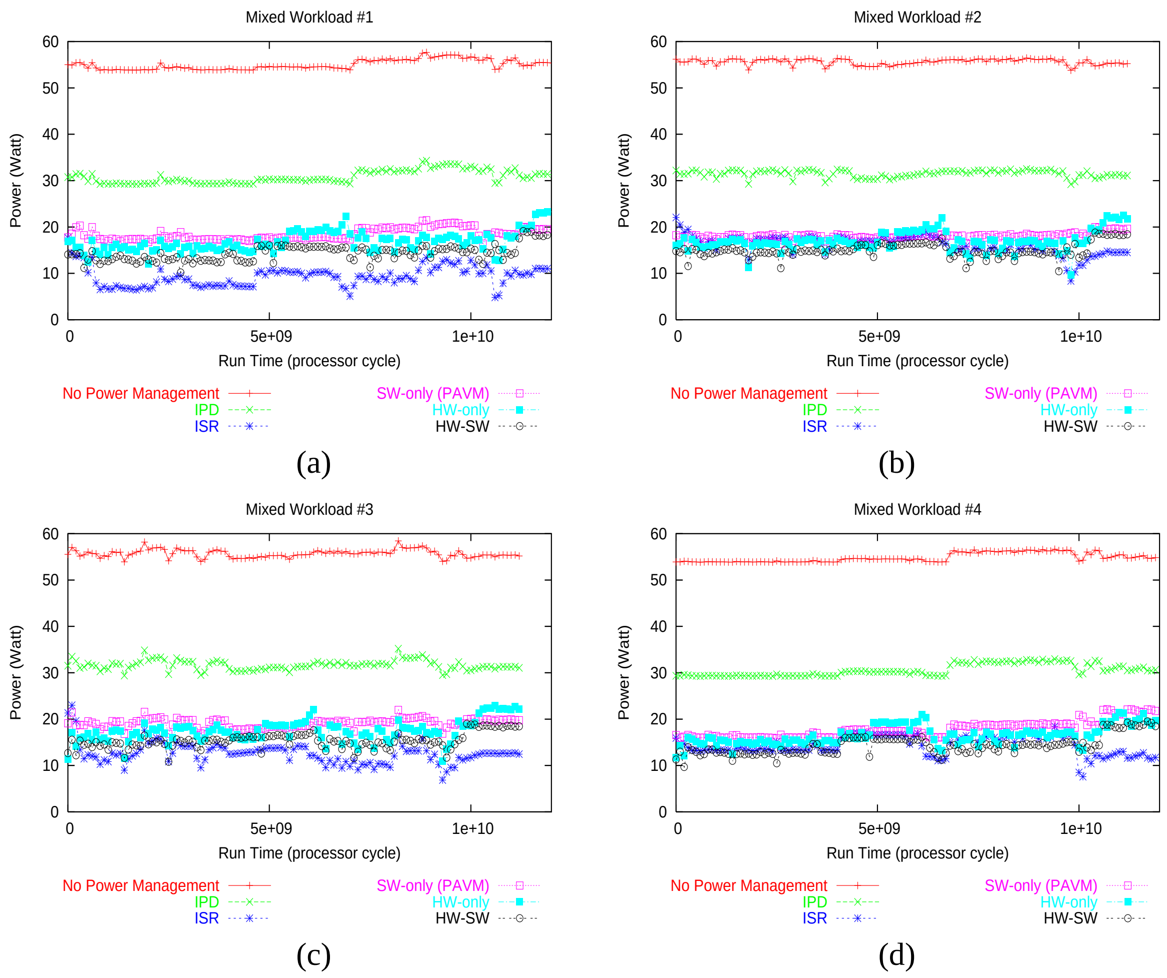


Figure 3.11: (a) Instantaneous power for the Mixed #1 workload. (b) Instantaneous power for the Mixed #2 workload. (c) Instantaneous power for the Mixed #3 workload. (d) Instantaneous power for the Mixed #4 workload.

For completeness, we also show the results of running all other pair-wise combinations of the four SPEC benchmarks with SPECjbb. Table 3.5 shows the runtime information of these workloads. Figure 3.11 shows the instantaneous power during the entire experimental run when different power-management techniques are applied. Tables 3.6, 3.7, 3.8, and 3.9 summerize the average power and average response time.

Benchmarks	Total Runtime (processor cycles)	% of Total Runtime	Read Operations	% of All Reads	Write Operations	% of All Writes	Context Switches
Mixed workload #1							
SPECjbb process 1	710,534,166	5.8%	621,551	2.61%	188,567	1.48%	1,321
SPECjbb process 2	1,686,377,599	13.8%	1,888,701	7.93%	936,527	7.33%	793
SPECjbb process 3	764,435,400	6.3%	707,345	2.97%	288,737	2.26%	439
SPECjbb process 4	719,960,469	5.9%	693,640	2.91%	263,203	2.06%	1,106
SPECjbb process 5	500,601,192	4.1%	609,364	2.56%	184,728	1.45%	1,119
SPECjbb process 6	587,960,139	4.8%	668,505	2.81%	184,658	1.44%	1,267
SPECjbb process 7	634,488,511	5.2%	729,032	3.06%	250,955	1.96%	1,079
SPECjbb process 8	409,730,082	3.4%	622,097	2.61%	177,900	1.39%	1193
256.bzip2	2,672,678,890	21.9%	3,658,348	15.36%	2,157,427	16.88%	1,160
181.mcf	3,264,133,644	26.8%	13,323,656	55.94%	8,068,578	63.12%	1,160
Total (benchmarks)	11,950,900,092	98.0%	23,522,239	98.77%	12,701,280	99.37%	10,637
Total (all observed)	12,189,976,977	100.0%	23,816,334	100.00%	12,782,238	100%	15,572
Mixed workload #2							
SPECjbb process 1	580,523,422	5.1%	773,649	1.69%	196,492	3.42%	412
SPECjbb process 2	1,582,313,571	13.96%	1,890,848	4.12%	404,719	7.05%	945
SPECjbb process 3	664,265,936	5.9%	774,605	1.69%	175,646	3.06%	1,145
SPECjbb process 4	414,782,393	3.7%	553,536	1.21%	142,946	2.49%	1,181
SPECjbb process 5	510,552,742	4.5%	627,382	1.37%	164,955	2.87%	1,108
SPECjbb process 6	883,810,143	7.8%	921,784	2.01%	218,601	3.80%	1,378
SPECjbb process 7	841,253,742	7.4%	947,265	2.06%	224,237	3.90%	973
SPECjbb process 8	590,082,292	5.2%	840,999	1.83%	203,887	3.55%	1,249
186.crafty	2,627,355,852	23.2%	1,407,925	3.07%	290,246	5.05%	1,159
179.art	2,423,214,001	21.4%	36,766,152	80.13%	3,696,418	64.35%	1,160
Total (benchmarks)	11,147,165,094	98.4%	45,504,145	99.18%	5,718,147	99.54%	10,710
Total (all observed)	11,332,601,664	100%	45,882,216	100.00%	5,744,542	100.00%	15,610
Mixed workload #3							
SPECjbb process 1	547,366,485	4.8%	716,863	1.53%	192,508	2.58%	817
SPECjbb process 2	933,866,021	8.2%	1,046,746	2.25%	258,570	3.47%	1,176
SPECjbb process 3	489,136,734	4.3%	631,695	1.36%	160,219	2.15%	856
SPECjbb process 4	1,711,277,898	15.0%	1,975,128	4.24%	495,406	6.64%	867
SPECjbb process 5	413,636,931	3.6%	679,162	1.46%	167,503	2.24%	284
SPECjbb process 6	887,571,915	7.8%	1,012,084	2.17%	278,087	3.73%	1,135
SPECjbb process 7	422,697,140	3.7%	762,833	1.64%	195,442	2.62%	961
SPECjbb process 8	954,661,929	8.4%	1,151,594	2.47%	276,698	3.71%	1,071
256.bzip2	2,494,274,438	21.9%	3,499,475	7.51%	1,266,475	16.97%	1,094
179.art	2,315,027,158	20.3%	34,842,138	74.78%	4,138,620	55.46%	1,094
Total (benchmarks)	11,169,516,649	98.1%	46,317,718	99.41%	7,429,528	99.56%	9,355
Total (all observed)	11,388,771,125	100%	46,590,620	100.00%	7,462,319	100.00%	14,261
Mixed workload #4							
SPECjbb process 1	529,160,789	4.4%	781,005	3.79%	232,881	2.24%	739
SPECjbb process 2	608,968,797	5.1%	717,720	3.48%	213,329	2.06%	1,134
SPECjbb process 3	899,079,878	7.5%	773,773	3.76%	252,860	2.44%	504
SPECjbb process 4	1,648,886,947	13.7%	2,000,809	9.71%	962,684	9.28%	783
SPECjbb process 5	636,026,872	5.3%	623,331	3.03%	191,727	1.85%	488
SPECjbb process 6	854,446,496	7.1%	862,621	4.19%	254,023	2.45%	662
SPECjbb process 7	612,802,410	5.1%	687,737	3.33%	196,147	1.89%	930
SPECjbb process 8	549,496,460	4.6%	739,180	3.59%	204,938	1.98%	906
186.crafty	2,496,272,177	20.7%	1,015,078	4.93%	581,945	5.61%	1,077
181.mcf	2,976,933,794	24.7%	12,123,533	58.86%	7,216,094	69.57%	1,077
Total (benchmarks)	11,712,074,620	97.1%	20,324,787	98.68%	10,317,628	99.47%	8,300
Total (all observed)	12,056,225,959	100%	20,596,848	100.00%	10,372,137	100.00%	13,159

Table 3.5: Summary of the mixed workloads. SPECjbb is ran with 8 warehouses, each spawned as a separate Java thread.

	No Power Management	IPD	ISR	SW-only (PAVM)	HW-only	HW–SW
Energy Consumption	419.13 J	234.84 J	72.16 J	140.87 J	128.88 J	109.88 J
Average Power	55.01 W	30.82 W	9.47 W	18.49 W	16.92 W	14.42 W
Average Response Time	123.62	135.58	959.17	137.21	137.25	138.98
Delayed Accesses Due to PD	0	14,513,726	0	12,477,003	14,480,901	12,465,340
Delayed Accesses Due to SR	0	0	4,209,023	905	14,314	6,069

Table 3.6: Summary of Mixed workload #1. All timings are in unit of cycles.

	No Power Management	IPD	ISR	SW-only (PAVM)	HW-only	HW–SW
Energy Consumption	394.68 J	223.91 J	115.95 J	129.07 J	121.34 J	107.14 J
Average Power	55.72 W	31.61 W	16.37 W	18.22 W	17.13 W	15.13 W
Average Response Time	143.28	152.90	931.37	154.57	155.07	155.17
Delayed Accesses Due to PD	0	12,324,762	0	10,993,260	12,293,860	10,981,748
Delayed Accesses Due to SR	0	0	3,934,854	238	15,032	5,599

Table 3.7: Summary of mixed workload #2. All timings are in unit of cycles.

3.4.3 Sensitivity Analysis

In the previous section, we set the window size of the threshold predictor to 500 μsec. In this section, we will study how window size affects performance and energy in the HW-only and HW–SW techniques. Since PAVM does not rely on any hardware monitor and predictor, it is not affected by this parameter. In Table 3.10, we show results when varying the window size to be 10, 500, and 5000 μsec.

As expected, if we decrease the window size, we can adapt more closely to the current memory access pattern, which results in more power savings. However, because we are looking within a smaller time window, we are more prone to over-compensate or to more aggressively set threshold values. This results in a significant increase in response time when we shrink the window size from 500 μsec to 10 μsec.

As we have shown previously, at the window size of 500 μsec, the HW–SW technique uses 11.7–16.7% less power than HW-only. When we shrink the window size to 10 μsec, the hardware can more quickly adapt, and therefore, the benefit of communicating information from the system software diminishes. Our result shows that HW–SW saves only 6.6–10.3% more power than the HW-only approach after reducing the window size to 10 μsec. However, this additional power reduction comes at a hefty performance penalty—an additional performance degradation of 13.4–74.8%, which most likely will not be acceptable in many systems. On the other hand, there is not much performance benefit in increasing the

	No Power Management	IPD	ISR	SW-only (PAVM)	HW-only	HW–SW
Energy Consumption	397.15 J	225.84 J	91.03 J	137.11 J	125.15 J	110.08 J
Average Power	55.80 W	31.73 W	12.79 W	19.26 W	17.58 W	15.47 W
Average Response Time	139.64	149.42	932.27	150.23	151.31	152.58
Delayed Accesses Due to PD	0	14,760,164	0	12,651,574	14,721,606	12,637,407
Delayed Accesses Due to SR	0	0	4,297,173	348	16,944	6,091

Table 3.8: Summary of Mixed workload #3. All timings are in unit of cycles.

	No Power Management	IPD	ISR	SW-only (PAVM)	HW-only	HW–SW
Energy Consumption	413.21 J	230.49 J	108.75 J	135.02 J	124.49 J	108.24 J
Average Power	54.84 W	30.59 W	14.43 W	17.92 W	16.52 W	14.36 W
Average Response Time	123.05	134.75	975.69	136.72	136.26	137.35
Delayed Accesses Due to PD	0	12,318,649	0	10,961,327	12,292,630	10,951,530
Delayed Accesses Due to SR	0	0	3,961,286	838	12,178	5,453

Table 3.9: Summary of Mixed workload #4. All timings are in unit of cycles.

window size to 5000 μsec, where we actually increased the average power dissipation by 12.5–18.7% while only improved the average response time by 0.3–1.2%.

3.5 Conclusions

In this chapter, we proposed a novel power management technique that makes use of cooperation between the system software and the memory controller hardware. It can significantly improve the accuracy of the PMU's threshold prediction logics. Using a full-system simulator, our HW–SW cooperative approach is shown to consume 8.18–14.02% less energy than the hardware-only approach with similar performance. It is also shown to consume 49.98–68.13% less energy than the software-only approach.

Alternative to this software-directed hardware power management technique, the hardware can provide feedback to the system software to create additional energy saving opportunities. For example, the hardware can inform the OS how frequently (hot) or infrequently (cold) each of the physical pages is 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. This is the main focus of Chapter 4.

Additionally, we would also like to explore direct cooperation between applications

	Window Size	HW-only		HW–SW	
		Avg Power	Avg Rsp Time	Avg Power	Avg Rsp Time
Low-Mem	10 μs	8.68 W	225.43 cycles	7.84 W	216.61 cycles
	500 μs	16.18 W	128.96 cycles	13.48 W	130.47 cycles
	5000 μs	19.00 W	131.64 cycles	17.00 W	128.88 cycles
High-Mem	10 μs	12.65 W	165.45 cycles	11.35 W	167.03 cycles
	500 μs	18.71 W	145.94 cycles	16.07 W	148.64 cycles
	5000 μs	21.05 W	145.01 cycles	18.39 W	148.22 cycles
Mixed#1	10 μs	10.14 W	175.32 cycles	9.21 W	173.22 cycles
	500 μs	16.92 W	137.25 cycles	14.42 W	138.93 cycles
	5000 μs	20.09 W	136.65 cycles	17.83 W	138.27 cycles
Mixed#2	10 μs	11.05 W	180.16 cycles	10.32 W	177.36 cycles
	500 μs	17.13 W	155.07 cycles	15.13 W	155.17 cycles
	5000 μs	19.35 W	154.46 cycles	17.73 W	154.62 cycles
Mixed#3	10 μs	11.16 W	176.25 cycles	10.31 W	175.53 cycles
	500 μs	17.58 W	151.31 cycles	15.47 W	152.58 cycles
	5000 μs	20.47 W	151.48 cycles	18.67 W	152.03 cycles
Mixed#4	10 μs	10.09 W	182.30 cycles	9.35 W	179.58 cycles
	500 μs	16.52 W	136.26 cycles	14.36 W	137.35 cycles
	5000 μs	18.56 W	136.09 cycles	17.32 W	136.78 cycles

Table 3.10: Comparing HW-only against HW–SW when varying the window size.

and the PMU. As applications themselves know more about their future memory access behavior than the OS. Such information will be valuable to the memory controller in its prediction logics, and thus, can be used to further enhance the proposed system.

CHAPTER 4

Active Memory-Traffic Reshaping

4.1 Motivation

In the previous chapter, we have shown that hardware techniques can save a significant amount of energy by using fine-grained monitoring and power-control mechanisms to observe and take advantage of the idle periods in memory ranks. However, not all idle periods can be exploited because state transitions take a non-negligible amount of time and energy. Only idle periods longer than the *break-even time* [45] are beneficial to transition to lower power states. The break-even times of different low-power states can vary significantly, and deeper power-saving states usually have longer break-even times as more components are disabled in these states and would take more time and energy to re-enable them before any new memory requests can be serviced. Therefore, to efficiently utilize the deeper power-saving states, having long and contiguous idle periods in memory traffic is essential.

Unfortunately these long idle periods are not commonly found in realistic workloads as physical memory is usually randomly accessed and driven completely by the current process's execution. As existing power-management techniques only passively monitor memory traffic, the deeper power-saving states are rarely fully exploited. In this chapter, we propose a new technique that minimizes short and unusable idle periods for power savings by aggregating them to create longer ones. As a result of reshaping the memory traffic in such a way, existing power management techniques are able to make better use of the idleness in the memory, thus saving more energy. Lebeck *et al.* [25] briefly mentioned

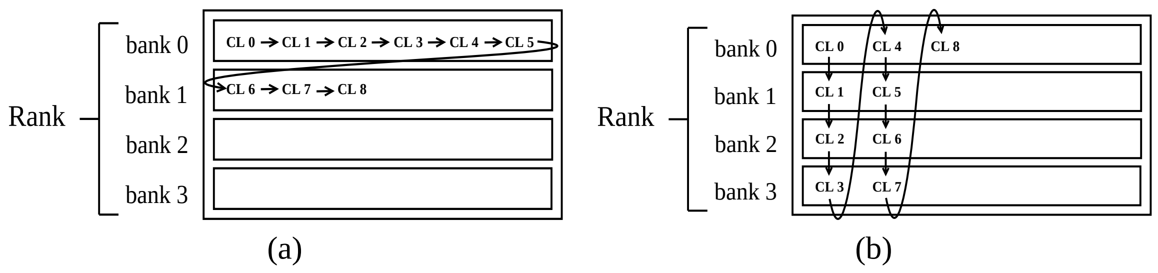


Figure 4.1: (a) Bank-fill interleave: consecutive cache lines (CL) are sequenced on the same bank until the bank is full. (b) Bank-spread interleave: consecutive cache lines are round-robined across the banks.

a frequency-based technique that is similar to our work, but they failed to recognize how such technique can be used to complement existing power-management techniques. Furthermore, unlike their work, we propose a practical technique that could be implemented in real systems using conventional operating systems and hardware. A more thorough evaluation is also presented here to fully assess the benefits and problems of this technique.

The rest of this chapter is organized as follows. Section 4.2 first discusses memory interleave schemes and their effect on power management. Section 4.3 gives some general intuitions about why passive power management is inefficient and how we can benefit from active management. Section 4.4 describes a practical technique that we can use to reshape memory traffic to our benefits. In Section 4.5, we evaluate the pros and cons of this technique using SPEC benchmarks. Finally, in Section 4.6, we conclude and highlight some possible future works.

4.2 Memory Interleaving

In this section, we describe how different interleave schemes can be applied to various levels of the memory hierarchy. We look at two types of interleave schemes—*spread* and *fill*, which are already implemented in many existing server systems. Bank is the lowest level where memory interleave can be applied. The two ways to interleave, *bank-fill* and *bank-spread*, are shown in Figure 4.1. In bank-fill interleave, consecutive cache lines are placed sequentially next to each other to fill up the whole bank. Once a bank is full, the next cache line is placed at the beginning of the next bank. In bank-spread interleave,

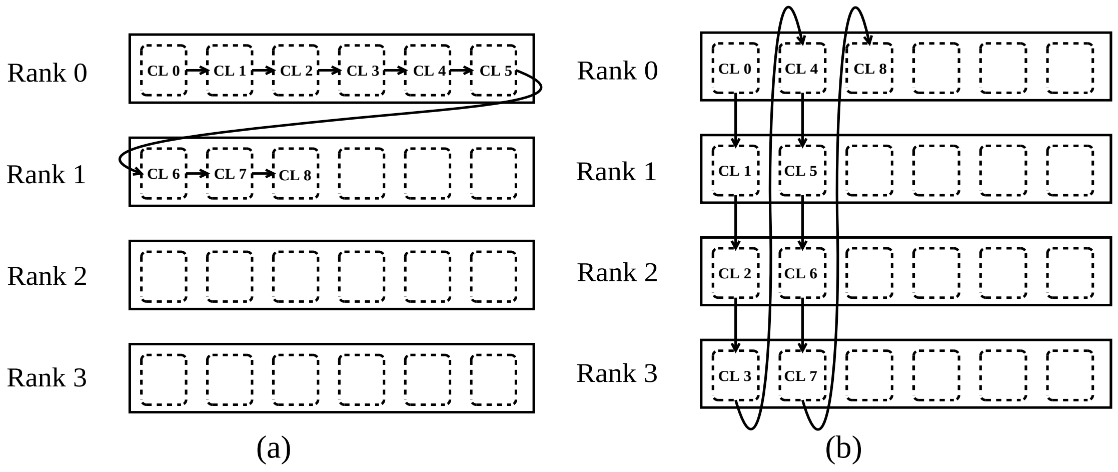


Figure 4.2: (a) Rank-fill interleave: consecutive cache lines (CL) are sequenced on the same rank until the rank is full. (b) Rank-spread interleave: consecutive cache lines are round-robined across the ranks.

consecutive cache lines are placed round-robin across all banks within a rank. In terms of performance, bank-spread is generally better than bank-fill. The reason is that due to data locality, sequential accesses are very common (i.e., affinitive cache lines are likely to be accessed closely in time). Bank-spread performs much better than bank-fill in sequential accesses because in the case of bank-spread, these sequential accesses can be serviced simultaneously by all the banks in the rank, whereas in the case of bank-fill, these sequential accesses will be serviced by only a single bank, thus resulting in longer queuing delays and larger average response time. Furthermore, since the smallest unit in power management in the memory is a rank, we cannot exploit these idle banks to save additional energy. Therefore, not only bank-fill negatively impacts performance due to its inability to use the all banks efficiently, but it also has no energy benefits. In the rest of the chapter, we will only consider bank-spread interleave.

Similarly, spread and fill interleave schemes can also be applied at the rank level, and they are orthogonal to the interleave schemes used at the bank level. *Rank-fill* and *Rank-spread* interleaves are shown in Figure 4.2. Again, spread interleave is generally better in performance as it can more efficiently utilize all the ranks. However, in terms of power, fill tends to be more efficient. This can be shown in an example in Figure 4.3. We show the

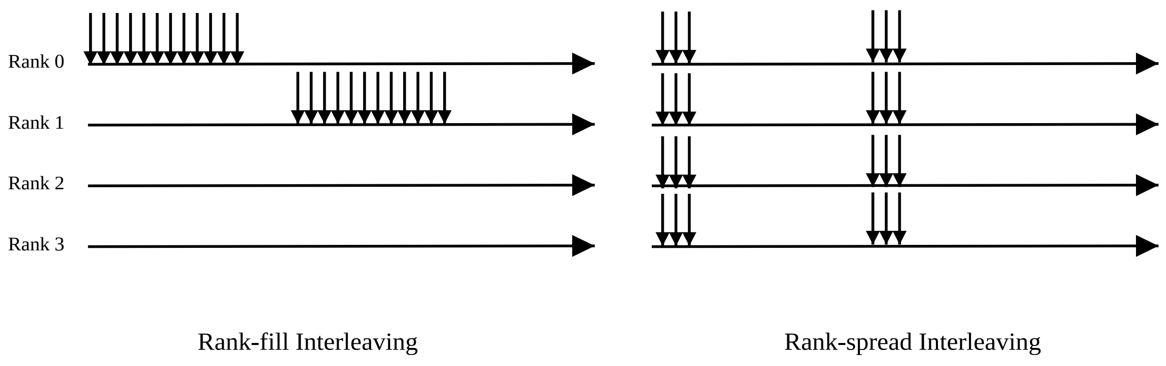


Figure 4.3: An example that shows two bursts of sequential accesses in a memory system with 4 ranks. Using rank-fill interleave, there are usually more and longer idle periods between memory accesses that can be used for saving energy, whereas in the rank-spread case, each time a burst of sequential memory requests occurs, all the ranks are touched, thus making the average idle time smaller. However, when performing power management, longer idle times are crucial to have for entering some of the deep power-saving states.

performance and energy tradeoff using different rank interleave schemes in Section 4.5.

4.3 Active vs. Passive Power Management

Even though read and write operations dissipate the most amount of power, they do not consume a significant amount of energy due to their short duration. Instead, most of the energy are consumed when memory is idling. We show in Figure 4.4 that for a SPECjbb workload, energy is mostly consumed in the Precharge state and by peripheral components, i.e., PLL and registers. This power can be significantly reduced by transitioning memory ranks and the peripheral components to a low-power state during idle periods.

Powerdown is one of the two low-power states implemented in the DDR memory architecture, and it uses 31% of the Precharge power. Having only a 5 ns resynchronization latency, using Powerdown, power can be reduced even with short idle periods; however, it is not nearly as power-efficient as the Self Refresh state where we can also put the PLL and the registers to their low-power state. Its potential benefit is clearly shown in Figure 4.4. However, due to having a much longer resynchronization latency when exiting from Self Refresh, idle periods of at least 19 μsec are needed just to break even. This is more than

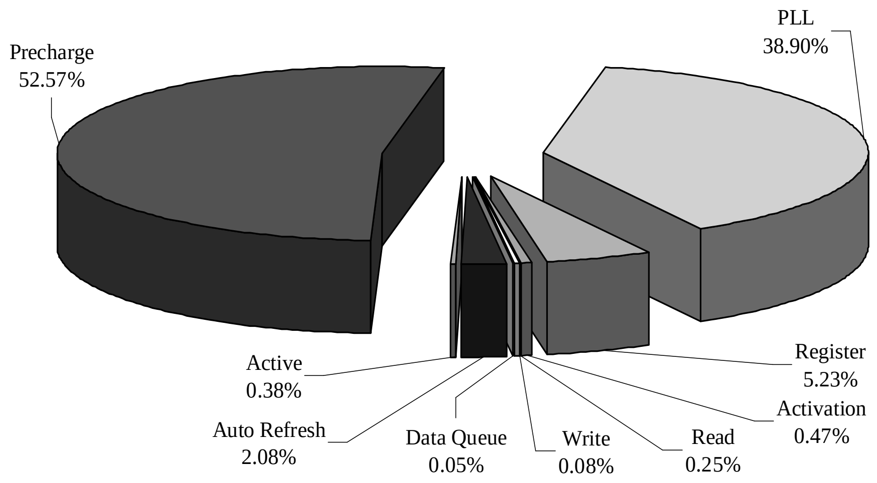


Figure 4.4: Breakdown of the energy consumed by DRAM.

three orders-of-magnitude longer than the break-even time for entering Powerdown. We calculate the break-even time using the Energy$\times$Delay metric as described in [45].

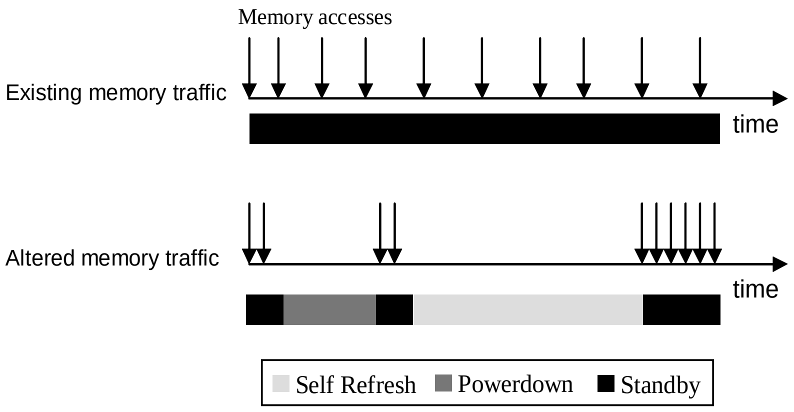


Figure 4.5: In the first case (above figure), the gaps between consecutive memory accesses are too short for entering low-power states to have any benefits. In the second case, by delaying and batching memory accesses, we can create longer idle periods, thus allowing power management to take advantage of some deeper power-saving states.

Due to the randomness in memory accesses, long idle periods are rarely observed in realistic workloads. As a result, it often inhibits the use of Self Refresh, which severely limits the amount of power that can be saved from using existing power-management techniques. This is illustrated by an example shown in Figure 4.5, where we show that simply making power-management decisions based on the monitored memory traffic is often not

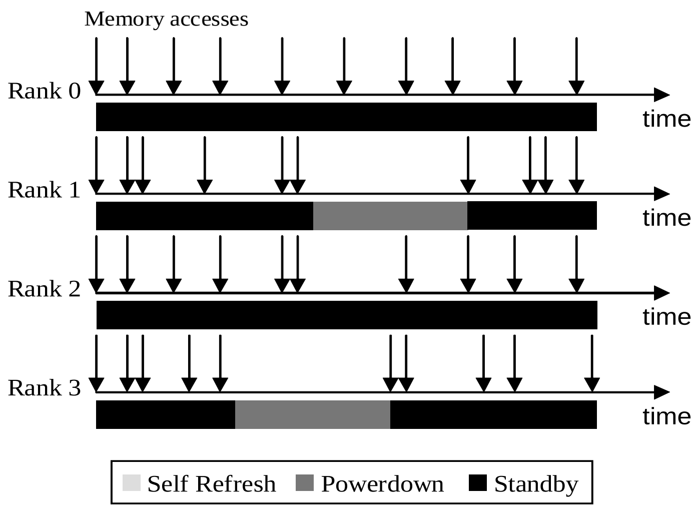


Figure 4.6: An example showing that if memory traffic is left unshaped, power management cannot take full advantage of deeper power-saving states since most idle periods are too short.

enough—the observed memory traffic might not present the necessary energy-saving opportunities. However, if we can alter the traffic pattern in a certain way, it is possible to create longer idle periods, from which power can be more effectively reduced. Unfortunately, the particular technique we show in Figure 4.5 is not very useful in practice as we cannot control memory accesses at such a fine granularity. Additionally, by delaying and batching memory accesses, we will pay a severe performance penalty. In the following section, we illustrate a more practical and low-overhead method of reshaping memory traffic to improve energy-efficiency.

4.4 Memory Traffic Reshaping

To reshape the memory traffic to our benefit, we must make memory accesses less random and more controllable. Conventional memory traffic often seems random because (1) the operating system arbitrarily maps virtual pages to physical pages, and (2) different pages are often accessed very differently at run-time. As a result of such randomness, the observed interarrival characteristic of memory requests is often not favorable for reducing power.

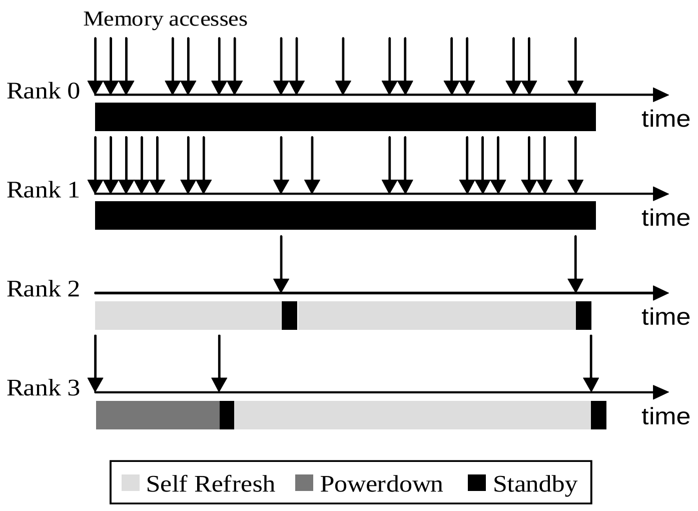


Figure 4.7: An example showing that if memory traffic can be loosely controlled (e.g., by migrating pages), some ranks will, as a result, have much longer idle periods, thus allowing the use of the deeper power-saving states.

To give an example, we use a 4-rank system shown in Figure 4.6. Due to the arbitrary OS's page mapping, memory requests will be randomly distributed across the 4 ranks. This creates a large number of small and medium-sized idle periods. The smaller idle periods are often completely useless and cannot be used for saving energy. As for the medium-sized ones, we can transition memory devices to Powerdown and obtain a moderate amount of power savings. However, to significantly reduce power, we need to take advantage of Self Refresh's ultra low-power property. Unfortunately, as it can be seen from this example, due to the lack of long idle periods, Self Refresh is very infrequently utilized.

4.4.1 Hot Ranks and Cold Ranks

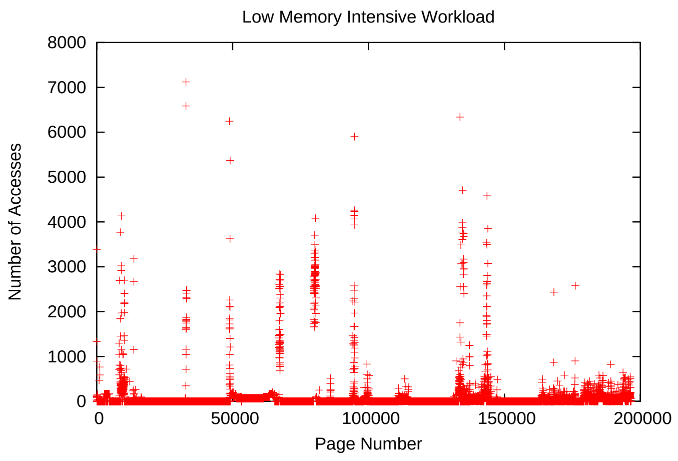
To elongate idle periods, we introduce the concepts of *hot* and *cold* ranks. Hot ranks are used to hold frequently accessed pages, which leaves infrequently used and unmapped pages on cold ranks. Hot ranks are ranks created by migrating frequently accessed memory pages onto from the cold ranks. The mechanism to migrate pages from one rank to another was previously described in full detail in [62]. The result of making this differentiation among ranks is shown in Figure 4.7. Here we assume that Rank 0 and 1 are used as hot ranks and Rank 2 and 3 are used as cold ranks. Essentially, we are increasing the utilization of hot ranks and decreasing the utilization of cold ranks. As a result, the additional memory requests imposed upon these hot ranks will “fill-in” between the existing idle gaps. As most of these gaps were small and could not be used for saving power anyways, by servicing additional requests during such time, we can make more efficient use of the power dissipated by the hot ranks. This might cause hot ranks to lose some energy-saving opportunities to use Powerdown (e.g., Rank 1 shown in Figure 4.6 and Figure 4.7), but as a result of that, more valuable opportunities are created on cold ranks where the deeper power-saving state, Self Refresh, can be more utilized. In our experiments, we found that we can save much more energy on cold ranks than the additional energy that we consume on hot ranks. We were able to observe that the average interarrival time on cold ranks were elongated by almost two orders-of-magnitude.

	75 Percentile	90 Percentile	95 Percentile	99 Percentile
LowMem Workload	5.68%	14.27%	18.60%	25.81%
HighMem Workload	0.53%	1.49%	4.98%	16.38%

Table 4.1: Shows what percentage of memory pages is responsible for 75%, 90%, 95% or 99% of all memory accesses.

4.4.2 Reducing Migration Overhead

Migrating pages causes additional memory traffic, which results in more queuing delays and contentions. Therefore, only a small number of pages can be moved without causing noticeable overheads. Fortunately, empirical observations from our experiments gave us some hints that allow us to do just that and still be able to reshape the memory traffic according to our needs. Memory traces collected from several workloads indicate that only a small percentage of pages are responsible for a majority of the memory traffic. This is shown in Figure 4.8, and we summarize the results in Table 4.1. From this table, it is clear that we can reshape the memory traffic to meet our needs by migrating only a very small percentage of pages. For example, if we want to control 90% of all memory traffic, we only need to control 1.5–14.3% of all pages. Only half of these pages would need to



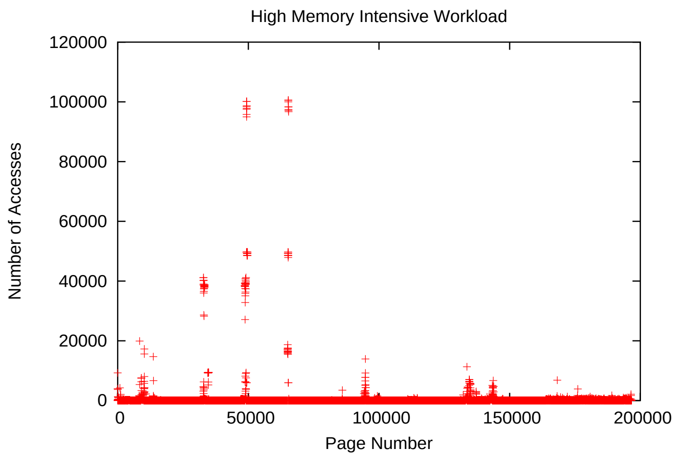


Figure 4.8: Number of times each physical page in the memory is accessed for low memory-intensive workload and high memory-intensive workload. These workloads are described in Section 4.5.

be migrated because pages are randomly allocated, and on average, 50% of the frequently accessed pages should have already been allocated on the hot ranks and do not need to move (this assumes that 50% of the memory ranks are used as hot ranks and 50% as cold ranks). Furthermore, since migration overhead is only a one-time cost, the longer a migrated page stays hot, the more we can amortize its migration cost over time. We can also think of other heuristics that we can use to further reduce the number of migrations and the migration overheads. For example, we can use process profiling to better predict the appropriate initial location where we should allocate pages for each process so that the number of page migrations can be reduced. Additionally, we can reduce migration overhead by avoiding moving heavily-shared pages, page-cache pages, and buffer-cache pages as there are more overheads in moving these types of pages. In particular, to move a shared page, we would need to change the page table entries of each of the processes sharing this page; the more processes sharing this page, the more page tables we would need to modify. Therefore, the best candidate pages to migrate are frequently accessed private pages.

4.4.3 Implementation

To determine which page to migrate, we keep a count of how many times each page was accessed. This is used by a kernel thread to find frequently access pages on cold ranks so these pages can be migrated. Currently, the page access-count table is maintained by the memory controller in our simulator. Alternatively, the same can be achieved by using software page faults to avoid hardware modification—sampling may be used to reduce page fault overheads.

4.5 Evaluation

We use the same simulation environment, analysis tools (Section 2.5.1), and workloads (Section 2.5.2) that we have used in the previous chapters. In Section 4.5.1, we first compare power and performance implications between rank-fill and rank-spread interleave schemes. Then Section 4.5.2 describes how memory traffic reshaping can significantly improve performance and energy.

Our memory simulator can simulate various power-management techniques. To compare our technique with the previously-proposed ones, we evaluate five techniques in power and performance, which are listed as follows.

- **No Power Management (NOPM):** Here, no power management is used, and ranks are transitioned to Precharge when they are idle.
- **Immediate Powerdown (IPD):** This is the simplest form of hardware power management. It is a static technique where the memory controller immediately transitions a rank to Powerdown when all memory requests on this rank have completed.

- **Immediate Self Refresh (ISR):** Same as IPD, but transitions to Self Refresh instead of to Powerdown.

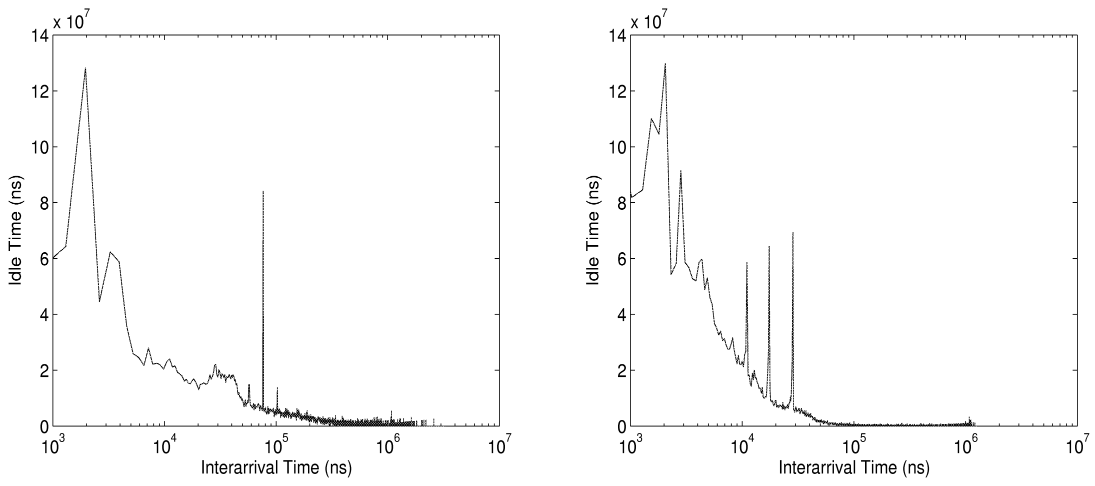
- **Dynamic Hardware Technique (HW):** This is a dynamic hardware power-management technique and is similar to the History-Based Predictor described in [41]. It monitors past memory accesses, and based on which, predicts after how long of an idle period should it transition a rank to Self Refresh (the threshold prediction algorithm is shown in Figure 2.9). Transitions to Powerdown have a zero threshold, which was previously shown to be the most efficient [25].

- **HW with Reshaped Memory Traffic ($\mathbf{HW_x}$):** Using the same HW technique as above but with a reshaped memory traffic. The x in $\mathrm{HW_x}$ represents the percentage of memory pages that we migrate when reshaping the memory traffic, i.e., $\mathrm{HW_0}$ is the same as HW.

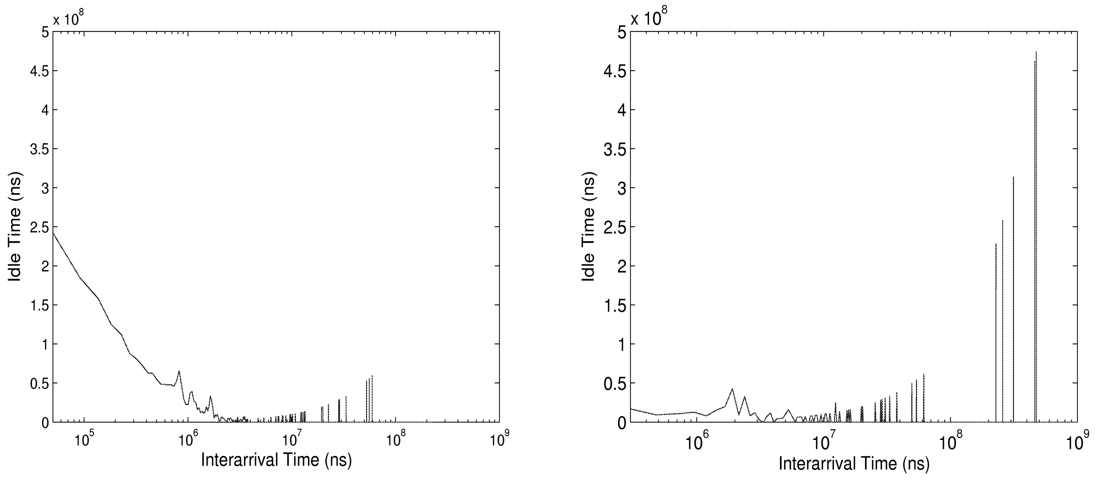
4.5.1 Rank Interleaving

When no power management is performed, results are shown in Table 4.2. As expected, power stays the same when varying the rank interleave scheme. However, as was mentioned previously, rank-spread is better than rank-fill in terms of performance. Specifically, rank-spread has a 9.0% and 14.3% faster average response time than rank-fill for the low memory-intensive and the high memory-intensive workloads, respectively.

Rank-spread's performance benefit comes from higher utilization of all the ranks. However, this leaves the interarrival time between consecutive memory requests smaller than the rank-spread interleave, thus having negative impact when power management is applied. In Table 4.3, we compare rank-spread and rank-fill under HW. It shows, despite some performance benefits, rank-spread is 80.8% and 72.5% less effective than rank-fill when power management is applied. Therefore, unless performance is critical in a system, the rank-fill interleave scheme should be used to maximize energy savings.



(a)



(b)

Figure 4.9: Part (a) shows idle time characteristic of a hot rank before (left) and after (right) migrating frequently accessed pages. Part (b) shows idle time characteristic of a cold rank before (left) and after (right) migrating frequently accessed pages. These are derived from the low memory-intensive workload. High memory-intensive workload gives similar result, thus is omitted here.

Interleaving	Low memory intensive workload			High memory intensive workload		
	Power	Norm Runtime	Avg Resp Time	Power	Norm Runtime	Avg Resp Time
Rank-fill	49.97 W	1.00	72.01 ns	52.15 W	1.00	84.13 ns
Rank-spread	49.97 W	0.99	65.55 ns	52.15 W	0.93	72.09 ns

Table 4.2: Comparing rank interleave schemes for NOPM.

Interleaving	Low memory intensive workload			High memory intensive workload		
	Power	Norm Runtime	Avg Resp Time	Power	Norm Runtime	Avg Resp Time
Rank-fill	10.24 W	1.00	90.14 ns	14.79 W	1.00	93.21 ns
Rank-spread	18.51 W	0.98	80.69 ns	24.47 W	0.91	78.66 ns

Table 4.3: Comparing rank interleave schemes when power is managed by HW.

4.5.2 Memory Traffic Reshaping

Results for the low memory-intensive and the high memory-intensive workloads are shown in Tables 4.4 and 4.5, respectively. In these tables, we show the average power dissipation, normalized runtime (with respect to no power management), and the average response time of memory accesses for each of the power-management techniques. We can see that even with the simplest static power management technique, IPD, a significant amount of power (45.73–48.89%) can be reduced without causing much impact on the performance (1.8–5.0%). Using ISR, additional power can be reduced. However, due to having a static policy to transition into Self Refresh and a much higher resynchronization latency when exiting from Self Refresh, ISR's overwhelming performance penalty (99.2–279.5%) makes it almost impractical to use in realistic workloads. On the other hand, using the dynamic hardware technique (HW), we show that if Self Refresh is utilized more carefully, a significant amount of power can be reduced (71.64–78.57%) but without significantly affecting the performance (3.5–5.6%).

Effect of Reshaping on Memory Traffic

Among these four techniques, HW is by far the most effective because it can dynamically adapt its power-management decisions as memory traffic's characteristic changes. To understand the implications of memory traffic reshaping, we re-evaluated HW with a reshaped memory traffic. However, before we delve into that, we will first look at the effects of migrating frequently accessed pages from cold ranks to hot ranks on memory traffic. In Figure 4.9, we show how the idle time characteristic (i.e., the distribution of the total

Power Management	Power	Normalized Runtime	Normalized Runtime	Average Response Time
NOPM	49.97 W	1.000	49.97	72.01 ns
IPD	25.54 W	1.018	26.00	82.17 ns
ISR	6.23 W	1.992	12.41	697.60 ns
HW	10.24 W	1.035	10.60	90.14 ns
NOPM_5	49.99 W	1.025	51.24	59.78 ns
IPD_5	25.55 W	1.051	26.85	70.47 ns
ISR_5	6.17 W	1.959	12.09	448.49 ns
HW_5	6.26 W	1.056	6.61	95.91 ns

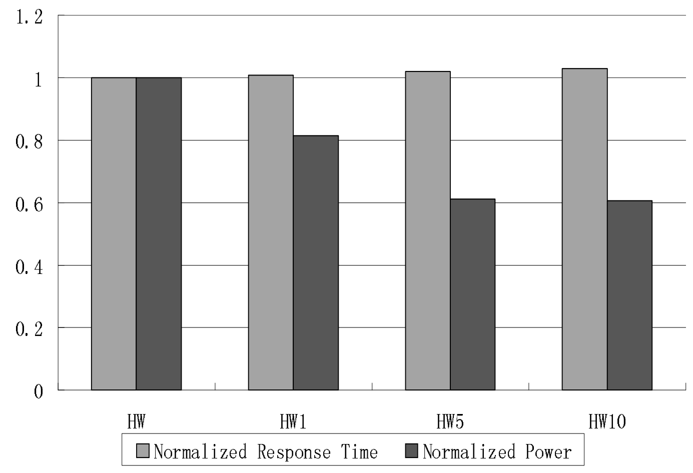
Table 4.4: Summary of the energy and performance results for the low memory-intensive workload.

Power Management	Power	Normalized Runtime	Normalized Runtime	Average Response Time
NOPM	52.15 W	1.000	52.15	84.13 ns
IPD	28.30 W	1.050	29.72	92.18 ns
ISR	14.47 W	3.795	54.92	537.45 ns
HW	14.79 W	1.056	15.62	93.21 ns
NOPM_5	52.16 W	1.105	57.63	64.85 ns
IPD_5	28.24 W	1.187	33.52	74.49 ns
ISR_5	13.16 W	3.556	46.80	384.99 ns
HW_5	9.52 W	1.190	11.33	106.25 ns

Table 4.5: Summary of the energy and performance results for the high memory-intensive workload.

idle time among different-sized idle periods) on a hot rank and a cold rank has changed after we reshaped the memory traffic. Due to having to service more memory requests, the average idle period on hot ranks has decreased from 3630 ns to 472 ns. This causes memory devices to lose opportunities to enter low-power states, but since we can benefit from entering Powerdown even with short idle periods, not much is really lost here. By redirecting a significant number of memory accesses away from cold ranks to these hot ranks, much longer idle periods are created on cold ranks. We found that after the migration of frequently accessed pages, the average idle period on cold ranks has increased from 1520 ns to 122210 ns (Figure 4.9(b)). This created more valuable opportunities where Self Refresh can be exploited.

The result of applying the HW technique to a reshaped memory traffic is shown in Tables 4.4 and 4.5, labeled as HW_5, for the low memory-intensive and the high memory-intensive workloads, respectively. Here, we migrated 5% of all memory pages to reshape the memory traffic. By doing so, we achieved 35.63–38.87% additional power savings than the original HW technique. However, due to the additional contention created on



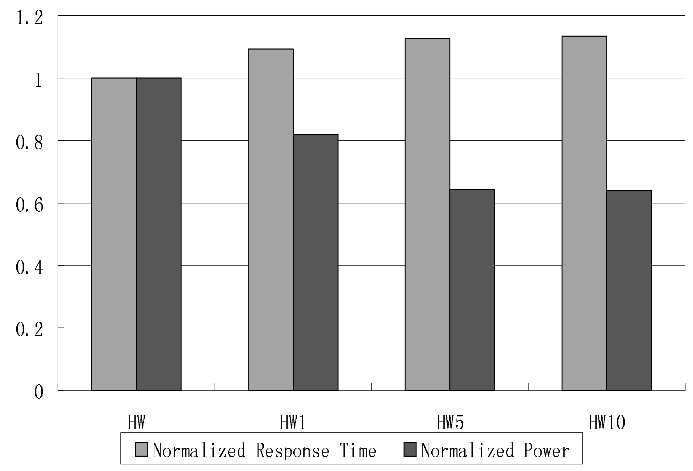


Figure 4.10: Effects of actively reshaping memory traffic by migrating 1%, 5%, and 10% of pages for the low memory-intensive workload (above) and high memory-intensive workload (below).

the hot ranks and the extra memory accesses needed to migrate pages, the performance is degraded. In the low memory-intensive workload, the performance degradation is only 2.0% compared to HW. However, in the high memory-intensive workload, performance is degraded by 12.7%. The reason for this is that pages are much more frequently accessed in the high memory-intensive workload,[1] and as a result of migrating frequently accessed pages onto the hot ranks, the additional contention created on hot ranks is much more severe in the high memory-intensive workload.

[1]Both workloads run for the same amount of time, but the high memory-intensive workload has 6 times more memory accesses than the low memory-intensive workload.

Now, we look at the effect of reshaping memory traffic on static power management techniques. Results are shown in the bottom half of Tables 4.4 and 4.5, for the low memory-intensive and the high memory-intensive workloads, respectively. Here we use the notation of $\mathrm{X_y}$ to denote that power is managed by using technique X to manage power with the memory traffic that is reshaped by migrating y% of all pages. First, we compare NOPM with $\mathrm{NOPM_5}$, and we can see that power dissipation is virtually unchanged. The very small amount of increase in power dissipation is caused by page migration, which as we can see if negligible. This makes us more confident that the overhead of page migration can be neglected, as we initially expected. With all the frequently accessed pages aggregated to a small number of hot ranks, performance, on the other hand, is more affected (2.0–10.0%) due to contention. When comparing $\mathrm{IPD_5}$ and $\mathrm{ISR_5}$ with IPD and ISR, we can see that they performed comparably. Without the ability to adapt dynamically, it is difficult for these static techniques to take advantage of the newly created longer idle periods.

To study the effect of memory traffic reshaping in more detail, we compare the results of migrating 1%, 5%, and 10% of pages. These are shown in Figure 4.10, where we normalized the average power and average runtime to that of HW with a unshaped memory traffic. Here we can see, migrating only 1% of pages gives only limited benefits in power reduction. On the other hand, migrating 10% of pages does not give any additional energy benefit beyond that of migrating 5%. In addition, it also suffers from more performance penalty due to having to migrate more pages. Therefore, migrating 5% of pages gives the best result for the workloads we ran.

Discussion

When memory is frequently accessed, as in the high memory-intensive workload, performance degradation due to contention on hot ranks can be more of a concern. However, this can be alleviated by implementing a detection mechanism that stops migration or even triggers a “reverse migration” when excessive contention is observed on hot ranks. However, this only minimally alleviates the problem. As we can see from Figure 4.10, migrating 1% as opposed to 5% of pages does not give much benefit in reducing performance penalty.

To solve the problem at its root, it calls for an alternative main memory design, where we should use high-performance, highly parallel memory for hot ranks and low-performance / power memory for cold ranks. This allows for faster access time for more frequently accessed pages and minimizes contentions on hot ranks. It also allows for more energy savings with the use of low-power memory devices to build cold ranks. Additionally, by using low-performance / power memory as cold ranks, such heterogenous main memory design can potentially lower the monetary cost of building the main memory subsystem.

4.6 Conclusions

In this chapter, we propose a technique to actively reshape memory traffic to produce longer idle periods so we can more effectively exploit idleness in the memory. Our extensive simulation in a multitasking system shows that a 35.63–38.87% additional energy can be saved by complementing existing power-management techniques with reshaped memory traffic. Our result also indicates that an alternative main memory design could be more efficient than today's homogenous design in terms of power efficiency, performance, and cost.

CHAPTER 5

FS^2: Free Space File System

5.1 Motivation

Magnetic disks have been the primary means of storing digital information on computers since 1957 when IBM first introduced RAMAC. Various aspects of magnetic disk technology, such as recording density, cost, and reliability, have been significantly improved over the years—the disk recording density has increased by more than 60% annually and the price-per-gigabyte has dropped by 35-40% annually [121]. However, due to the slowly-improving mechanical positioning components (i.e., arm assembly and rotating platters) of the disk, disks' performance is falling behind the rest of the system and has become a major bottleneck to overall system performance.

To reduce the head positioning latencies (i.e., seek time and rotational delay), we propose a novel technique that *dynamically* places copies of data in file system's *free blocks* according to the disk access patterns observed at runtime. As one or more replicas can now be accessed in addition to their original data block, choosing the "nearest" replica that provides fastest access can reduce the amount of time disk I/O operations take.

We implemented and evaluated a prototype file system based on the popular Ext2 file system. In our prototype, since the file system layout is modified only by using the free / unused disk space (hence the name *Free Space File System*, or FS^2), users are completely oblivious to how the file system layout is modified in the background; they will only notice performance improvements over time. For several workloads we ran under Linux, FS^2 is shown to have reduced disk access time by 41–68% (as a result of a 37–78% shorter seek time and a 31–68% shorter rotational delay) resulting in a 16–34% overall user-perceived performance improvement. The reduced disk access time also led to a 40–71% per-disk-access energy reduction.

5.1.1 Motivating Example

To minimize mechanical delays in magnetic disks, traditional file systems tend to place objects that are likely to be accessed together so that they are close to one another on the disk. For example, BSD UNIX Fast File System (FFS) [34] uses the concept of *cylinder group* (one or more physically-adjacent cylinders) to cluster related data blocks and their meta-data together. For some workloads, this method can significantly improve both disks' I/O latency and throughput. But for many other workloads in which the disk access pattern deviates from that was assumed by the file system, it may result in significantly degraded performance. This problem occurs because the file system is unable to take into account the disk access patterns at runtime when making disk layout decisions. This inability to use the *observed* disk access patterns to make data placement decisions will lead to poor disk utilization.

Using a simple example, we now show how traditional file systems can sometimes perform poorly even for common workloads. In this example, we monitored disk accesses during the execution of a CVS *update* command in a local CVS directory containing the Linux 2.6.7 kernel source tree. As Figure 5.1(a) shows, almost all accesses are concentrated in two narrow regions of the disk, corresponding to the locations at which the local CVS directory and the central CVS repository are placed by the file system (Ext2). By using file directory structures, Ext2 is shown to do well in clustering files that are statically related. Unfortunately, when accesses to files from two regions are interleaved, the disk head will have to move back and forth constantly between the different regions (as shown in Figure 5.1(b)), resulting in poor disk performance in spite of the efforts that were made by Ext2 for reasonable placements. Consequently, user will experience longer delays and the disk will consume more energy. In our experiment, the CVS update command took 33

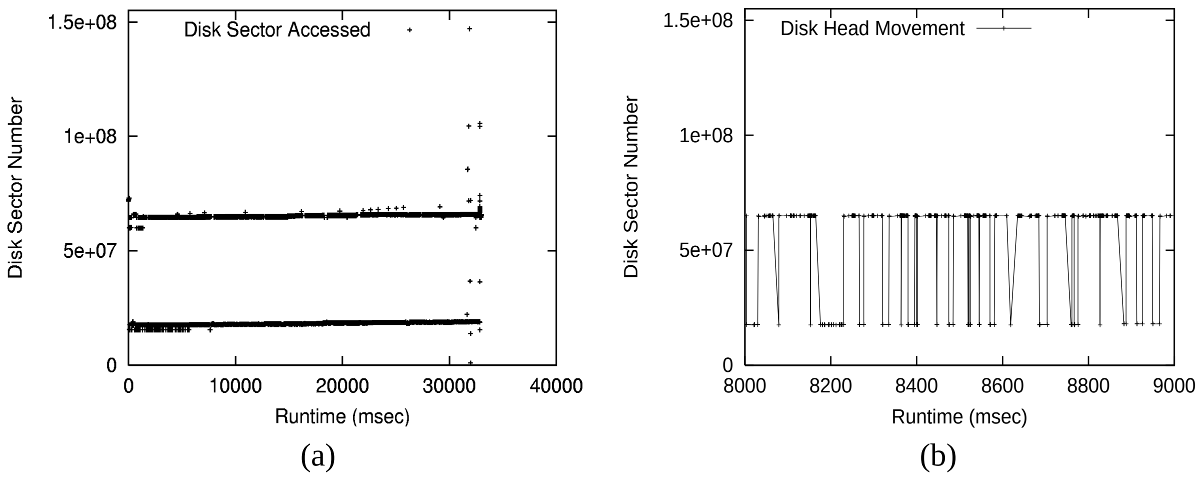


Figure 5.1: Part (a) shows disk sectors that were accessed when executing a *cvs -q update* command within a CVS local directory containing the Linux 2.6.7 source code. Part (b) shows the disk head movement within a 1-second window of the disk trace shown in part (a).

seconds to finish on an Ext2 file system using the anticipatory I/O scheduler. By managing the disk layout dynamically, the same update command took only 22 seconds to complete in our implementation of FS^2.

Traditional file systems perform well for most types of workloads and not so well for others. We must, therefore, be able to first identify the types of workloads and environments that traditional file systems generally perform poorly (listed below), and then develop good performance-improving solutions for them.

Shared systems: File servers, mail servers, and web servers are all examples of a shared system, on which multiple users are allowed to work and consume system resources independently of each other. Problems arise when multiple users are concurrently accessing the storage system (whether it is a single disk or an array of disks). As far as the file system is concerned, files of one particular user are completely independent of those of other users, and therefore, are unlikely to be stored physically close to those belong to other users. However, having concurrent disk accesses from multiple users can potentially cause the storage system to become heavily multiplexed (therefore less efficiently utilized) between several statically unrelated, and potentially distant, disk regions. A PC is also a shared system with similar problems; its disk, instead of being shared among multiple users, is shared among concurrently executing processes.

Database systems: In database systems, data and indices are often stored as large files on top of existing file systems. However, as file systems are well-known to be inefficient in storing large files when they are not sequentially accessed (e.g., transaction processing), performance can be severely limited by the underlying file system. Therefore, for many commercial database systems [50, 65], they often manage disk layout themselves to achieve higher performance. However, this increases complexity considerably in their implementation, and for this reason, many database systems [50, 65, 92, 113] still rely on file systems for data storage and retrieval.

Systems using shared libraries: Shared libraries have been used extensively by most OSs as they can significantly reduce the size of executable binaries on disk and in memory [93]. As large applications are increasingly dependent on shared libraries, placing shared libraries closer to application images on disk can result in a shorter load time. However, since multiple applications can share the same set of libraries, determining where on disk to place these shared libraries can be problematic for existing file systems as positioning a library closer to one application may increase its distance to others that make use of it.

5.1.2 Dynamic Management of Disk Layout

Many components of an operating system—those responsible for networking, memory management, and CPU scheduling—will re-tune their internal policies/algorithms as system state changes so they can most efficiently utilize available resources. Yet, the file system, which is responsible for managing a slow hardware device, is often allowed to operate inefficiently for the entire life span of the system, relying only on very simple static heuristics. We argue, like other parts of the OS, the file system should also have its own runtime component. This allows the file system to detect and compensate for any poorly-placed data blocks it has made statically. To achieve this, various schemes [2, 51, 104, 123] have been proposed in the past. Unfortunately, there are several major drawbacks in these schemes, which limited their usefulness to be mostly within the research community. Below, we summarize these drawbacks and discuss our solutions.

- In previous approaches, when disk layout is modified to improve performance, the most frequently-accessed data are always shuffled toward the middle of the disk. However, as frequently-accessed data may change over time, this would require reshuffling the layout every time the disk access pattern changes. We will show later that this re-shuffling is actually unnecessary in the presence of good reference locality and merely incurs additional migration overheads without any benefit. Instead, blocks should be rearranged only when they are accessed together but lack spatial locality, i.e., those that cause long seeks and long rotational delays between accesses. This can significantly reduce the number of replications for improving disk layout.

- When blocks are moved from their original locations they may lose useful data sequentiality. Instead, blocks should be *replicated* so as to improve both random and sequential accesses. Improving performance for one workload at the expense of another is something we would like to avoid. However, replication consumes additional disk capacity and introduces complexities in maintaining consistency between original disk blocks and their replicas. These issues will be addressed in the next section.

- Using only a narrow region in the middle of the disk to rearrange disk layout is not always practical if the rearrangement is to be done online, because interference with foreground tasks has to be considered. Interference can be significant when foreground tasks are accessing disk regions that are far away from the middle region. We will demonstrate ways to minimize this interference by using the available free disk space near current disk head position.

- Some previous techniques reserve a fixed amount of storage capacity just for rearranging disk layout. This is intrusive, as users can no longer utilize the full capacity of their disk. By contrast, using only free disk space is completely transparent to users, and instead of being wasted, free disk space can now be utilized to our benefit. To hide from users the fact that some of their disk capacity is being used to

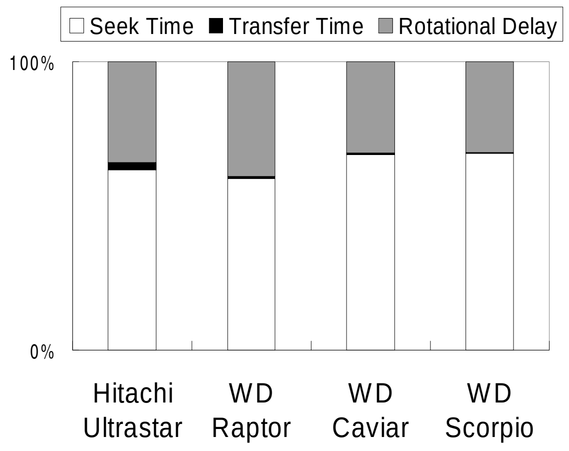


Figure 5.2: Breakdown of the disk access time in 4 drives, ranging from a top-of-the-line 15000-RPM SCSI drive to a slow 5400-RPM IDE laptop drive.

hold replicas, we need to invalidate and free replicas quickly when free disk capacity becomes limited. As the amount of free disk space can vary, this technique provides a variable quality of service, where the degree of performance improvement we can achieve depends on the amount of free disk space that is available. However, empirical evidence shows that plenty of free disk space can be found in today's computer systems, so this is rarely a problem.

- Most of previous work is either purely theoretical or based on trace analysis. We implemented a real system and evaluated it using both server-type workloads and common day-to-day single-user workloads, demonstrating significant benefits of using FS^2.

Reducing disk head positioning latencies can make a significant impact on disk performance. This is illustrated in Figure 5.2, where the disk access time of 4 different disk drives is broken down to components: transfer time, seek time, and rotational delay. This figure shows that transfer time—the only time during which disk is doing useful work—is dwarfed by seek time and rotational delay. In a random disk workload (as shown in this figure), seek time is by far the most dominating component. However, rotational delay can become the dominating component when disk access locality reaches 70% (Lumb *et al.* [83] studied this in more detail). In addition to improving performance, reduction in seek time (distance) also saves energy as disk dissipates a substantial amount of power when seeking (i.e., accelerating and decelerating disk heads at 30–40g).

The rest of this chapter is organized as follows. In Section 5.2, we first give some background in magnetic storage technology and discuss its inherent implications on performance, reliability, and energy-efficiency. Section 5.3 gives an overview of our system design. Section 5.4 discusses implementation details of our prototype based on the Ext2 file system. Section 5.5 presents and analyzes experimental results of FS^2 and compares it with Ext2. Future research directions are discussed in Section 5.6, and the chapter concludes with Section 5.7.

5.2 Background

In this section, we dissect a disk drive to understand how its internal components work together to accomplish basic I/O operations. Disk drives contain both mechanical and electronic parts. The mechanical portion is consisted of recording components (i.e., disk platters and read / write heads) and positioning components (i.e., an arm assembly that moves the heads into the correct position and a track-following system that keeps it in place). The controller portion contains a microprocessor, cache memory, and an I/O bus interface. The disk controller manages the storage and retrieval of data to and from the disk and performs mappings between the logical address requested by the CPU or DMA controller and the physical disk location. We now detail the mechanical components and the controller components, each in turn, in the following sections.

5.2.1 Mechanical Components

Modern disks range in size from 1 to 8 inches in diameter: 2.5, 3.5, and 5.25 inches are the most common sizes. Smaller disks have less surface area and thus cannot store as much data as their larger counterparts; however, they consume less power, can spin up and down faster, and have smaller distances to seek. These disks are particularly popular in mobile devices, e.g., laptops and PDAs. As for their larger counterparts, these disks offer additional capacity, more resistance to shocks, and more I/O bandwidth, which are essential to large server systems.

Storage density is one of the most rapidly advancing areas in magnetic storage technology, and its rapid growth is allowing us to store an ever increasing amount of digital data being generated. This rapid increase in storage density comes as a result of improvements in two areas. The first is linear recording density, which is determined by the maximum rate of flux change that can be recorded and read back, and the second is track-to-track distance. Being able to squeeze more bits per track and more tracks per inch significantly increased disks' areal density. However, before the 90's, disks' areal density was only improving at a rate of 25% annually. At that time, semiconductor (DRAM) was advancing at a faster pace of 40% annually, and due to which, was about to replace magnetic disks as a cheaper and higher density alternative. It never happened. In 1991, the areal density of magnetic disks had a sudden jump. Since then, it has sustained an amazing annual growth rate of 60%. This dramatic increase in areal density has allowed large-capacity drives to be manufactured with fewer platters and actuators, and consequently, reduced the price-per-gigabyte by 35–40% annually.

In spite of the rapid increase in disk density, large-capacity disks built from a large number of platters are sometimes needed. More platters offers higher capacity, but they also require more time to spin up and down, and more power to keep them spinning. Each platter may have either one or two recording surfaces, with each having a head responsible for writing and reading magnetic flux variations to store and retrieve data. However, there can only be one active head at any instant in time since there is only a single read-write data channel that is shared among all the heads. This channel is used for encoding and decoding data streams into or from a series of magnetic phase changes. This limits disk performance somewhat, but its impact on disks' performance is fairly minor when compared to other factors.

Disks have always been one of the slowest components in computer systems. Over the years, its access time has improved by only 7–10% per year [105]. Compared to the rate of improvement in solid-state devices (e.g., microprocessor, cache, and DRAM), which has been following Moore's Law closely in the past two decades, disk is actually becoming relatively slower over time. This causes an alarming rate of disparity between the speed of accessing elements (e.g., microprocessor) and the accessed elements (e.g., disk). This widening disparity between components' speeds is often the source of performance inefficiency in many systems.

Disks are slow and hard to speed up because it is very difficult to speed up its mechanical positioning delays—seek time and rotational delay. A seek operation is composed of the following phases.

- *Speedup:* When the disk arm is accelerating and has not reached half of the seek distance or its maximum velocity.
- *Coast:* When the disk arm is moving at a constant speed of its maximum velocity.
- *Slowdown:* When the disk arm is decelerating until it rests close to the desired track.
- *Settle:* When the disk arm slightly adjusted itself among few neighboring track and finally brought itself to rest close to the desired track.

Very short seeks (less than 2 to 4 cylinders) are dominated by the settle time. In fact, a seek may not even occur; the head may just *resettle* itself into position on the destination track. Short seeks (less than 200 to 400 cylinders) spend almost all their time in the acceleration and deceleration phase, and the time is proportional to the square root of the seek distance plus the settle time. Long seek operations spend most of their time moving at a constant speed, taking an amount of time that is linearly proportional to the seek distance plus a constant overhead.

Once the disk arm is positioned over the track that contains the data of interest (i.e., at the completion of a seek), the disk waits for a *rotation latency,* which can vary from zero to a full disk rotation time, before the head is positioned directly above the first sector of the data to be accessed. With disk platters rotating in lockstep on a central spindle, 7200 RPM is the most common rotational speed among modern disks, however, 10000 and 15000 RPM disks are also becoming more common in high-end systems. The median rotation speed is increasing at a compound rate of about 12 percent per year. This allows a higher transfer rate and shortens the rotation latency. Unfortunately, power dissipation is also increased.

We will see shortly, it is due to these mechanical and electro-magnetic components that disk is slow compared to other components in the system. More specifically, this slowness is mainly due to seeking and waiting for rotation delay. We will discuss ways to reduce them shortly.

5.2.2 Controller Components

The disk controller mediates access to the mechanical components, runs the track-following system, transfers data between the disk and its clients, and, in many cases, manages an embedded cache. Controllers are built around specially designed embedded processors, which often have digital signal processing capability and special interfaces that let them control hardware directly. The trend is toward more powerful controllers for handling increasingly sophisticated interfaces and for reducing costs by replacing previously dedicated electronic components with firmware.

5.2.3 Performance, Fault-Tolerance, and Power

In the following subsections, we will briefly discuss the three majors areas of disk drives—performance, fault-tolerance, and energy-efficiency. We delve into techniques for improving these areas in Section 5.3.

Performance

Magnetic disk is much slower to access than solid-state devices. DRAM takes tens of nanoseconds to access, whereas magnetic disk would take tens of milliseconds to access—almost six orders-of-magnitude slower. The primary reason it is slow is that disk access time can be overwhelmingly dominated by overheads. Disk access time is consisted of (i) seek time T_s, (ii) rotation delay T_r, and (iii) transfer time T_t, but it is only during the transfer time, is the disk doing any useful work; both T_s and T_r are overheads. As demonstrated in [83], for a random workload, T_t is usually minuscule compared to T_s or T_r, thus often resulting in poor disk utilization and slow disk access time. Therefore, to improve disk performance, we need to either reduce seek time or rotation delay, or both.

Fault-Tolerance

Disk's performance plays a critical role in the overall system's performance, and therefore, is important to optimize. However, protecting data against corruption and loss is equally important, if not more. Ideally, we would like our storage components to have 100% availability and never fail. However, in reality, hard disk failures are fairly common, which is mainly due to its mechanical parts. The two most common types of failures in hard disk are electronic failures and mechanical failures. An electronic failure occurs when the controller board on the disk dies. This is most commonly caused by a transient electrical surge. Luckily, in most cases, data are recoverable in the aftermath as disk platters are usually left unscathed. To recover, it is often sufficient to replace the controller board with an identical one from another drive. Mechanical failures, on the other hand, have more serious consequences. They can often render all data on disk unrecoverable. Some common causes of this type of failures include mechanical shocks, contamination, condensation, servo errors, and head crashes (cause approximately 45% of the cases where data is lost [56]). Mechanical failures occur fairly frequently, and without means to guard against them, all data on disk can disappear in a few seconds and beyond recovery.

Various fault-tolerance techniques have been proposed previously to protect against data loss. Redundant Array of Inexpensive Disks, or RAID [94], is widely used by large data centers, Internet service providers, corporations, and even individuals. It provides a flexible way to tradeoff between performance, fault-tolerance, and cost. However, we feel that additional fault-tolerance can be gained without sacrificing the other dimensions.

Energy Efficiency

In both small mobile devices and large data centers, magnetic disks can dissipate a significant percentage of the total power. This is also true for data centers, where a large number of high-performance and high-power disk drives are always kept in the full power mode. According to [4, 89], cost of electricity for powering computing and cooling equipments can be as much as 30% of the total cost of ownership (TCO) in data centers. Among various components of a data center, disk is one of the largest consumers of energy. A recent industry report [12] shows that storage devices account for almost 27% of the total energy consumed in a data center. This problem is exacerbated by the availability of faster disks that consume more power, as well as the recent shift from tape backup systems to disk backup to speed up storage and retrieval operations.

Power management techniques that exploit idle periods have been previously proposed for magnetic disks. However, it was recently shown that for server-type workload, most idle periods are too small to be exploited for saving energy [57, 132].

5.3 Design

This section details the design of FS^2 on a single-disk system. Most of the techniques used in building FS^2 on a single-disk can be directly applied to an array of disks (e.g., RAID), yielding a performance improvement similar to that can be achieved on a single disk (in addition to the performance improvements that can be achieved by RAID). More on this will be discussed in Section 5.6.

FS^2 differs from other dynamic disk layout techniques due mainly to its exploitation of free disk space, which has a profound impact on our design and implementation decisions. In the next section, we first characterize the availability of free disk space in a large university computing environment and then describe how free disk space can be used for our purpose. Details on how to choose candidate blocks to replicate and how to use free disk space to facilitate the replication of these blocks are given in the following section. Then, we describe the mechanisms we used to keep track of replicas both in memory and on disk so that we can quickly find all alternative locations where data is stored, and then choose the "nearest" one that can be accessed fastest.

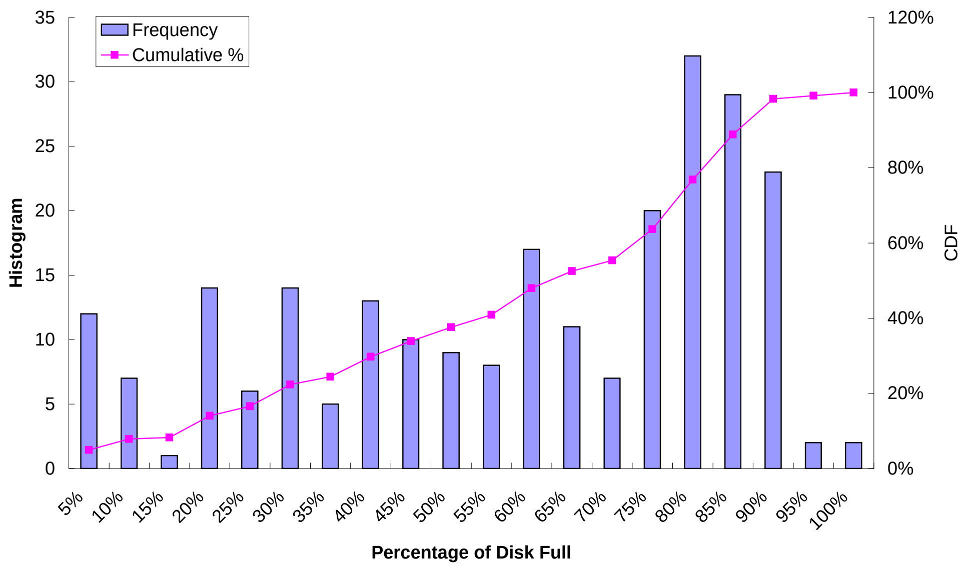


Figure 5.3: This figure shows the amount of free disk space available on 242 disks from 22 public server machines.

5.3.1 Availability of Free Disk Space

Due to the rapid increase in disk recording density, much larger disks can be built with a fewer number of platters, reducing their manufacturing cost significantly. With bigger and cheaper disks being widely available, we are much less constrained by disk capacity, and hence, are much more likely to leave a significant portion of our disk unused. This is shown from our study of the 242 disks installed on the 22 public servers maintained by the University of Michigan's EECS Department in April 2004. The amount of unused capacity that we have found on these disks is shown in Figure 5.3. As we have expected, most of the disks had a substantial amount of unused space—60% of the disks had more than 30% unused capacity, and almost all the disks had at least 10% unused capacity. Furthermore, as file systems may reserve additional disk space for emergency / administrative use, which we did not account for in this study, our observation gives only a lower bound of how much free disk space there really was.

Our observation is consistent with the study done by Douceur *et al.*[19] who reported an average use of only 53% of disk space among 4801 Windows personal computers in a commercial environment. This is a significant waste of resource because, when disk space is not being used, it usually contributes nothing to users / applications. So, we argue that free disk space can be used to dynamically change disk layout so as to increase disk I/O performance.

Free disk space is commonly maintained by file systems as a large pool of contiguous blocks to facilitate block allocation requests. This makes the use of free disk space attractive for several reasons. First, the availability of these large contiguous regions allows discontiguous disk blocks which are frequently accessed together, to be replicated and aggregated efficiently using large sequential writes. By preserving replicas' adjacency on disk according to the order in which their originals were accessed at runtime, future accesses to these blocks will be made significantly faster. Second, contiguous regions of free disk blocks can usually be found throughout the entire disk, allowing us to place replicas to the location closest to current disk head's position. This minimizes head positioning latencies when replicating data, and only minimally affects the performance of foreground tasks. Third, using free disk space to hold replicas allows them to be quickly invalidated and their used space to be converted back to free space whenever disk capacity becomes limited. Therefore, users can be completely oblivious to how the "free" disk capacity is used, and will only notice performance improvements when more free disk space becomes available.

The use of free disk space to hold replicas is not without costs. First, as multiple copies of data may be kept, we will have to keep data and their replicas consistent. Synchronization can severely degrade performance if for each write operation, we have to perform an additional write (possibly multiple writes) to keep its replicas consistent. Second, when the user demands more disk space, replicas should be quickly invalidated and their occupied disk space returned to the file system so as to minimize user-perceived delays. However, as this would undo some of the work done previously, it should be avoided as much as possible. Moreover, as the amount of free disk space can vary over time, deciding how to make use of the available free disk space is also important, especially when there is only a small amount of it left. These issues will be addressed in the following subsections.

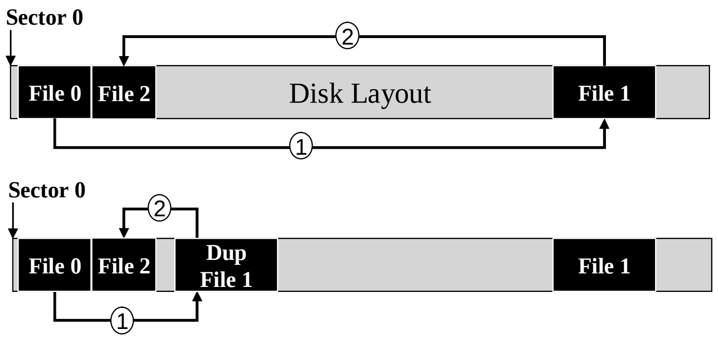


Figure 5.4: Files are accessed in the order of 0, 1, and 2. Due to the long inter-file distance between File 1 and the other files, long seeks would result if accesses to these files are interleaved. In the bottom figure, we show how the seek distance (time) could be reduced when File 1 is replicated using the free disk space near the other two files.

5.3.2 Block Replication

We now discuss how to choose which blocks to replicate so as to maximize the improvement of disk I/O performance. We start with an example shown in Figure 5.4, describing a simple scenario where three files—0, 1, and 2—are accessed, in this order. One can imagine that Files 0 and 2 are binary executables of an application, and File 1 is a shared library, and therefore, may be placed far away from the other two files, as shown in Figure 5.4. One can easily observe that File 1 is placed poorly if it is to be frequently accessed together with the other two files. If the files are accessed sequentially, where each file is accessed completely before the next, the overhead of seeking is minimal as it would only result in two long seeks—one from 0 to 1 and another from 1 to 2. However, if the accesses to the files are interleaved, performance will degrade severely as many more long-distant seeks would result. The use of I/O prefetching can alleviate some unnecessary head movements in case of sequential accesses, but its benefit is limited when the files are accessed simultaneously with random I/Os. To improve performance, one can try to reduce the seek distance between the files by replicating File 1 closer to the other two as shown in the bottom part of Figure 5.4. Replicating File 1 also reduces rotational delay when its meta-data and data are placed onto the disk consecutively according to the order in which they were accessed. This is illustrated in Figure 5.5. Reducing rotational delay in some workloads (i.e., those already with a high degree of disk access locality) can be more effective than reducing seek time in improving disk performance. The algorithm we used in our implementation to select candidate blocks to replicate and decide where to place the replicas is described in 5.4.1.

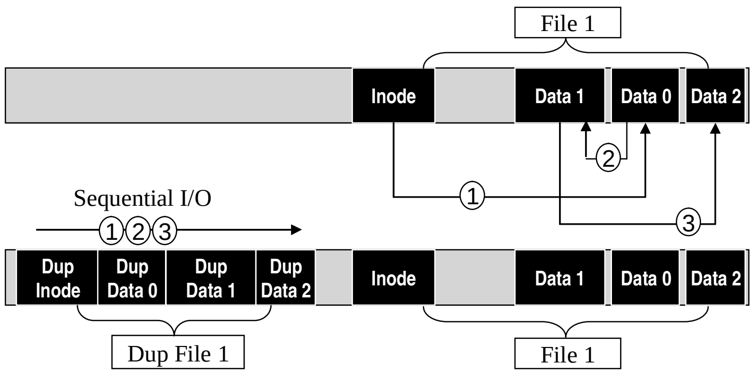


Figure 5.5: This figure shows a more detailed disk layout of File 1 shown in Figure 5.4. As file systems tend to place data and their meta-data (and also related data blocks) close to one another, intra-file seek distance (time) is usually short. However, as these disk blocks do not necessarily have to be consecutive, rotational delay will become a more important factor. In the lower portion of the figure, we show how rotational delay can be reduced by replicating File 1 and placing its meta-data and data in the order that they were accessed at runtime.

In real systems, the disk access pattern is more complicated than that shown in the above examples. The examples are mainly to provide an intuition for how seek time and rotational delay can be reduced via replication. We present more practical heuristics used in our implementation in Section 5.4.

5.3.3 Keeping Track of Replicas

To benefit from having replicas, we must be able to find them quickly and decide whether it is more beneficial to access one of the replicas or the original data. This is accomplished by keeping a hash table in memory as shown in Figure 5.6. It occupies only a small amount of memory: assuming 4 KB file system data blocks, a 4 MB hash table suffices to keep track of 1 GB of replicas on disk. As most of today's computer systems have hundreds of megabytes of memory or more, it is usually not a problem to keep all the hash entries in the memory. For each hash entry, we maintain 4 pieces of information: the location of the original block, the location of its replica, the time when the replica was last accessed, and the number of times that the replica was accessed. When free disk space drops below a low watermark (at 10%), the last two items are used to find and invalidate replicas that have not been frequently accessed and were not recently used. At this point, we suspend the replication process. It is resumed only when a high watermark (at 15%) is reached. We also define a critical watermark (at 5%), which triggers a large number of replicas to be invalidated in a batched fashion so a contiguous range of disk blocks can be quickly released to users. This is done by grouping hash entries of replicas that are close to one another on disk to be linked to one another in memory, as shown in Figure 5.6. When the number of free disk blocks drops below the critical watermark, we simply invalidate all the replicas of an entire disk region, one region at a time, until the number of free disk blocks is above the low watermark or when there are no more replicas. During the process, we could have potentially invalidated more valuable replicas and kept less valuable ones. This is acceptable since our primary concern under such a condition is to provide users with the needed disk capacity as quickly as possible.

Given that we can find replicas quickly using the hash table, deciding whether to use an original disk block or one of its replicas is as simple as finding the one closest to the current disk head location. The location of the disk head can be predicted by keeping track of the last disk block that was accessed in the block device driver.

As a result of replication, there may be multiple copies of a data block placed at different locations on disk. Therefore, if a data block is modified, we need to ensure that all of its replicas remain consistent. This can be accomplished by either updating or invalidating the replicas. In the former (updating) option, modifying multiple disk blocks when a single data block is modified can be expensive. Not only it generates more disk traffic, but it also incurs synchronization overheads. Therefore, we chose the latter option because both data invalidation and replication operations can be done cheaply and in the background.

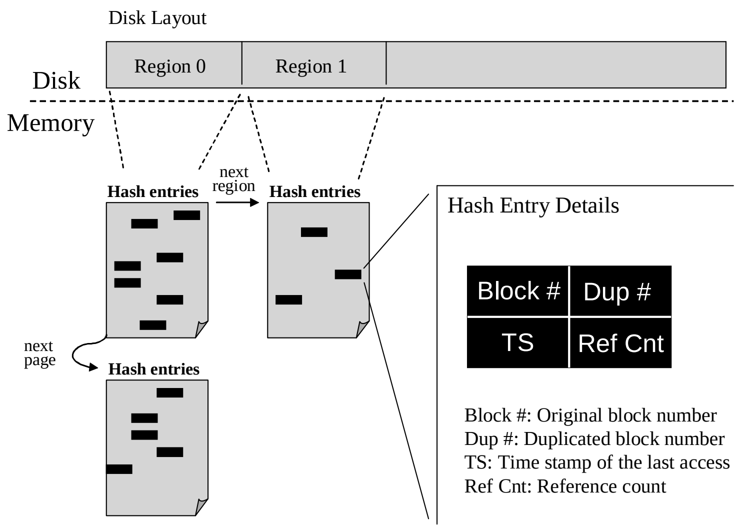


Figure 5.6: A detailed view of the hash data structure used to keep track of replicas. Hashing is used to speed up the lookup of replicas. Replicas that are close to one another on disk also have their hash entries close to one another in memory, so a contiguous range of replicas can be quickly invalidated and released to users when needed.

5.4 Prototype

Our implementation of FS^2 is based on the Ext2 file system [11]. In our implementation, the functionality of Ext2 is minimally altered. It is modified mainly to keep track of replicas and to maintain data consistency. The low-level tasks of monitoring disk accesses, scheduling I/Os between original and replicated blocks, and replicating data blocks are delegated to the lower-level block device driver. Implementation details of that in the block device driver and the file system are discussed in Section 5.4.1 and Section 5.4.2, respectively. Section 5.4.3 describes some of the user-level tools we developed to help users manage the FS^2 file system.

5.4.1 Block Device Implementation

Before disk requests are sent to the block device driver to be serviced, they are first placed onto an I/O scheduler queue, where they are queued, merged, and rearranged so the disk may be better utilized. Currently, the Linux 2.6 kernel supports multiple I/O schedulers that can be selected at boot time. It is still highly debatable which I/O scheduler performs best, but for workloads that are highly parallel in disk I/Os, the anticipatory scheduler [67] is clearly the winner. It is used in both the base case Ext2 file system and our FS^2 file system.

In our implementation, the anticipatory I/O scheduler is slightly modified so a replica can be accessed in place of its original if the replica can be accessed faster. A disk request is represented by a starting sector number, the size of the request, and the type of the request (read or write). For each read request (write requests are treated differently) that the I/O scheduler passes to the block device driver, we examine each of the disk blocks within the request. There are several different cases we must consider. In the simplest case, where none of the blocks have replicas, we have no choice but to access the original blocks. In case where all disk blocks within a request have replicas, we access the replicas instead of their originals when (i) the replicas are physically contiguous on disk, and (ii) they are closer to the current disk head's location than their originals. If both criteria are met, to access the replicas instead of their originals, we would need to modify only the starting block number of that disk request. If only a subset of the blocks has replicas, they should not be used as it would break up a single disk request into multiple ones, and in most cases, would take more time. For write accesses, replicas of modified blocks are simply invalidated, as it was explained in Section 5.3.3.

Aside from scheduling I/Os between original and replicated blocks, the block device driver is also responsible for monitoring disk accesses so it can decide which blocks we should replicate and where the replicas should be placed on disk. We first show how to decide which blocks should be replicated. As mentioned earlier, good candidates are not necessarily frequently-accessed disk blocks, but rather, temporally-related blocks that lack good spatial locality. However, it is unnecessary to replicate all the candidate blocks. This was illustrated previously by the example shown in Figure 5.4. Instead of replicating all three files, it is sufficient to replicate only File 1—we would like to do a minimal amount of work to reduce mechanical movements.

To do so, we partition disks into multiple regions.[1] Among them, we find a single *hot*

[1]For convenience, we define a region to be a multiple of Ext2 block groups.

region, which is an area on the disk where the disk head has been most active. As workload changes, the hot region may also change from time to time. Its location is estimated in our implementation by keeping track of the number of disk I/Os serviced from each region for each time epoch. This is only an approximation because some disk accesses are faster than others—(i) some disk blocks can be directly accessed using the on-disk cache, thus not involving any mechanical movements, and (ii) some read accesses can trigger 2 seek operations if it causes a cache entry conflict and the cache entry is dirty. However, in the disk traces we have collected (described in Section 5.5), this heuristic is found to be sufficient for our purpose.

Ideally, we would like to keep the disk head from moving out of the hot region to minimize positioning latencies. Inefficiency happens when the disk head has to move outside of the target region to service a disk request. To prevent or minimize the chance of this happening, we identify all the disk blocks accessed outside of the target region as candidate blocks for replication. By placing their replicas to the available free space within the target region, similar disk access patterns can be serviced much faster in future; we achieve shorter seek times due to shorter seek distances and smaller rotational delays because replicas are laid out on disk in the same order as they were previously accessed.

Although the block device driver can easily find a list of candidate blocks to replicate and the hot region to place these replicas, it cannot perform replication by itself because it cannot determine how much free disk space is there, nor can it decide which blocks are free and which are in use by the file system. Implicit detection of data liveness from the block device level was shown by Sivathanu *et al.* [35] to be impossible in the Ext2 file system. Therefore, replication requests, include which blocks to replicate and which disk region to replicate them to, are forwarded from the block device driver to the file system, where free disk space can be easily located. Contiguous free disk space is used if possible so that replicas can be written sequentially to disk. When choosing a region to place replicas, not only is the free space in the target region searched, several neighboring disk regions (within 5% of the total disk capacity) are also searched if the target region is completely full. In case even the neighboring regions are completely full, we simply discard replication requests to this region. To avoid completely using a single disk region, we place more weight on regions with more free disk space for placing replicas. This naive approach seems to work well from our observation, but one can imagine implementing a replica-replacement policy to discard rarely used replicas to make room for new (and potentially more valuable) replicas. Lastly, we rely on the default I/O scheduler for committing replicas to disk. As replicas are often placed to contiguous disk regions, the overhead of replication can be amortized.

5.4.2 File System Implementation

Ext2 is a direct descendant of BSD UNIX FFS [34], and it is commonly chosen as the default file system by many Linux distributions. Ext2 splits a disk into one or more block groups, each of which contains its own set of inodes (meta-data) and data blocks to enhance data locality and fault-tolerance. More recently, journaling capability was added to Ext2, to produce the Ext3 file system [122], which is backward-compatible to Ext2. We could have based our implementation on other file systems, but we chose Ext2 because it is the most popular file system used in today's Linux systems. As with Ext3, FS^2 is backward-compatible with Ext2.

First, we modified Ext2 so it can assist the block device driver to find free disk space near the current disk head's location for placing replicas. As free disk space of each Ext2 block group is maintained by a bitmap, large regions of free disk space can be easily found. We also modified Ext2 to assist the block device driver in maintaining data consistency. In Section 5.4.1, we discussed how data consistency is maintained for write accesses in the block device level. However, not all data consistency issues can be observed and addressed in the block device level. For example, when a disk block with one or more replicas is deleted or truncated from a file, all of its replicas should also be invalidated and released. As the block device driver is completely unaware of block deallocations, we modified the file system to explicitly inform the block device driver of such events so it can invalidate these replicas and remove them from the hash table.

To allow replicas to persist across restarts, we flush the in-memory hash table to the disk when the system shuts down. When the system starts up again, the hash table is read back into the memory. We keep the hash table as a regular Ext2 file using a reserved inode #9 so it cannot be accessed by regular users. To minimize the possibility of data inconsistency that may result from a system crash, we flush modified hash entries to disk periodically. We developed a set of user-level tools (discussed in Section 5.4.3) to restore the FS^2 file system back to a consistent state when data consistency is compromised after a crash. An alternative solution is to keep the hash table in flash memory. Flash memory provides a persistent storage medium with low-latency so the hash table can be kept consistent even when the system unexpectedly crashes. Given that the size of today's flash memory devices usually ranges between a few hundred megabytes to tens of gigabytes, a hash table of a few megabytes can be easily stored.

Ext2 is also modified to monitor the amount of free disk space to prevent replication from interfering with user's normal file system operations. We defined a high (15%), a low (10%), and a critical (5%) watermark to monitor the amount unused disk space and to respond with appropriate actions when a watermark is reached. For instance, when the amount of free disk space drops below the low watermark, we will start reclaiming disk space by invalidating replicas that have been used neither recently nor frequently. If the amount of free disk space drops below the critical watermark, we start batching invalidations for entire disk regions as described in Section 5.3.3. Moreover, replication is suspended when the amount of free disk space drops below the low watermark, and it is resumed only when the high watermark is reached again.

5.4.3 User-level Tools

We created a set of user-level tools to help system administrators manage the file system. *mkfs2* is used to convert an existing Ext2 file system to an FS^2 file system, and vice versa. Normally, this operation takes less than 10 seconds. Once an FS^2 file system is created and mounted, it will automatically start monitoring disk traffic and replicating data on its own. However, users with higher-level knowledge about workload characteristics can also help make replication decisions by asking *mkfs2* to statically replicate a set of disk blocks to a specified location on disk. It gives users a knob to manually tune how free disk space should be used.

As mentioned earlier, a file system may crash, resulting in data inconsistency. During system startup, the tool chkfs2 restores the file system to a consistent state if it detects that the file system was not cleanly unmounted, similarly to what Ext2's e2fsck tool does. The time to restore an FS^2 file system back to a consistent state is only slightly longer than that of Ext2, and the time it takes depends on the number of replicas in the file system.

5.5 Experimental Evaluation

We now evaluate FS^2 under some realistic workloads. By dynamically modifying disk layout according to the disk access pattern observed at runtime, seek time and rotational delay are shown to be reduced significantly, which translates to a substantial amount of performance improvement and energy savings. Section 5.5.1 describes our experimental setup and Section 5.5.2 details our evaluation methodology. Experimental results of each workload are discussed in each of the subsequent subsections.

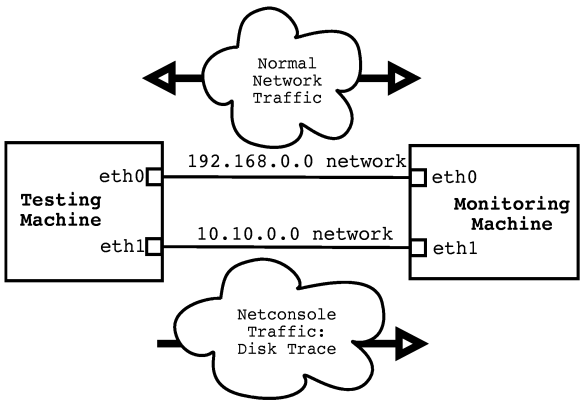


Figure 5.7: The testbed consists a testing machine for running workloads and a monitoring machine for collecting disk traces. Each machine is equipped with 2 Ethernet cards—one for running workloads and one for collecting disk traces using *netconsole*.

5.5.1 Experimental Setup

Our testbed is set up as shown in Figure 5.7. The IDE device driver on the testing machine is instrumented, so all of its disk activities are sent via *netconsole* to the monitoring machine, where the activities are recorded. We use separate network cards for running workloads and for collecting disk traces to minimize interference. System configuration of the testing machine and the specification of its disk are provided in Table 5.1.

Components	Specification
CPU	Athlon 1.343 GHz
Memory	512MB DDR2700
Network cards	2x3COM 100-Mbps
Disk	Western Digital
Bus interface	IDE
Capacity	78.15 GB
Rotational speed	7200 RPM
Average seek time	9.9 ms
Track-to-track seek time	2.0 ms
Full-stroke seek time	21 ms
Average rotational delay	4.16 ms
Average startup power	17.0 W
Average read/write/idle power	8.0 W
Average seek power	14.0 W

Table 5.1: System specification of the testing machine.

Several stress tests were performed to ensure that the system performance is not significantly affected by using netconsole to collect disk traces. We found that the performance was degraded by 5% in the worst case. For typical workloads, performance should only be minimally affected. For example, the time to compile a Linux 2.6.7 kernel was not affected at all when its disk activities were recorded via netconsole. As this overhead is imposed on both the base case (Ext2) and FS^2, we believe our tracing mechanism had only a negligible effect on our results.

5.5.2 Evaluation Methodology

For each request in a disk trace, we recorded its start and end times, the starting sector number, the number of sectors requested, and whether it is a read or a write access. The interval between the start and end times is the disk access time, denoted by T_A, which is defined as:

$$T_A = T_s + T_r + T_t,$$

where T_s is the seek time, T_r is the rotational delay, and T_t is the transfer time. Decomposing T_A is necessary for us to understand the implications of FS^2 on each of these components. If the physical geometry of the disk is known (e.g., number of platters, number of cylinders, number of zones, sector per track, etc.), this can be easily accomplished with the information that was logged. Unfortunately, due to increasingly complex hard drive designs and fiercer market competition, disk manufacturers are no longer disclosing such information to the public. They present only a logical view of the disk, which, unfortunately, bears no resemblance to the actual physical geometry; it provides just enough information to allow BIOS and drivers to function correctly. As a result, it is difficult to decompose a disk access into its different components. Various tools [1, 44, 107, 119] have been implemented to extract the physical disk geometry experimentally, which usually takes a very long time. Using the same empirical approach, we show that a disk can be characterized in seconds for us to accurately decompose T_A into its various components.

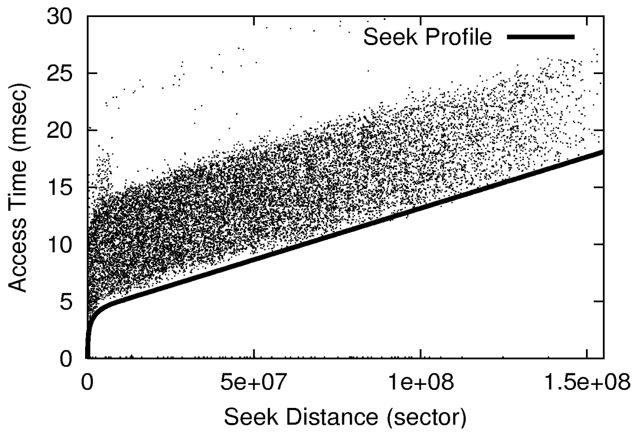


Figure 5.8: This figure shows logical seek distance versus access time for a Western Digital SE 80GB 7200 RPM drive. Each point in the graph represents a recorded disk access.

Using the same experimental setup as described previously, a random disk trace is recorded from the testing machine. From this trace, we observed a clear relationship between logical seek distance (i.e., the distance between consecutive disk requests in unit of sectors) and T_A, which is shown in Figure 5.8. The vertical band in this figure is an artifact of the rotational delay's variability, which, for a 7200 RPM disk, can vary between 0 and 8.33 msec (exactly the height of this band). A closer examination of this figure reveals an irregular bump in the disk access time when seek distance is small. This bump appears to reflect the fact that the seek distance is measured in number of sectors, but the number of sectors per track can vary significantly from the innermost cylinder to the outermost cylinder. Thus, the same logical distance can translate to different physical distances, and this effect is more pronounced when the logical seek distance is small.

As T_A is composed of T_s, T_r, and T_t, by removing the variation caused by T_r, we are left with the sum of T_s and T_t, which is represented by the lower envelope of the band shown in Figure 5.8. As T_t is negligible compared to T_s for a random workload (shown previously in Figure 5.2), the lower envelope essentially represents the seek profile curve of this disk. This is similar to the seek profile curve, in which, the seek distance is measured in the number of cylinders (physical distance) as opposed to the number of sectors (logical distance). As shown in Figure 5.8, the seek time is usually linearly related to the seek distance, but for very short distances, it can be approximated as a third order polynomial equation. We observed that for a seek distance of x sectors, T_s (in unit of msec) can be expressed as:

$$T_s = \begin{cases} 8\text{E-}21x^3 - 2\text{E-}13x^2 + 1\text{E-}6x + 1.35 & x < 1.1 \times 10^7 \\ 9\text{E-}8x + 4.16 & \text{otherwise.} \end{cases}$$

As seek distance can be calculated by taking the difference between the starting sector numbers of consecutive disk requests, seek time, T_s, can be easily computed from the above equation. With the knowledge of T_s, we only have to find either T_r or T_t to completely decompose T_A. We found it easier to derive T_t than T_r. To derive T_t, we first measured the effective I/O bandwidth for each of the bit recording zones on the disk using large sequential I/Os (i.e., so we can eliminate seek and rotational delays). For example, on the outer-most zone, the disk's effective bandwidth was measured to be 54 MB/sec. With each sector being 512 bytes long, it would take 9.0 μsec to transfer a single sector from this zone.

From a disk trace, we know exactly how many sectors are accessed in each disk request, and therefore, the transfer time of each can be easily calculated by multiplying the number of sectors in the request by the per-sector transfer time. Once we have both T_s and T_t, we can trivially derive T_r. Due to disk caching, some disk accesses will have an access time below the seek profile curve, in which case, since there is no mechanical movement, we treat all of their access time as transfer time.

The energy consumed to service a disk request, denoted as E_A, can be calculated from the values of T_s, T_r and T_t as:

$$E_A = T_s \times P_s + (T_r + T_t) \times P_i,$$

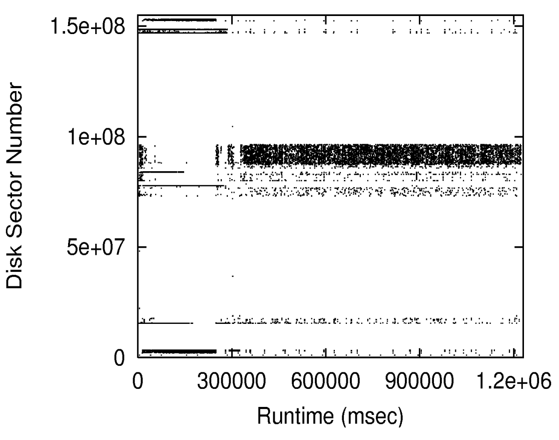
where P_s is the average seek power and P_i is the average idle/read/write power (shown in Table 5.1). The total energy consumed by a disk can be calculated by adding the sums of all E_A's to the idle energy consumed by the disk.

5.5.3 The TPC-W Benchmark

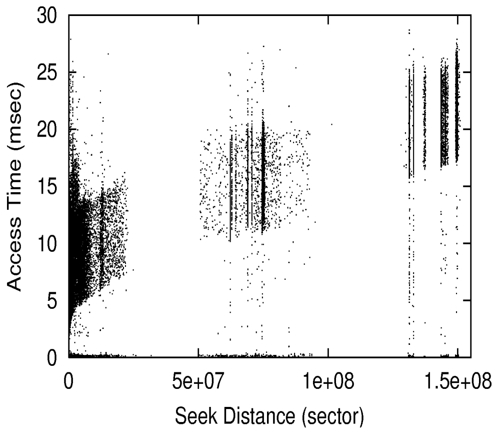
The TPC-W benchmark [15], specified by the Transaction Processing Council (TPC), is a transactional web benchmark that simulates an E-commerce environment. Customers browse and purchase products from a web site composed of multiple servers including an application server, a web server and a database server. In our experiment, we use a MySQL database server and an Apache web server on a single machine, simulating a very small web server with limited traffic. We simulate 25 simultaneous customers accessing this web server for 20 minutes on a separate machine. Complying with the TPC-W specification, we assume a mean session time of 15 minutes and a mean think time of 7 seconds for each client. Now we will compare FS^2 and Ext2 with respect to performance and energy consumption.

Ext2: Base System

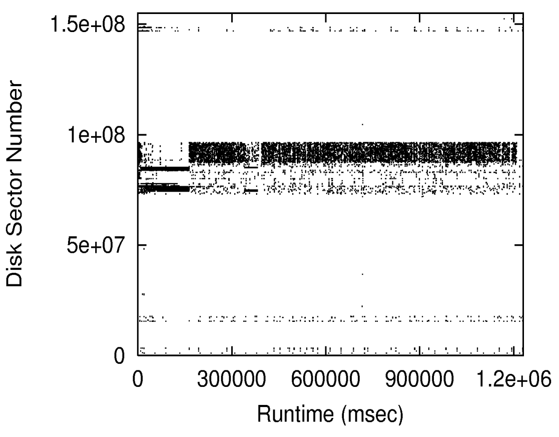
Figure 5.9(a1) shows a disk trace collected from an Ext2 file system when running the TPC-W benchmark. Almost all disk accesses are concentrated in several distinct regions



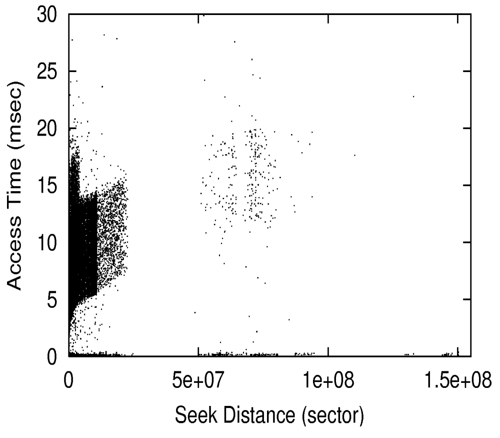
(a1) Ext2 Disk Trace



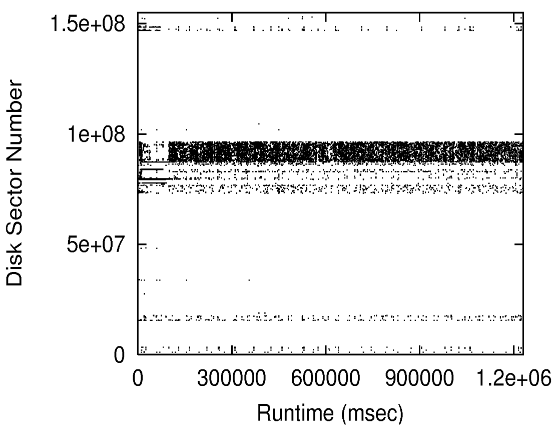
(a2) Ext2 Disk Access Time



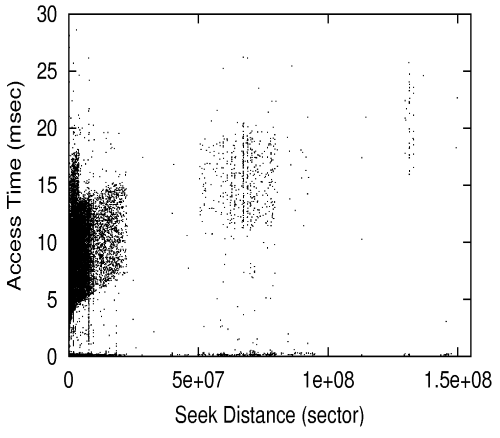
(b1) FS^2-static Disk Trace



(b2) FS^2-static Disk Access Time



(c1) FS^2-dynamic Disk Trace



(c2) FS^2-dynamic Disk Access Time

Figure 5.9: Plots (a1–c1) show the disk sectors that were accessed for the duration of a TPC-W run. Plots (a2–c2) show the measured access times for each request with respect to the seek distance for the TPC-W benchmark. Plots a, b, and c correspond to the results collected on Ext2, FS^2-static, and FS^2-dynamic file systems.

of the disk, containing a mixture of the database's data files and index files, and the web server's image files. The database files can be large, and a single file sometimes can span multiple block groups. Because Ext2 uses a quadratic hash function to choose the next block group from which to allocate disk blocks when the current block group is full, a single database file is often split into multiple segments with each being stored at a different location on the disk. When the file is later (randomly) accessed, the disk head would have to travel frequently between multiple regions. This is illustrated in Figure 5.9(a2), showing a large number of disk accesses having long seek times.

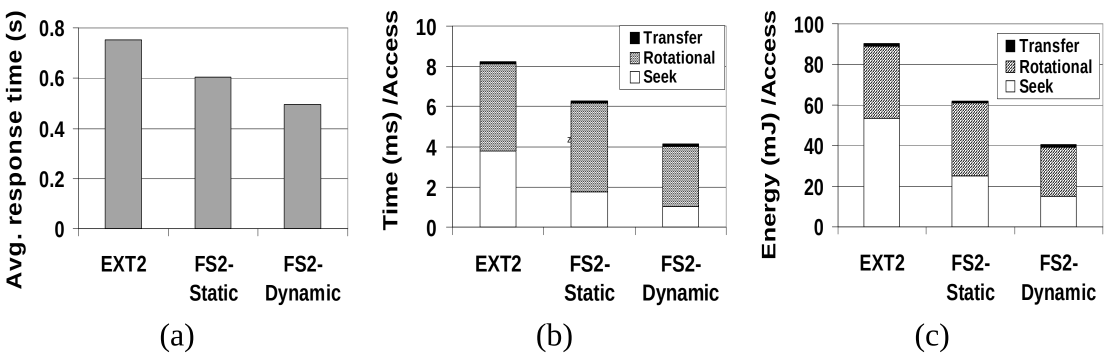


Figure 5.10: For the TPC-W benchmark, part (a) shows the average response time of each transaction perceived by clients. Part (b) shows the average disk access time and its breakdown. Part (c) shows the average energy consumed by each disk access and its breakdown.

	Disk Busy	Performance Improvement			Energy Improvement
		T_A	T_s	T_r	
FS^2-static	23%	24%	53%	-1.6%	31%
FS^2-dynamic	17%	50%	72%	31%	55%

Table 5.2: For the TPC-W benchmark, this table shows percentage improvement in performance and energy-consumption for each disk access, with respect to the Ext2 file system, where the disk was busy 31% of the time.

We observed that the average response time perceived by TPC-W's clients is 750 msec when the benchmark is ran on an Ext2 file system. The average disk access time is found to be 8.2 msec, in which 3.8 msec is due to seek time and 4.3 msec is due to rotational delay. The average energy consumed by each disk access is calculated to be 90 mJ, in which 54

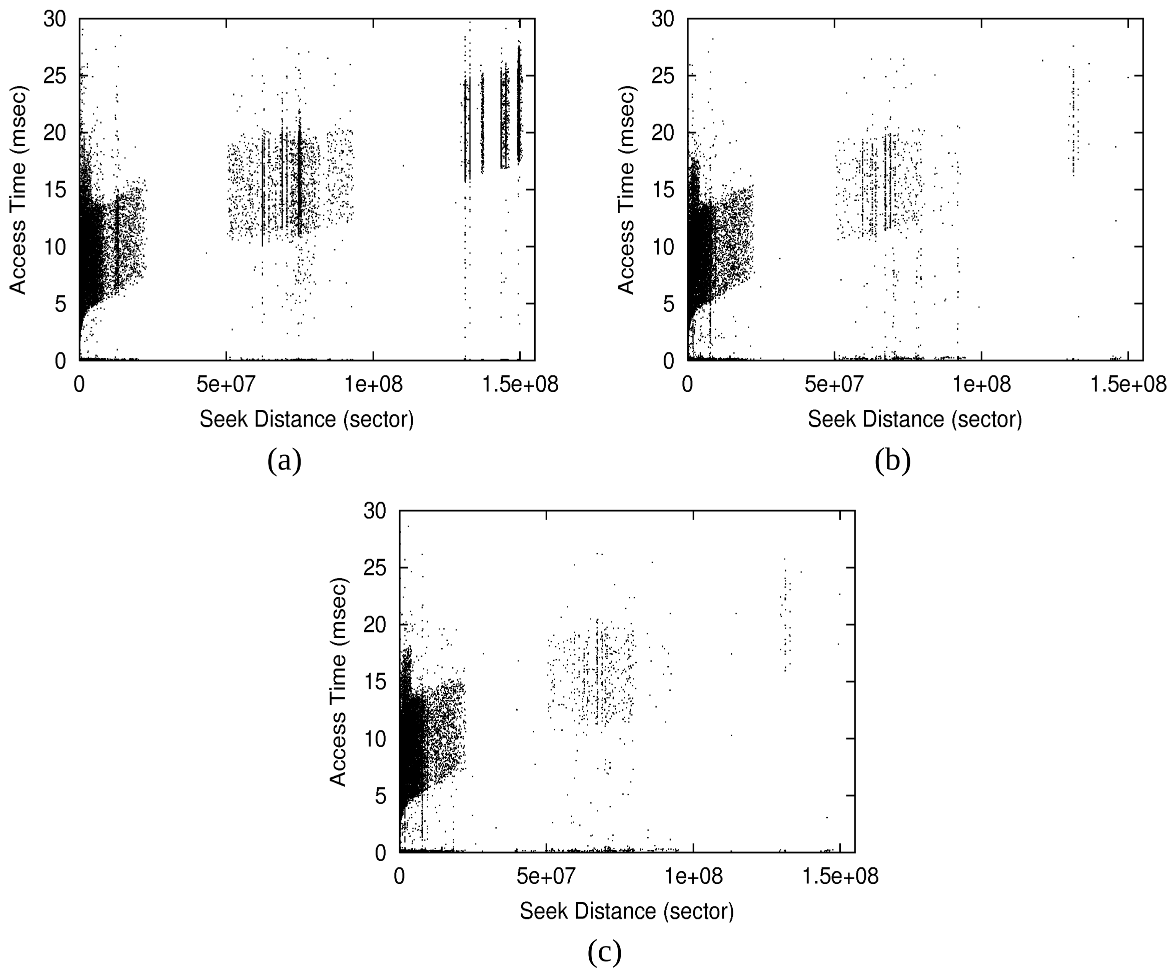


Figure 5.11: For the TPC-W benchmark, parts (a–c) show disk access time for the 1^{st}, the 2^{nd}, and the 7^{th} FS^2-dynamic run.

mJ is due to seek and 36 mJ (including a small amount of energy consumed for transferring data) is due to disk rotation. These results are plotted in Figure 5.10 and then summarized in Table 5.2.

FS^2-Static

From the TPC-W disk trace shown previously, an obvious method to modify the disk layout would be to group all the database and web server files closer to one another on the disk so as to minimize seek distance. To do so, we first find all the disk blocks belong to the database and web server, and then *statically* replicate these blocks (using the *mkfs2* tool) toward the middle of the disk.[2] After the disk layout is modified, we ran the TPC-W workload again, and the resulting disk trace and the access time distribution are plotted in Figures 5.9(b1) and (b2), respectively. One can see from these figures that as replicas can now be accessed in addition to their originals, most disk accesses are concentrated within a single region. This significantly reduces the range of motion of the disk heads.

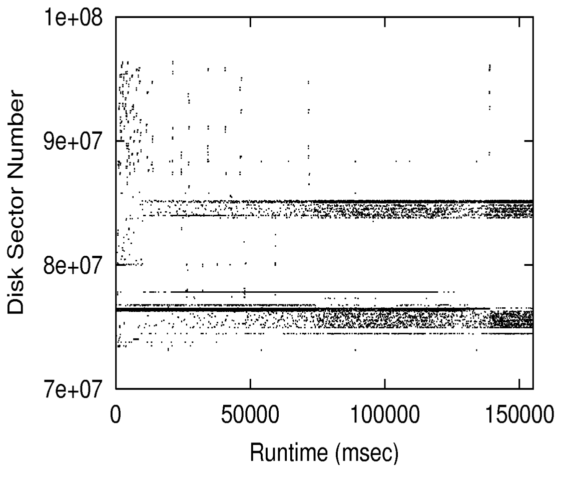
As shown in Figure 5.10, by statically replicating, the average response time perceived by the TPC-W clients is improved by 20% over Ext2. The average disk access time is improved by 24%, which is completely due to having shorter seek time (53%). Rotational delay remains mostly unchanged (-1.6%) for reasons to be given shortly. Due to having shorter average seek distance, the energy consumed per-disk-access is reduced by 31%.

FS^2-Dynamic

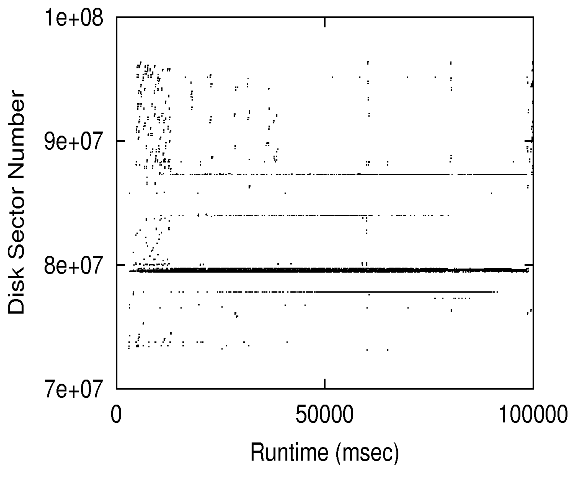
We call the technique described in the previous section *FS^2-static* because it statically modifies the data layout on disks. Using FS^2-static, seek time is significantly reduced. However, this method has no practical use because expecting users to hand-pick candidate blocks to replicate for tuning their file system's data layout is not very realistic. This process needs to be automated. Additionally, static replication is shown to be ineffective in reducing rotational delay. But as we will see for workloads with repeated disk I/O patterns, rotational delay can be reduced. The ineffectiveness in reducing rotational delay is attributed to FS^2-static's rigid aggregation of *all* the database and web server files. Even though relevant disk blocks are much closer to one another, as long as consecutively-accessed data blocks are not placed to the same track with small distance between each pair, reduction in rotational delay is not guaranteed.

As discussed in Section 5.3.2, replicating data and placing them *adjacent* to one another according to the order in which they were accessed not only reduces seek time but can also reduce rotational delay. This technique is called *FS^2-dynamic* as we dynamically monitor disk accesses and replicate candidate blocks on-the-fly without any user interactions. It is implemented as described in Sections 5.3 and 5.4. Starting with no replicas, multiple

[2] We could have chosen any region, but the middle region of the disk is chosen simply because most of the relevant disk blocks were already placed there. Choosing this region, therefore, would result in the fewest number of replications.



(a)



(b)

Figure 5.12: Plots (a) and (b) show the zoom-in section of Figures 5.9 (b1) and (c1), respectively.

runs are needed for the system to determine which blocks to replicate before getting any benefit from the replicas. The replication overhead was observed to be minimal. Even in the worst case (the first run), the average response time perceived by the TPC-W clients was only 1.7% longer (a 0.8% additional energy was consumed). This is due to the additional disk accesses needed for replicating data, which has caused the disk to become 2.5% more utilized. Since the workload is small and has ran for only 20 minutes, none of the replicas made in the first run were accessed during this run; otherwise, the replication cost would be somewhat amortized. To compare FS^2-dynamic with FS^2-static in a workload with repeated disk I/O patterns, we use the same random seed for each TPC-W run. With most replicas being made in the first run, the replication overhead of subsequent runs is almost negligible. The 7^{th} run of FS^2-dynamic is shown in Figures 5.9(c1) and (c2). To illustrate the progression between consecutive runs, we show disk access time distributions of the 1^{st}, the 2^{nd}, and the 7^{th} run in Figure 5.11. During each run, because the middle region of the disk is accessed most, those disk blocks that were accessed in other regions will get replicated here. Consequently, the resulting disk trace looks very similar to that of FS^2-static (shown in Figures 5.9(b1) and (b2)). There are, however, subtle differences between the two, which enable FS^2-dynamic to out-perform its static counterpart.

Figures 5.12(a) and (b) provide a zoom-in view of the first 150 seconds of the FS^2-static disk trace and the first 100 seconds of the FS^2-dynamic disk trace, respectively. In these two cases, the same amount of data was accessed from the disk, giving a clear indication that the dynamic technique is more efficient in replicating data and placing them onto the disk than the static counterpart. As mentioned earlier, the static technique indiscriminately replicates all statically-related disk blocks together without any regard to how the blocks are temporally related. As a result, the distance between temporally-related blocks becomes smaller but often not small enough for them to be located on the same track. On the other hand, as the dynamic technique places replicas onto the disk in the same order that the blocks were previously accessed, it substantially increases the chance for temporally-related data blocks to be on the same cylinder, or even on the same track. From Figure 5.12, we can clearly see that under FS^2-dynamic, the range of the accessed disk sectors is much narrower than that of FS^2-static.

As a result, FS^2-dynamic makes a 34% improvement in the average response time perceived by the TPC-W clients compared to Ext2. The average disk access time is improved by 50%, which is due to a 72% improvement in seek time and a 31% improvement in rotational delay. FS^2-dynamic reduces both seek time and rotational delay more significantly than FS^2-static. Furthermore, it reduces the average energy consumed by each disk access by 55%. However, because the TPC-W benchmark always runs for a fixed amount of time (i.e., 20 minutes), faster disk access time does not reduce this workload's run time, and hence, the amount of energy savings is limited to only 6.7% as idle energy will dominate. In other workloads, when measured over a constant number of disk accesses, as opposed to constant time, the energy savings is more significant.

In this section we introduced FS^2-static and compared it with FS^2-dynamic mainly to illustrate the importance of using online monitoring to reduce mechanical delays and alleviating users from manually choosing candidate blocks to replicate. As the dynamic technique was shown to perform better than the static counterpart, we will henceforth consider only FS^2-dynamic and refer to it simply as FS^2 unless specified otherwise. Moreover, only the 7^{th} run of each workload is used when we present results for FS^2 as we want to give enough time for our system to learn frequently-encountered disk access patterns and to modify the disk layout.

Caveats

In the previous section, we used the same random seed for each TPC-W run, which might have put FS^2-dynamic in more favorable light than what really happen for this particular workload. Using different random seeds for each run would result in less predictable disk I/O patterns, thus reducing replication benefits. This would especially affect the effectiveness of our technique in reducing rotational latency as it is highly depended on having repeated disk I/O patterns. Nevertheless, we believe that given enough training runs in the TPC-W workload, FS^2-dynamic should do as well as FS^2-static minus having to hand-pick blocks to replicate.

As most of the database and web server files are placed at the center of the disk, if we assume random disk accesses in the TPC-W workload, each FS^2-dynamic run would "pull" more data (relevant to the TPC-W workload) that were previously allocated to other disk locations toward the center of the disk. With more and/or longer runs, temporally-related data will be aggregated closer to one another on the disk over time.

In this work, we focus mostly on those workloads with repeated I/O patterns, e.g, starting X server, booting the kernel, CVS operations, etc. We encounter many such workloads on a daily basis, and for such workloads, FS^2-dynamic can seamlessly improve their I/O response time and bandwidth. We will study a few of these workloads in the following sections.

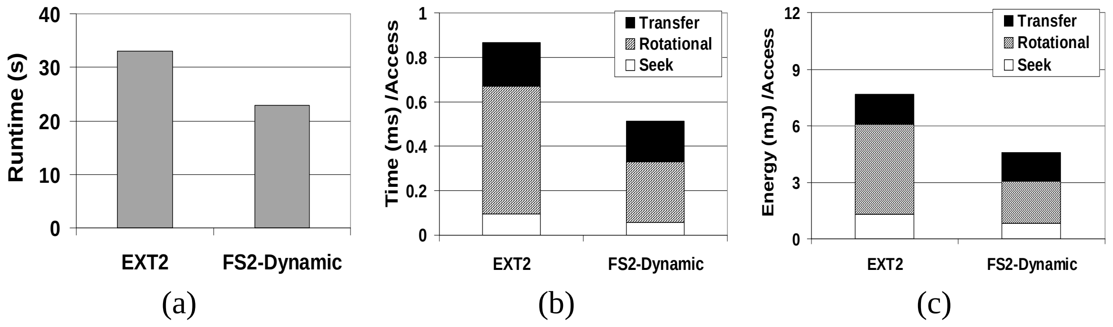


Figure 5.13: Results for the CVS update command: (a) total runtime, (b) average access time, and (c) energy consumed per access.

	Disk Busy	Performance Improvement			Energy Improvement
		T_A	T_s	T_r	
FS^2	73%	41%	38%	53%	40%

Table 5.3: A summary of results for the CVS update command. The disk was busy 82% of the time on Ext2.

5.5.4 CVS

CVS is a versioning control system commonly used in software development environments. It is especially useful for multiple developers who work on the same project and share the same code base. CVS allows each developer to work with his own local source tree and later merge the finished work with a central repository.

CVS update command is one of the most frequently-used commands when working with a CVS-managed source tree. In this benchmark, we ran the command *cvs -q update* in a local CVS directory containing the Linux 2.6.7 kernel source tree. The results are shown in Figure 5.13. On an Ext2 file system, it took 33 seconds to complete. On FS^2, it took only 23 seconds to complete, a 31% shorter runtime. Even on the first run of FS^2, the workload took only 35 seconds to finish, and the additional 2 seconds (6%) is due to replicating a lot of data blocks during this run. A 7% additional energy is consumed.

The average disk access time is improved by 41% as shown in Figure 5.13(b). Rotational delay is improved by 53%. Seek time is also improved (37%), but its effect is insignificant as the disk access time in this workload is mostly dominated by rotational delay.

In addition to the performance improvement, FS^2 also saves energy in performing disk I/Os. Figure 5.13(c) plots the average energy-consumption per disk access for this workload. As shown in Table 5.3, FS^2 reduces the energy-consumption per access by 46%. The total energy dissipated on Ext2 and FS^2 during the entire execution of the CVS update command is 289 J and 199 J, respectively, making a 31% energy savings.

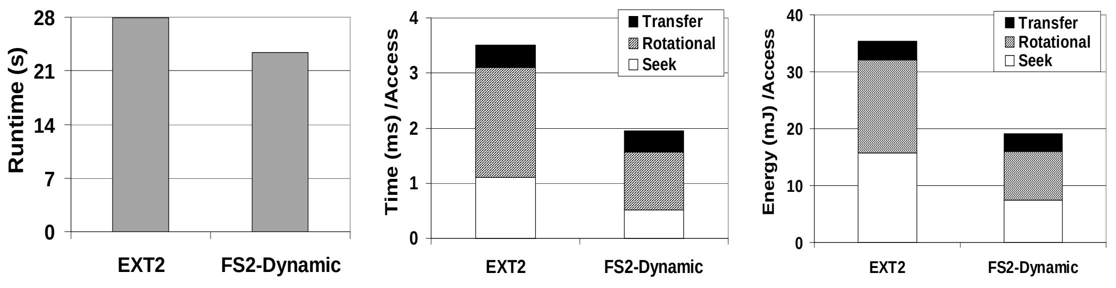


Figure 5.14: Results for starting X server and KDE windows manager: (a) total runtime, (b) average access time, and (c) energy consumed per access.

	Disk Busy	Performance Improvement			Energy Improvement
		T_A	T_s	T_r	
FS^2	26%	44%	53%	47%	46%

Table 5.4: A summary of results for starting X server and KDE windows manager. The disk was busy 37% of the time on Ext2.

5.5.5 Starting X Server and KDE

The *startx* command starts the X server and the KDE windows manager on our testing machine. As shown in Figure 5.14(a), the startx command finished 4.6 seconds faster on FS^2 than Ext2, yielding a 16% runtime reduction. Even though the average disk access time in this workload is significantly improved (44%), which comes as a result of a 53% shorter average seek time and 47% shorter rotational delay (shown in Table 5.4), the runtime is only moderately improved. The reason is that this workload is not disk I/O-intensive, i.e., much less than the CVS workload. For the CVS workload, the disk was busy for 82% of the time, whereas for this workload, the disk was busy for only 37% of the time. Even on the first run of FS^2, it took 1.5 seconds less time and 4.3% less energy, despite having incurred additional disk accesses. We believe this is due to a change in I/O scheduling when additional replicas are made.

In addition to improving performance, FS^2 also reduces the energy consumed per disk access for the startx command as shown in Figure 5.14(c). Starting the X server is calculated to dissipate a total energy of 249 J on Ext2 and 201 J on FS^2, making a 19% savings in energy.

5.5.6 SURGE

SURGE is a network workload generator designed to simulate web traffic by imitating a collection of users accessing a web server [5]. To best utilize the available computing resources, many web servers are supporting multiple virtual hosts. To simulate this type of environment, we set up an Apache web server supporting two virtual hosts servicing static contents on ports 81 and 82 of the testing machine. Each virtual host supports up to 200 simultaneous clients in our experiment.

FS^2 reduced the average response time of the Apache web server by 18% compared to Ext2 as illustrated in Figure 5.15(a). This improvement in the average response time was made possible despite the fact that the disk was utilized for only 13% of the time. We tried to increase the number of simultaneous clients to 300 per virtual host to observe what would happen if the web server was busier, but the network bandwidth bottleneck on our 100 Mb/sec LAN did not allow us to drive up the workload. With more simultaneous clients, we expect to see more performance improvements. However, we show that even for non-disk-bound workloads, performance can still be improved reasonably well. As far as we could see, the first run of FS^2 had little overhead and was comparable with Ext2 runs.

On average, a disk access takes 68% less time on FS^2 than on Ext2. In this particular workload, reductions in seek time and rotational delay contributed almost equally in lowering the disk access time, as shown in Table 5.5.

The total energy consumed was reduced by merely 1.9% on FS^2 because this workload always runs for a fixed amount time like the TPC-W workload. There might be other energy-saving opportunities. In particular, FS^2 allows disk to become idle more often—in the TPC-W workload, the disk was idle 70% of the time on Ext2 and 83% on FS^2, while in the CVS workload, the disk was idle 18% of the time on Ext2 and 27% on FS^2. Combined with traditional disk power-management techniques, which save energy during idle intervals, more energy-saving opportunities can be exploited.

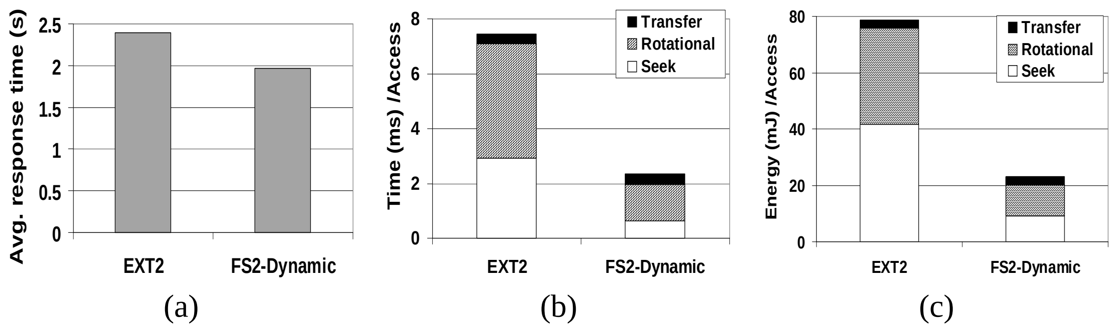


Figure 5.15: Results for SURGE: (a) average response time, (b) average access time, and (c) energy consumed per access.

	Disk	Performance Improvement			Energy
	Busy	T_A	T_s	T_r	Improvement
FS^2	4.2%	69%	78%	68%	71%

Table 5.5: A summary of results for running SURGE. The disk was busy 13% of the time on Ext2.

5.6 Discussion

5.6.1 Degree of Replication

Our implementation of FS^2 allows system administrators to set the maximum degree of replication; in an n-replica system, a disk block can be replicated at most n times. According to this definition, traditional file systems are 0-replica systems. There are certain tradeoffs in choosing the parameter n. Obviously, with a large n, replicas can potentially consume more disk space. However, allowing a larger degree of replication may more significantly improve performance under certain situations. One simple example is the placement of shared libraries. Placing multiple copies of a shared library close to each application image that makes use of it, application startup time can be reduced. It is interesting to find the "optimal" degree of replication for a certain disk access pattern. However, in the workloads that we have studied, due to lack of data/code sharing, the additional benefit from having multiple replicas per disk block would be minimal, and therefore, we simply used a 1-replica system in our evaluation. Lo [78] explored the effect of degree of replication in more details through a trace-driven analysis.

5.6.2 Reducing Replication Overhead

Placing replicas of data in free space near the current disk head location minimizes the interference with foreground tasks. Using *freeblock scheduling* [82, 83], it is possible to further reduce this overhead by writing replicas to disk only during rotational delay. However, as indicated by our experimental results, our heuristic is already fairly effective in keeping the replication overhead low.

5.6.3 Multi-Disk Issues

The most straightforward way to improve disk performance is to spread data across more disks. Redundant Array of Inexpensive Disks (RAID) was first introduced by Patterson *et al.* [94] as a way to improve aggregated I/O performance by exploiting parallelism between multiple disks, and since then it has become widely used.

FS^2 is operating orthogonally to RAID. The techniques that we have used to improve performance on a single disk can be directly applied to each of the disks in a RAID. Therefore, the same amount of performance improvement that we have seen before can still be achieved here. The problem gets more interesting when free disk space from multiple disks can be utilized. It can be used not only to improve performance, but also to enhance the fault-tolerance of data, complementing RAID. For example, a traditional RAID-5 system can sustain only one disk failure before data loss. However, if we are able to use free disk space on one disk to hold recently-modified data on other disks, it may be possible that even with more than one disk failure, all data can still be recovered—recently-modified data can be recovered from replicas on non-faulty disks and older data can be recovered from a nightly or weekly backup system.

This introduces another dimension where free disk space can be utilized to our advantage. Now, we can use free space of a disk to hold either replicas of its own blocks or replicas from other disks. Both have performance benefits, but holding replicas for other disks can also improve fault-tolerance. These issues are beyond the scope of this work. We are currently investigating these issues and building a RAID-based prototype.

5.6.4 Journaling File System

Journaling file systems, e.g., Ext3, are likely to replace traditional file systems, e.g., Ext2, in the near future as they can quickly convert a crashed file system back to a consistent state. This is done by committing data and meta-data to a journal file in an orderly fashion and having them written to disk twice. The concept of FS^2 can be potentially applied to a journaling file system. The advantage of applying FS^2 ideas to a journaling file system is that replicas already exist in the journal and we do not need to make additional replications. If the journal is persistently kept, we will always have two copies of everything, both data and meta-data. In which case, the problem of selecting candidate blocks to replicate becomes the problem of invalidating excessive replicas. Additionally, by classifying the hash table as another type of meta-data and committing it to the journal in an orderly fashion, some of the consistency problems caused by system crashing discussed in Section 5.4.3 can be avoided.

Despite these advantages, there are several difficulties related to this approach. First, using the journal to hold replicas intrinsically limits the file system to an 1-replica system (unless the implementation of the journal is extensively modified.) Second, disk blocks used by the journal are best kept sequential to optimize for write operations. Sequentially placing replicas at a single location on disk, however, will not effectively minimize head positioning latencies. Finally, keeping a large number of replicas persistently in the journal can slow down operations on directory indexes that are used to keep track of the journal's data. As we can see, realizing FS^2 ideas in a journaling file system has some practical challenges. Therefore, additional analysis is needed before these ideas can be integrated seamlessly with a journaling file system.

5.7 Conclusions

In this chapter, we presented the design, implementation and evaluation of a new file system, called FS^2, which contains a runtime component responsible for dynamically reorganizing disk layout. The use of free disk space allows a flexible way to improve disks' I/O performance and energy-efficiency, while being nonintrusive to users.

To evaluate the effectiveness of FS^2, we conducted experiments using a range of workloads. FS^2 is shown to improve the client-perceived response time by 34% in the TPC-W workload. In SURGE, the average response time is improved by 18%. Even for everyday tasks, FS^2 is shown to provide significant performance benefits: the time to complete a CVS update command is reduced by 31% and the time to start an X server and a KDE windows manager is reduced by 16%. As a result of reducing seek distance, energy dissipated due to seeking is also reduced. Overall, FS^2 is shown to have reduced the total energy consumption of the disk by 1.9–15%.

CHAPTER 6

Disk Failure Characterization

6.1 Motivation

In the previous chapter, we described the design, implementation, and evaluation of the Free Space File System (FS^2). By means of replicating disk blocks and placing the replicas in unused disk regions, we have shown both energy and performance benefits while keeping overheads to a minimum. In addition to improving energy-efficiency and performance, we believe replication is also an effective method to increase data fault-tolerance and can be practically implemented in existing file systems. Replication is already a common technique used by RAID in multi-disk environments, but it has not been successfully applied to systems with only a single disk.

Unlike FS^2, in which it is fairly easy to determine which disk blocks to replicate (those with long access time), where to replicate (the unused disk regions close or adjacent to the temporally-related blocks), and when to commit replicas to disk (anytime the replicas are still in memory) to improve performance and energy-efficiency, replicating disk blocks to improve fault-tolerance involves more complex decision-making.

Determining the candidate blocks to replicate for improving fault-tolerance involves higher level knowledge than what can be observed at the block device level, making access frequency information meaningless (more frequently accessed data does not imply higher importance). This process should happen at the file system level by mining the file system's meta-data or getting direct inputs from users. For example, knowing that files in */etc* and */home* directories are more valuable than those in */tmp* or */opt* directory, we could give higher priority to replicate the first set of files than the second. We could also use files' meta-data to determine their relative importance, e.g., give higher priority to non-executable files (word documents and source files) than executable (programs and shared libraries) as the latter can often be recovered (e.g., re-installed) or regenerated (e.g., re-compiled) fairly easily.

However, automatically mining the file system to find candidate blocks is not always sufficient; user's involvement can sometimes significantly improve the effectiveness of this process. A user can write rules to specify replication strategy for his / her private files and directories, e.g., give higher priority to replicate *.c files and the files in the *works* directory. Additionally, sometimes common sense can also help us find candidate blocks and calculate their relative importance. For example, as damaging a small amount of file system's meta-data could render the entire file system inaccessible, it makes sense to give higher priority to replicate the meta-data than user data.

Both system-wide and user-specific rules can drive and give hints to the replication engine on which disk blocks to replicate and the blocks' relative importance. These rules can be easily specified by system administrators and users, and the rules are highly tolerable of errors. In case where an error is made in these rules, only the effectiveness of replication will be affected, and the system will continue to function correctly.

A more critical decision than determining which disk blocks to replicate is of which location we should place their replicas. Determining the relative location of the replicas with respect to the location of the original block is crucial to the performance and fault-tolerance of any file system that makes use of replication. To ensure high fault-tolerance, we would like to keep replicas as far away as possible from the location of the original data to decrease the probability of a permanent data loss when the original data and all its replicas are simultaneously compromised. However, by placing replicas far away from their original data, changing the content of the original data would require synchronizing all its replicas, which can result in long mechanical delays and unnecessarily hurt disk I/O performance when the replicas are placed far away. Even if we can avoid performance penalties when placing replicas far away from each other and their original (e.g., by using asynchronous update operations), it is unclear how we can place disk blocks *far away* from each other as the physical geometry of today's disks is often hidden from us and abstracted using logical addresses. For example, on a particular disk, two sectors that are separated by 1,000,000 sectors (logical address) might be located on different zones, different surfaces, and different platters, but on another disk, the same two sectors that are 1,000,0000 sectors apart might be located on the same zone, same surface, and the same platter. Therefore, using logical address to place replicas *far away* from each other might not necessarily achieve the expected result.

To provide an effective way to improve the fault-tolerance of a single disk, there are several challenges to overcome. First, we must be able to accurately and quickly detect the physical geometry of disks and expose this information so that the file system can use these physical attributes to better safeguard data. Having precise information about disk's physical geometry allows file system to accurately place data and their replicas to minimize the probability of a permanent data loss. For example, if we know that the correlation of disk blocks located on different platters are simultaneously destroyed is extremely low, we should avoid placing a replica on the same platter as its original data block. Our second challenge is to find ways to better understand disk failure characteristics as this is still an area where disk manufacturers are holding back a lot of information. We do not have precise data on how disk drives commonly fail or how data become lost or corrupted in the process. In other words, whether placing a replica on a different platter, or a different track, or a different surface from the location of its original data is the most effective way to improve data fault-tolerance is difficult to determine. Even more difficult is the fact that we do not know whether the most effective method to replicate data is the same or different among disks of different manufacturers and generations.

Understanding disk failure characteristics is the goal of this chapter. With a better understanding, future file systems can more efficiently utilize unused disk capacity for replication to better guard against unexpected data loss. Section 6.2, we discuss the algorithm we used to probe for disk's physical geometry. Section 6.3 describes our study of 60 failed hard drives, and finally, Section 6.4 concludes this chapter.

6.2 Probing for Physical Geometry

A disk is consisted of one or more platters, which rotate in lockstep fashion at a fixed rotational speed. A platter has either one or two recording surfaces, each with a read / write head. A surface is divided into multiple zones with each containing consecutive tracks that have the same number of sectors. Cylinders on the outer zones have more sectors per track, and thus, can achieve a higher I/O throughput. A cylinder is a vertical set of tracks (one per surface) that are the same distance away from the center of the disk.

Despite disk manufacturers' usual unwillingness to publish information about their disks' physical geometry, some physical attributes are always published, e.g., rotational speed and the number of platters and recording surfaces. However, to accurately determine the physical location of any given sector, we need to know some additional information about the disk. Namely, we need to know the number of zones and the number of sectors per track. The former can be obtained if the latter is known as sectors-per-track is distinct for each zone.

There are several previously-proposed techniques [1, 29, 44, 107, 119] to extract disks' physical features. Worthington and Schindler [29, 107] relied on interrogative SCSI commands for extraction, but unfortunately, this technique is not applicable to IDE disks. Experimental methods using empirical observations were first proposed by Aboutabl *et al.* [1] and later improved by Talagala *et al.* [119] and Dimitrijevic *et al.* [44]. The probing algorithm we propose here is also based on observing empirical results. However, unlike the previous work, which tires to extract as much information as possible about disks (even though some information is already published), we only need to extract the sectors-per-track information for our purpose.

When extracting the sectors-per-track information, we want to do it as quickly as possible if the extraction happens offline or want to incur as small number of I/O operations as possible if it is done online. We believe our technique performs well in both modes. The probing algorithm is presented in Figure 6.1.

Essentially, our algorithm is similar to the technique proposed by Dimitrijevic [44]—sequentially write to disk, two sectors at a time, and compare its access time with the access

```
#Definitions
DEFINE  SECTOR_SIZE          512
DEFINE  SETTLE_TIME          500
DEFINE  BACKTRACK_SECTORS    5
DEFINE  REQUIRED_CONFIRMS    10
DEFINE  FULL_ROTATION        8333

# Initialization
i_save            = 0;
confirmed         = 0;
warp_location     = 0;
warp_multiplier   = 1;
last_spt          = 0xbeefbeef;
last_atime        = FULL_ROTATION;

FOR (i = 0; i <= end_sector; i++)
    # Seek to the next sector and write 2 sectors
    lseek(i * SECTOR_SIZE);
    gettimeofday(&t1);
    write(2 * SECTOR_SIZE);
    gettimeofday(&t2);
    atime = t2 − t1;
    diff = abs(atime − last_atime);
    # True when this might be a track/cylinder boundary
    IF ((diff > SETTLE_TIME) AND
        (i_saved = i OR confirmed = 0))
        IF (++confirmed = REQUIRED_CONFIRMS)
            # We have confirmed multiple times this is a track/cylinder
            # boundary and we modify warp_multiplier to skip ahead
            IF (warp_multiplier > 1)
                IF ((i − warp_location + 1) MOD last_spt = 0)
                    warp_multiplier *= 2;
                ELSE
                    # Warped to a wrong location so we go back
                    warp_multiplier = 1;
                    i = warp_location + last_spt − BACKTRACK_SECTORS;
                    continue;
            ELSE
                IF (i − warp_location + 1 == last_spt)
                    warp_multiplier = 2;
                ELSE last_spt = i − warp_location + 1;
            # From warp_location to i, sectors_per_track is last_spt
            warp_location = i + 1;
            confirmed = 0;
            i += last_spt * warp_multiplier − BACKTRACK_SECTORS;
        ELSE
            # We backtrack multiple times to make sure that sector i
            # is really a track/cylinder boundary
            i_saved = i;
            backtrack_sectors = BACKTRACK_SECTORS;
            i −= BACKTRACK_SECTORS;
    # Sector i does not seem to be a track/cylinder boundary
    ELSE
        IF (confirmed && −−backtrack_sectors == 0)
            # After backtracked, we failed to confirm
            confirmed = 0;
        last_atime = atime;
```

Figure 6.1: The proposed algorithm to quickly and accurately probe for the sectors-per-track information. Some boundary cases are not included to make the algorithm more succinct and readable.

Sectors	Tracks	SPT
0–503	1	504 per track
504–1007	1	504 per track
1008–2015	2	504 per track
2016–4031	4	504 per track
4032–8063	8	504 per track
8064–16127	16	504 per track
16128–32255	32	504 per track
32256–64511	64	504 per track
...		

Table 6.1: Sectors-per-track probed for an IBM Ultrastar disk. From sector 0 to 503, we write 2 sectors at a time and find that sector 503 is a track boundary. We then make the assumption that the following tracks also have 504 sectors. Instead of writing sequentially, we write only to the assumed boundary region. If we guessed correctly, we can leap ahead and quickly finish the probing process.

time of the two previously-accessed sectors. If the difference between the two access times is greater than a certain threshold, it is marked as a cylinder or a track boundary. In our algorithm, when we find a boundary, we try to backtrack and re-probe multiple times to make certain that the boundary found is really a boundary because transient behaviors, especially in IDE disks, are fairly common. By backtracking and rewriting multiple times, we can eliminate inaccuracies in our result. Additionally, to speed up the probing process and minimize the number of I/O operations, we take advantage of the fact that there is a very high probability that the following tracks will have the same number of sectors on them as the track we just probed. This allows us to *skip ahead* by hundreds of sectors and probe only the disk region where we think the next boundary is located. If we have guessed these boundaries correctly multiple times, we will start *warping ahead* in multiples of tracks, which can further speed up the probing process. When we guessed correctly, especially in the latter case, which happens often, we only need a few disk accesses to probe hundreds of tracks as opposed to hundreds of accesses for a single track. If we guessed wrong, we can always backtrack. This rarely happens, and even if it does happen, the overhead of making the necessary correction is very small. A sample of the extracted sectors-per-track information for an IBM Ultrastar disk is shown in Table 6.1.

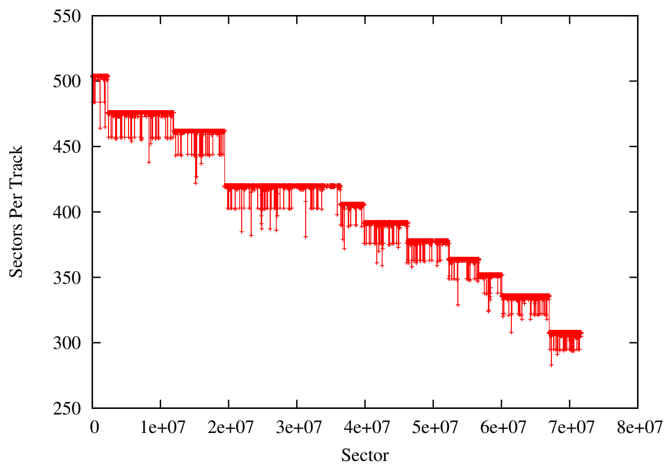


Figure 6.2: A complete extraction of sectors-per-track for the entire IBM Ultrastar disk.

The sectors-per-track information we extracted from the IBM disk is shown in Figure 6.2. From this figure, we can clearly see that the disk has 11 zones. On each zone, most of the tracks have the same number of sectors, but there are some variations. These variations are caused by slip sectors and spare sectors. Slip sectors are used to skip physical defects on media surfaces created during manufacturing time, and they reduce the number of usable sectors on the affected tracks. Spare sectors are used to remap bad sectors created at runtime. In case a good sector becomes defective, disk firmware will automatically remap the defective sector to one of the closeby spare sectors. Spare sectors also cause the number of usable sectors to decrease on the tracks where they are pre-allocated. As we will see in the next section, these variations are important to take into account when characterizing disk failures.

6.3 A Case Study of Failed Disks

To study failed disks, we must first have a large number of them in our possession, which is a challenge by itself. Privacy is a major concern in the process of obtaining such disks as data can sometimes be recovered partially or even completely from these disks. After promising a safe and complete disposal of all content on disk and signing the required paperwork, we received 60 failed disks—10 from EECS's Department Computing

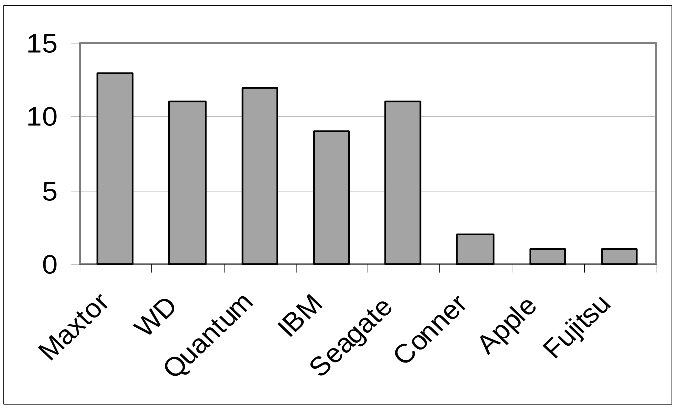


Figure 6.3: Categorizing the 60 failed disks by their manufacturers.

Organization (DCO) and another 50 from University of Michigan's Property Disposition (PD).

6.3.1 Overview

The 60 failed disks we are studying in this chapter are shown in Figure 6.3 grouped by their manufacturers. Not surprisingly, most are produced by some of the larger disk manufacturers, i.e., Maxtor, Western Digital, Seagate, Quantum, and IBM, simply because these companies have a much larger market share than the smaller companies. However, the relative number of these disks does not accurately reflect their manufacturer's true market share as (i) the sample size used in our study is small, and (ii) disks made by some companies are more prone to failures than others.

We categorize these failed disks into three groups—non-bootable disks, disks with bad sectors, and perfectly healthy disks. We now describe each of the three categories in more details.

- **Non-bootable:** These are disks not recognizable by BIOS at the machine power-on time, and they account for more than half of our disks (37). This observation initially led us to believe that a disk will become useless when a failure occurs. However, after studying these non-bootable disks in more details, we negated our hypothesis. Instead, we believe the reason these disks are now in an non-bootable state is highly unlikely the same reason they were initially taken out of commission. All 37 disks came from the batch of disks we obtained from Property Disposition (PD) with most having apparent external physical damages (e.g., dents and broken interface pins). Having external damage on a disk is highly unusual as disks are internal components that are always physically protected by a sturdy external computer case. We believe these external damages should have happened after the disks were diagnosed as having problems and were taken out of the system. Therefore, it is most likely these damages were incurred during the handling process at PD. As disks in this condition cannot be accessed without using specialized equipment, it is impossible for us to assess the extent of damage on their recording surfaces to characterize such failures. Therefore, we have to exclude this type of failures from our analysis.

- **Bad-sectors:** These are bootable disks (9) with at least one detected bad sector. Having one bad sector out of billions of sectors on a disk is not that uncommon. However, losing 512 bits of information can either be harmless if the affected sector is not currently mapped to any data or can be very destructive if the affected sector contains important file system meta-data. A sector can become corrupted for various reasons: material degeneration (caused by overuse or old age) and head crash. By identifying and mapping the exact physical location of bad sectors, we hope to gain a better understanding of how data became corrupted due to disk problems. With this knowledge, we can build file systems that can place data on disk in a more fault-tolerant manner and use data replication techniques to better guard against the unexpected.

- **Healthy:** Ironically, some "failed" disks (14) are actually perfectly healthy, but for some reason, were identified as bad and taken out of commission. These disks have no bad sectors nor any anomalies in their SMART status. SMART stands for Self-Monitoring Analysis and Reporting Technology, and it monitors mechanical and electronic components on disk to give advance warnings to predictable failures. We believe there are several possibilities for which these healthy disks were falsely identified. One possibility is bit flipping, which happens approximately once for every 10^{14} bits accessed. Another possibility is bad writes—sometime a write operation

Capacity	75 GB
Zones	15
Cylinders	27724
Platters	5
Surfaces	10
RPM	7200
Interface	IDE

Table 6.2: IBM 75GXP: Disk specification.

can be carried out at a wrong location or in between two tracks. Besides these hardware errors, bugs in the file system and human errors could also have caused healthy disks to be falsely identified. But as file system code is becoming very mature and good management software is becoming more commonly deployed to minimize human errors, we believe concentrating on disk hardware errors is more fruitful and beneficial. Using replication can also help alleviate this type of disk failures.

Our focus is to analyze the disks with bad sectors. Among the 9 disks, there are 2 IBM, 2 Western Digital, 2 Quantum, 2 Maxtor, and 1 Seagate disks. The Seagate disk has only 1 defective sector and is not interesting to discuss, and therefore, it will not be included in our discussion. We will now characterize and dissect the extent of damage on the 8 remaining disks.

6.3.2 IBM Deskstar 75GXP

One of the disks we studied is an IBM 75GXP. Its specification is listed in Table 6.2 showing its capacity, revolutions per second (RPM), interface type, and its number of zones, cylinders, platters, and recording surfaces. This information (except zones) is generally made public for modern disks and does not require probing. By using commercial software to scan its recording surfaces, we are able to pin-point all its bad sectors. The location of these corrupted sectors is critical in determining how data become lost or damaged.

Figure 6.4 shows a list of (575) bad sectors we found on the IBM disk. They are located in only low-address regions on the disk, which indicates that by simply separating replicas from their original data block by a reasonably large logical distance (in unit of sectors), the data can be stored more safely. This is the conventional and the most straight-forward

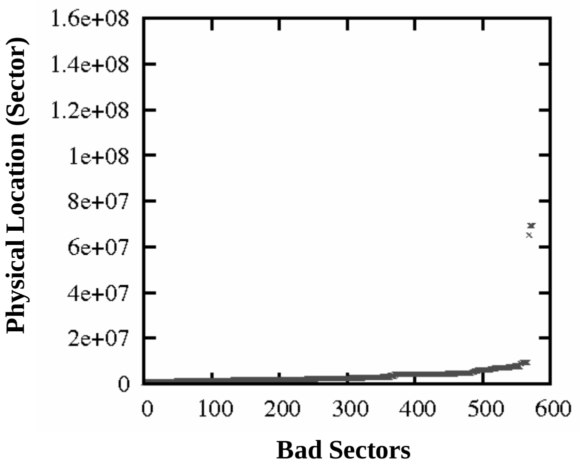


Figure 6.4: IBM 75GXP: The disk's list of bad sectors.

way of placing replicas to improve fault-tolerance on a single disk. As we have pointed out earlier, placing related data far away from each other can significantly hurt performance. We hope, from studying disk failures in more details, we can devise better strategies in placing data and their replicas.

Zones	Cyl start	Cyl end	Sectors/Track
0	0	1375	702
1	1376	2831	684
2	2832	4239	666
3	4240	6975	648
4	6976	9759	612
5	9760	11551	594
6	11552	13631	567
7	13632	16239	540
8	16240	18319	504
9	18320	19567	486
10	19568	21199	459
11	21200	23519	432
12	23520	25215	396
13	25216	26319	378
14	26320	27724	351

Table 6.3: IBM 75GXP: disk physical geometry.

Analyzing the physical locations of bad sectors (as opposed to logical locations and distances) is the key in gaining a better understanding of disk failures. Mapping the logical address of bad sectors to their physical location would require a detailed map of the disk's physical geometry. Fortunately, the physical geometry (shown in Figure 6.3) of this particular disk model has already been made available in the public domain, and no additional work is needed to probe for it. Using the published physical geometry information, it is straight-forward to map bad sectors to their physical location so we know exactly which platter, surface, zone, cylinder, and track on which each of the defective sectors is located.

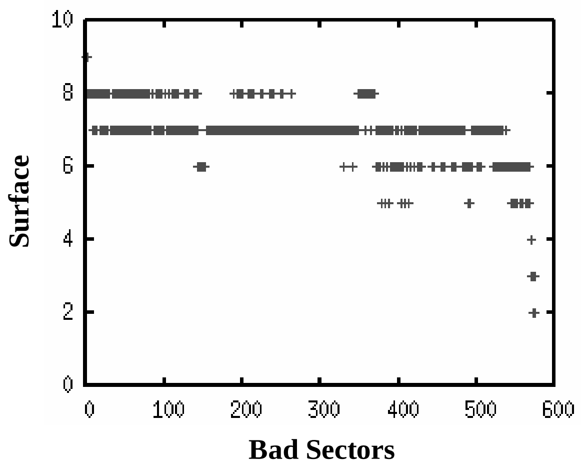


Figure 6.5: IBM 75GXP: A mapping of bad sectors to recording surfaces using the published physical geometry.

Using the published specification of the disk, we first mapped each of the bad sectors to its recording surface, which is shown in Figure 6.5. This figure indicates that surface #7 has suffered the most amount of damage with 352 bad sectors. Other damaged sectors are scattered across surfaces #2, #3, #4, #5, #6, #8, and #9, indicating that bad sectors exhibit very poor physical locality. This observation was disappointing and counter-intuitive because it makes sense that some disk regions, such as the outer-most recording surfaces, are more prone to damage than others due to a difference in air pressure within the disk's enclosure that pushes free particles toward the outer-most recording surfaces. Or regions that had been previously-damaged are more prone to failures than others. None of these hypotheses are supported in our observation from the previous figure.

A closer look at our result led us to find a critical mistake in our calculation by placing too much trust in the published disk geometry. As it turns out, the real disk geometry is actually different, although not by much, from the published data. However, the small difference between the two, shown in Figure 6.6, made a huge difference in our analysis. This difference is caused by slip sectors and spare sectors that are not taken into account when the disk's geometry is specified in its data sheet. Their omission is understandable

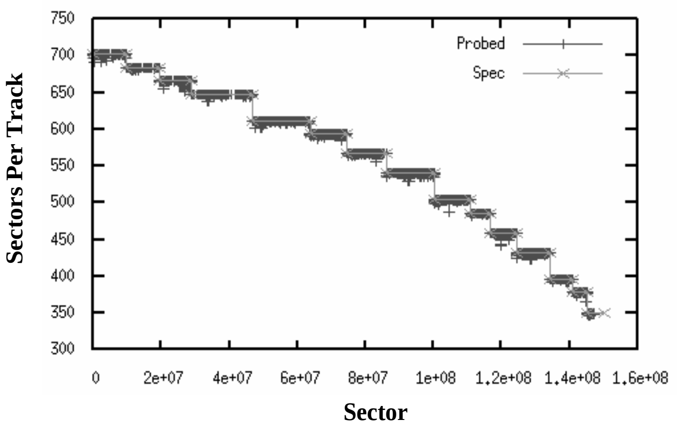


Figure 6.6: IBM 75GXP: Comparing the *actual* physical geometry (Probed) of the disk with its published data (Spec).

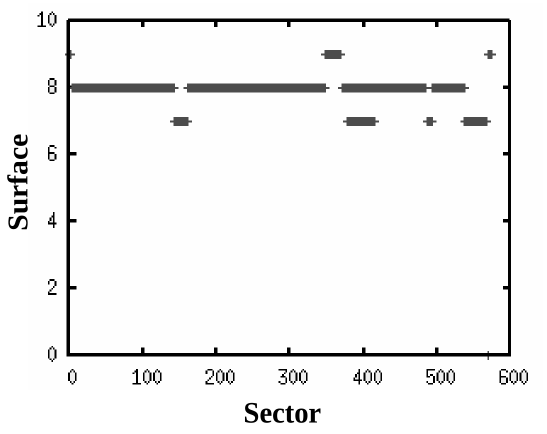


Figure 6.7: IBM 75GXP: A mapping of bad sectors to recording surfaces using the actual physical geometry of the disk.

as each disk is different. To obtain the disk's *actual* physical geometry, we used the probe method described in the previous section. The mapping of the bad sectors to recording surfaces using the disk's actual geometry is shown in Figure 6.7.

Using the actual physical geometry of the disk to map out bad sectors we found that surface #8 had the most number of damaged sectors (467). Additionally, other bad sectors, instead of being scattered across many different recording surfaces, are now localized to only two neighboring surfaces #7 and #9. This finding implies that it is not necessary to have replicas to be placed far apart to store data more safely. Sometimes, it is sufficient to place the replicas to a close-by track as long as they are on a different surface from the location of their original data. This allows data to be stored as safely as the traditional method (i.e., placing replicas far apart from their original) while keeping performance degradation to a minimum for keeping replicas consistent.

Sector	Track	
813160	1158	10 tracks
819801	1168	10 tracks
827144	1178	10 tracks
833786	1188	...
841128	1198	
847770	1208	
855112	1218	
862455	1229	
869096	1238	
876439	1248	
883080	1258	
890423	1268	
897064	1278	
904407	1288	
911048	1298	
918391	1308	
925032	1318	
932375	1328	
939017	1338	
946359	1348	
953702	1359	
960343	1368	
967686	1378	
974327	1388	
...	...	

Table 6.4: IBM 75GXP: A sample mapping of bad sectors to their track.

We next mapped the defective sectors to tracks, and a sample of this mapping is shown in Table 6.4. A closer examination of this table shows many of these sectors are often 10 tracks apart from each other. The fact this disk has 5 platters and 10 recording surfaces implies these sectors are on neighboring tracks of the same surface. Plotting their sector offset (within a track), as shown in Figure 6.8, we can see there is a clear pattern. We immediately suspected the regular pattern in the sector offsets is an artifact of head skews and cylinder skews. It is also an indication that these sectors might be located closely to one another on neighboring tracks, e.g., as part of a scratch or a hole on a surface.

To verify our hypothesis is correct, we first obtained the disk's head skew (HS) time (1.2 ms) and cylinder skew (CS) time (1.7 ms) from its specification sheet. Instead of using the actual time, we converted the time unit to the number of sectors that would pass under the disk head for that amount of time, as shown below.

$$Head\_Skew_{sectors} = \frac{Head\_Skew_{time}}{Full\_Rotation_{time}} \times SPT$$

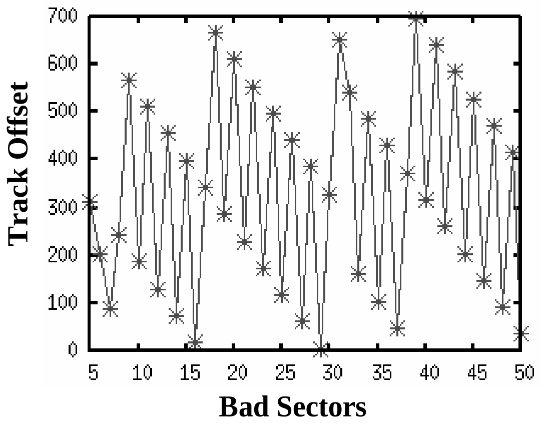


Figure 6.8: IBM 75GXP: A sample of bad sectors' sector offsets.

$$Cylinder\_Skew_{sectors} = \frac{Cylinder\_Skew_{time}}{Full\_Rotation_{time}} \times SPT.$$

A full rotation takes 8.33 ms on a 7200 RPM disk, and SPT stands for sectors per track. HS and CS are found to be 102 and 144 sectors, respectively. Next, we calculate the skewness of neighboring tracks on the same surface, called surface skew, as shown below.

$$Surface\_Skew_{sectors} = (HS_{sectors} \times (Surfaces - 1) + CS_{sectors})\%SPT.$$

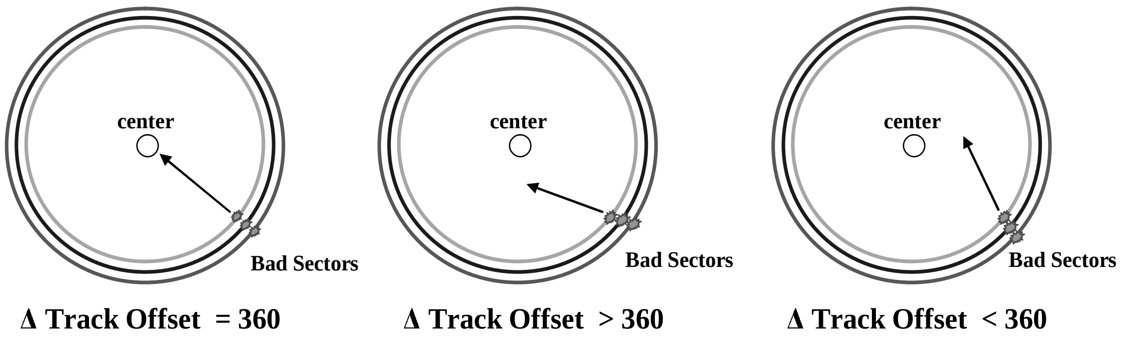


Figure 6.9: Using sector offset difference to predict sectors' relative location.

The surface skew (SS) of this disk is approximately 360 sectors. This means if two sectors (on neighboring tracks of the same surface) have a sector offset difference of 360 sectors, they will fall on the same radius line passing through the center of the disk. An example illustrating different variations of this scenario is shown in Figure 6.9.

The sector offset difference between bad sectors of neighboring tracks is shown in Figure 6.10. Their offset difference lies consistently around 375–380 sectors—a strong

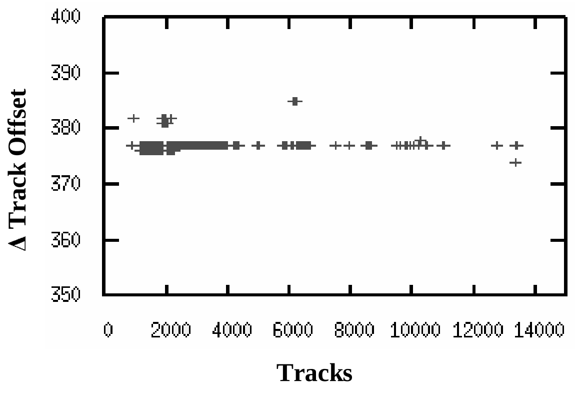


Figure 6.10: IBM 75GXP: Difference between bad sectors' sector offsets.

indication of surface scratches. This might be caused by a head crash as a head will leave a regular scratch pattern on the recording surface every time it crashes. The analysis of sector offset allows us to propose an alternative and potentially better method to replicate data—placing replicas almost physically adjacent (i.e., on a neighboring track of the same surface) to the original data, as long as the sector offset is taken into account. Therefore, an update operation that synchronously modifies the original data and all its replicas can be completed with a very small amount of mechanical delays. This method of replication allows replicas to be placed much closer to their original data while data are kept as fault-tolerant as the previous method.

The analysis of the bad sectors' physical location found on the IBM 75GXP disk shows evidence that suggests the existence of better placement strategies of replicas than the conventional method, which is to place replicas far away from each other and the original data. By placing replicas on a neighboring track (of the same cylinder or the same surface) of the original data, we can achieve the same level of fault-tolerance as the conventional method but with much less performance overhead in maintaining replicas.

On this IBM disk, all its bad sectors are found on zone #0. This can be caused by either head crash or an overuse of this disk region. Some file systems, e.g., NTFS, try to place more frequently used files on the outer edge (low-address zones) of the disk to achieve higher I/O bandwidth. However, this causes some parts of the disk to be used much more frequently than others, and the excessive wear can cause damage over time. A file system that makes more balanced use of different disk regions, e.g., Ext2, can alleviate

Capacity	36.7 GB
Zones	11
Cylinders	15110
Platters	6
Surfaces	12
RPM	10000
Interface	SCSI

Zones	Cyl start	Cyl end	Sectors/Track
0	0	378	504
1	379	2069	476
2	2070	3416	462
3	3417	6788	420
4	6789	7493	406
5	7494	8863	392
6	8864	10167	378
7	10168	11195	364
8	11196	11999	352
9	12000	13714	336
10	13715	15109	308

Table 6.5: IBM 36LZX: Disk specification.

such problems. In the following sections, we will study and characterize other failed disks.

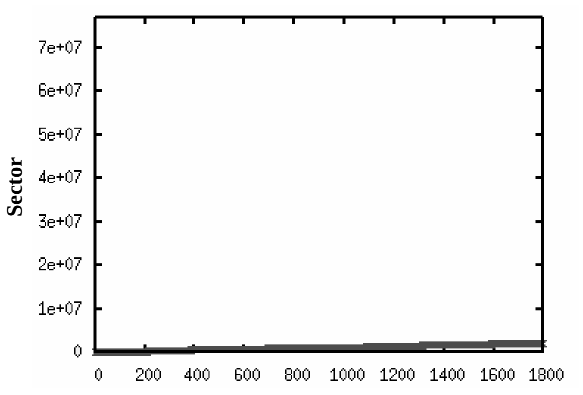


Figure 6.11: IBM 36LZX: A list of bad sectors.

6.3.3 IBM Ultrastar 36LZX

In this section, we will analyze another failed IBM disk—an Ultrastar 36LZX. Its specification is shown in Table 6.5. We found 3 times more number of bad sectors on this disk than the first IBM disk (1799 in total). Similar to the first IBM disk, all bad sectors are found on zone #0 (shown in Figure 6.11, which, as we have suggested previously, may have been caused by an overuse of that region. Additionally, bad sectors had a high concentration on only one of the surfaces (shown in Figure 6.12, and the rest was found on a neighboring surface, which suggests a strong surface locality of defective sectors.

On both IBM disks, the most badly damaged recording surface is the inner surface of the outer-most platter. It is not clear whether or not this is a failure characteristic of only

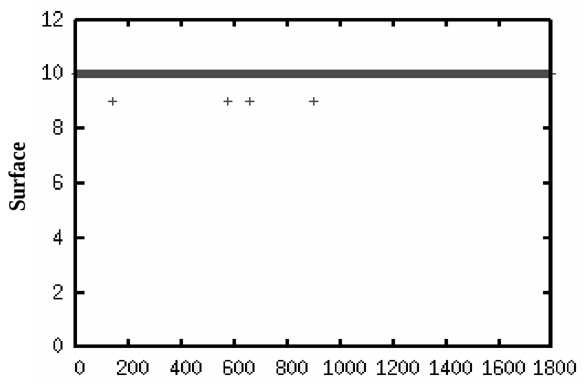


Figure 6.12: IBM 36LZX: Mapping of bad sectors to recording surfaces using the actual physical geometry of the disk.

IBM disks or disks in general as our sample size is not big enough to make such a general statement. Although it does suggest that by exploiting the surface locality of defective sectors when placing replicas, data might be more safely stored on IBM disks.

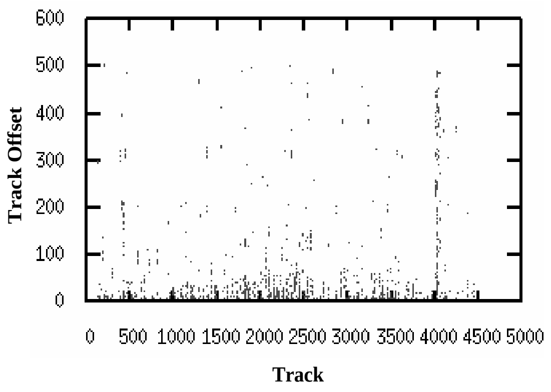


Figure 6.13: IBM 36LZX: Sector offsets of defective sectors.

Despite many similarities in failure characteristics between the two disks, there are also some differences. Unlike the first IBM disk, where we found regular patterns in sector offsets of defective sectors, we did not find similar patterns on the second disk. Sector offsets of the bad sectors found on the second disk are shown in Figure 6.13. This figure shows most defective sectors have a small sector offset within their track. This is more clearly shown in Figure 6.14. However, it is not clear what might have caused it. To guard against such failures, file systems should refrain from placing meta-data and important user data towards the beginning of tracks.

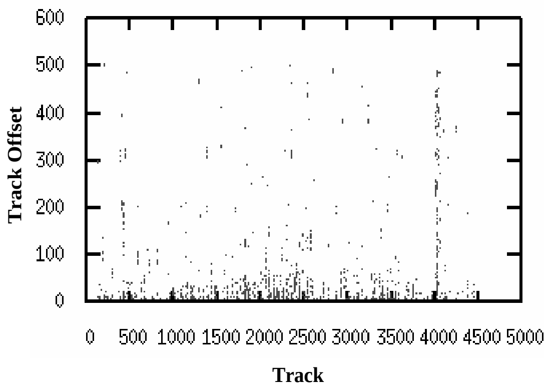


Figure 6.14: IBM 36LZX: A histogram of sector offsets—smaller ones dominate.

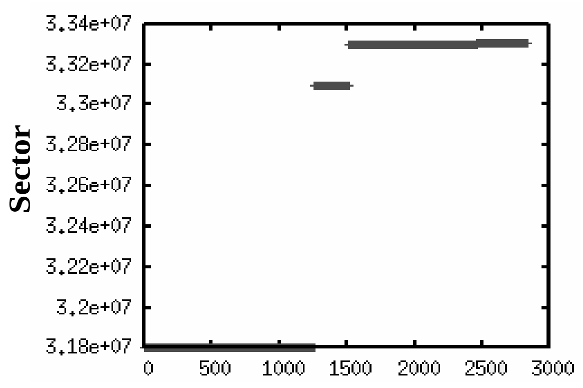
6.3.4 Western Digital Caviar 205BA

Caviar 205BA is the first of two Western Digital (WD) disks we will study in this subsection. Its specification is shown in Table 6.6. The significant difference between WD and IBM disks from our perspective is that WD disks have more zones than IBM disks.

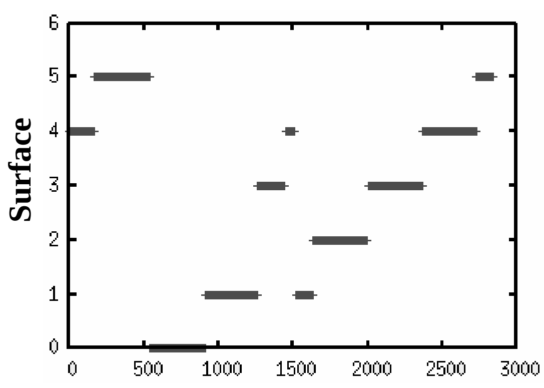
Capacity	20.5 GB
Zones	20
Platters	3
Surfaces	6
RPM	7200
Interface	IDE

Table 6.6: WD 205BA: Disk specification.

We found 2835 bad sectors on this disk. These sectors are plotted in Figure 6.15(a). Among them, 2832 are consecutive. Long streams of consecutive defective sectors are commonly found on failed WD disks. As a result, all platters and surfaces are likely to have some damaged regions (shown in Figure 6.15(b)). Thus, the technique we have proposed for IBM disks, which is to place replicas on the same cylinder as or a neighboring track of the original data, will not be effective for WD disks. Defective sectors on WD disks usually have long runs, but in terms of affected tracks or cylinders, only a few are affected. Therefore, placing replicas a few cylinders away can be a good simple solution.



(a)



(b)

Figure 6.15: WD 205BA: (a) A list of bad sectors. (b) Mapping of the bad sectors to recording surfaces.

6.3.5 Western Digital Caviar WD400BB

This is another WD disk but of a different generation as its platters have a much higher recording density than the previous WD disk. Its specification is given in Table 6.7. This disk also has more zones than the IBM disks.

Capacity	40 GB
Zones	18
Platters	2
Surfaces	4
RPM	7200
Interface	IDE

Table 6.7: WD 400BB: disk specification.

We found a list of 1069 defective sectors on this disk. More than half of them (692) are consecutive, similar to the first WD disk. This is shown in Figure 6.16(a) around sector #7.5e7. Other than these consecutive sectors, there are also 4 smaller defective regions. These are mapped to all 4 surfaces of the disk, as shown in the beginning part of Figure 6.16(b). A closer examination of these defective sectors exposed some regular patterns in their physical location. A sample mapping of these sectors to their track and surface is shown in Table 6.8. It is apparent from this table that the bad sectors are all one track apart. Partitioning these sectors by surface, this is the exactly the same type of damage that we have seen on the IBM 75GXP disk. As opposed to having scratches on only one of the surfaces, the WD 400BB disk seems to have scratches on all surfaces. The sector offset

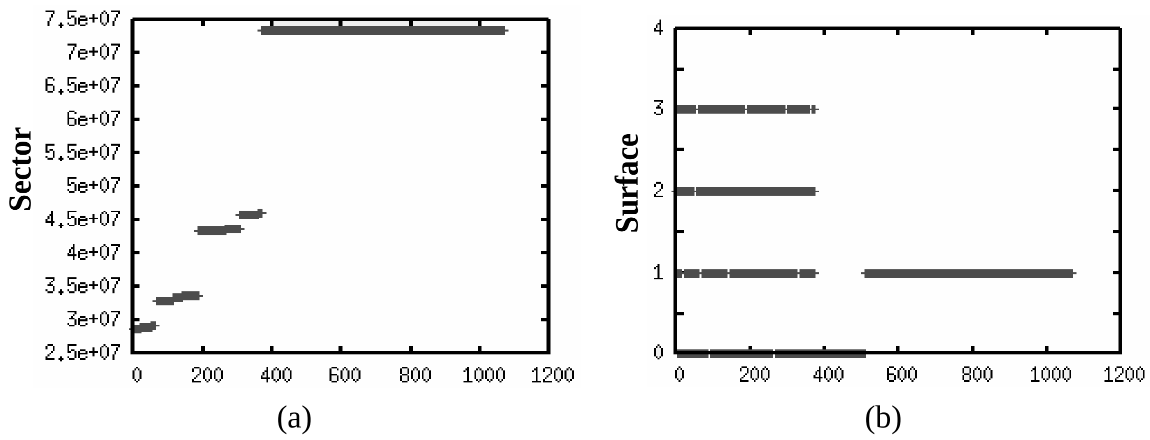


Figure 6.16: WD 400BB: (a) A list of bad sectors. (b) Mapping of the bad sectors to recording surfaces.

differences of these sectors are shown in Figure 6.17. The consistency in the sector offset differences (indication of having a scratch) is apparent from this figure.

Sector	Track	Surface
28888401	31438	2
28889142	31439	3
28889882	31440	0
28890623	31441	1
28891363	31442	2
28892987	31443	3
28893728	31444	0
28894468	31445	1
28895209	31446	2
28895949	31447	3
28896690	31448	0
28898314	31449	1
28899054	31450	2
28899795	31451	3
28900535	31452	0
28901276	31453	1
28902016	31454	2
28902757	31455	3
28904381	31456	0
28944372	31502	2
28945996	31503	3
28946736	31504	0

Table 6.8: WD 400BB: A sample mapping of bad sectors to their track and surface.

6.3.6 Maxtor 4K020H1

We have two failed Maxtor 4K020H1 disks with exactly the same (published) physical geometry, and both will be covered in this section. The specification of these disks is shown

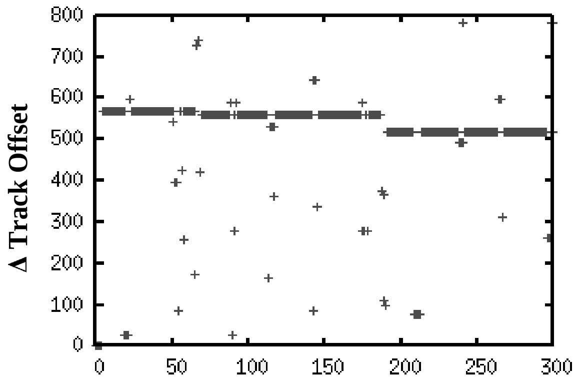


Figure 6.17: WD 400BB: Sector offset differences of bad sectors.

in Table 6.9. These disks are simpler and slower than the previously discussed ones, and they have only one platter with one recording surface. Their rotational speed is only 5400 RPM as opposed to 7200 or 10000 RPM.

Capacity	20 GB
Zones	15
Platters	1
Surfaces	1
RPM	5400
Interface	IDE

Table 6.9: Maxtor 4K020H1: Disk specification.

The list of bad sectors found on the two disks are shown in Figure 6.18. It is common on the two disks that most of the defective sectors are very close to one another. On one of the disks, 63 out of a total of 84 defective sectors are co-located on the same track with at least one other defective sector. On the other disk, 93 out of 214 defective sectors are co-located on the same track with at least one other defective sector. All the defective sectors are localized within 3 or 4 zones on these disks. These observed failure characteristics are fairly similar to the failed Western Digital disks.

6.3.7 Quantum Fireball LCT10

Fireball LCT10 is one of the two Quantum disks we will study in this subsection. Its specification is given in Table 6.10. We plotted all defective sectors in Figure 6.19 The failure characteristic of this Quantum disk is fairly similar to the Maxtor and Western Digital

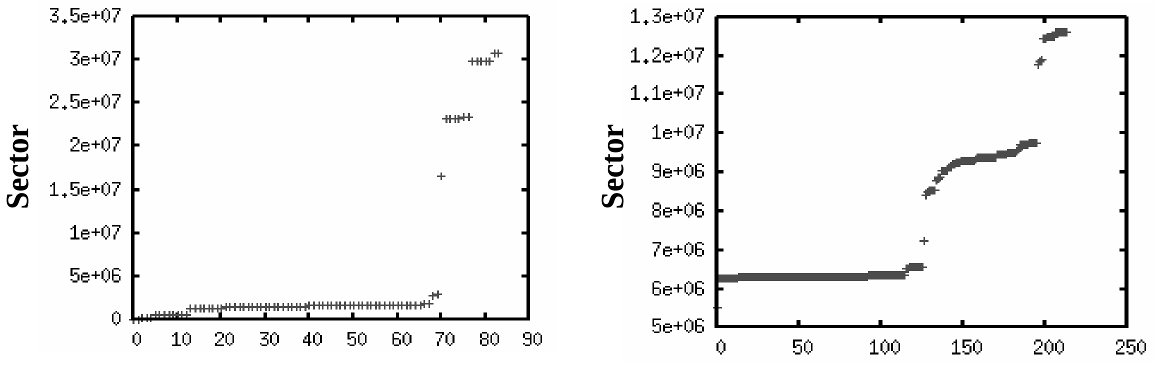


Figure 6.18: Maxtor 4K020H1: A list of bad sectors on each of the Maxtor disks.

disks. It has 31 bad sectors located on only 2 zones, and 27 of them are consecutive. Additionally, only 2 out of 4 surfaces are affected. The same replication techniques proposed for Maxtor and WD disks should be applicable here.

Capacity	36.7 GB
Zones	8
Platters	2
Surfaces	4
RPM	10000
Interface	SCSI

Table 6.10: Quantum LCT10: Disk specification.

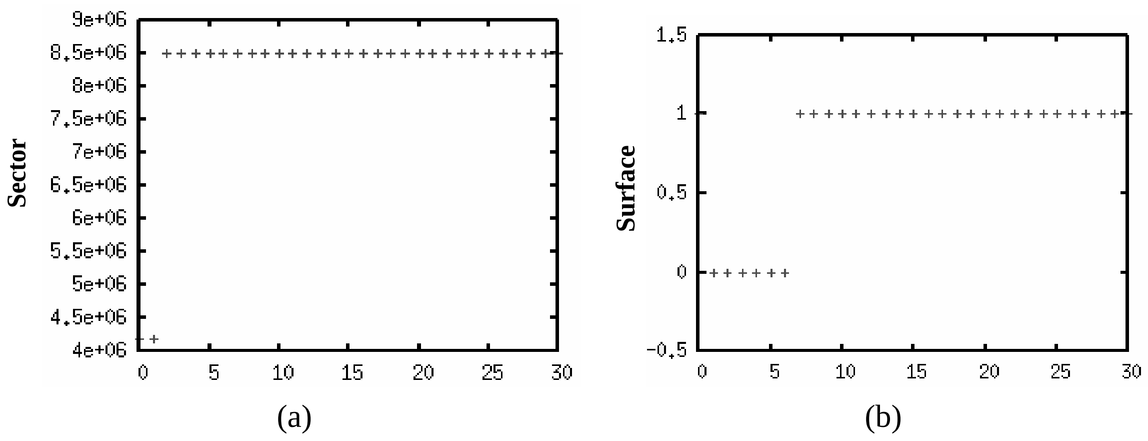


Figure 6.19: Quantum LCT10: (a) A list of bad sectors. (b) Mapping of the bad sectors to recording surfaces.

6.3.8 Quantum Fireball CX10.2A

Fireball CX10.2A is a slightly older disk. Its specification is shown in Table 6.11. 112 defective sectors (shown in Figure 6.20) were found and appeared to be randomly distributed on the disk—on all 3 recording surfaces and on 7 out of 15 zones. Only 21 sectors are co-located on the same track with at least one other defective sector.

Capacity	10.2 GB
Zones	15
Platters	2
Surfaces	3
RPM	5400
Interface	IDE

Table 6.11: Quantum CX10.2A: Disk specification.

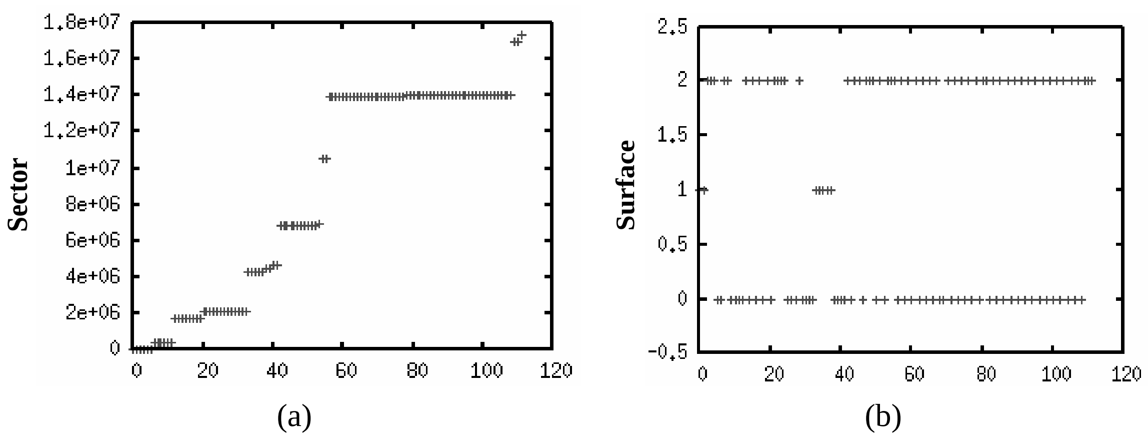
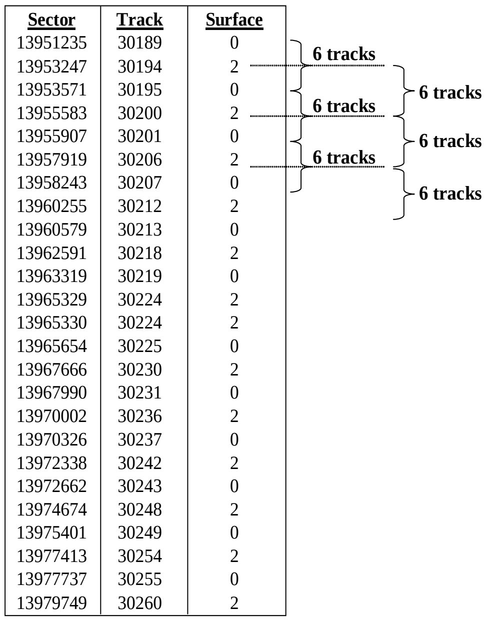


Figure 6.20: Quantum CX10.2A: (a) A list of bad sectors. (b) Mapping of the bad sectors to recording surfaces.

A closer examination of the defective sectors revealed something that we did not see in the other disks. A sample mapping of sectors to their track and recording surface is shown in Table 6.12. In this table, we can see a large percentage of these sectors are mapped to surfaces #0 and #2, and the failed sectors are often separated by exactly 6 tracks (or 2 cylinders). This is not likely to have been caused by mechanical problems, e.g., head crash, as the damaged sectors do not resemble a scratch of any kind. It is more likely to have caused by a problem in the servo system. Simply placing replicas to a few cylinders away from the original data or on a different surface will not be sufficient.

6.4 Conclusions

Replication appears to be an effective method to enhance data fault-tolerance on a single disk and can be practically implemented in existing file systems. We focus our attention on single-disk systems as there are no existing solutions. To use replication to improve fault-tolerance, there are several challenges to overcome. First, we must be able to accurately and quickly detect the physical geometry of the disk and expose this information to the file system for it to be used to better safeguard data. Second, having a good understanding of disk drives' failure characteristics is crucial in making the right replication decision, but there is a lack of such study.


6 tracks
6 tracks
6 tracks
6 tracks
6 tracks
6 tracks

Sector	Track	Surface
13951235	30189	0
13953247	30194	2
13953571	30195	0
13955583	30200	2
13955907	30201	0
13957919	30206	2
13958243	30207	0
13960255	30212	2
13960579	30213	0
13962591	30218	2
13963319	30219	0
13965329	30224	2
13965330	30224	2
13965654	30225	0
13967666	30230	2
13967990	30231	0
13970002	30236	2
13970326	30237	0
13972338	30242	2
13972662	30243	0
13974674	30248	2
13975401	30249	0
13977413	30254	2
13977737	30255	0
13979749	30260	2

Table 6.12: Quantum CX10.2A: a sample mapping of defective sectors to their track and recording surface.

We attacked both of these challenges, first by developing a set of tools to probe for disk physical geometry, and using which, we then analyzed the failure characteristics of 60 failed disks. Our experimental results confirmed that the simple method of placing replicas far away from their original data will work, but not very effectively. Our empirical observation indicates that placing replicas to be very near (on the same cylinder or on a neighboring track) their original data can often provide the same level of fault-tolerance and will only require a minimal amount of resynchronization overheads to keep replicas consistent.

Despite only 9 out of 60 disks were in condition to be analyzed in our study, we observed some interesting phenomena that concurred with some common practices and raised the possibility of potentially better approaches. We found that disks made by different manufacturers sometimes have different failure characteristics. Therefore, the best way to place replicas on a disk will depend on who has manufactured it, and this information is readily available from disks' firmware. Taking advantage of the rapidly increasing disk density, these observations can be exploited by future file systems to make better placement of data to increase data fault-tolerance. This can protect against some data corruption scenarios and give early warnings to users to minimize the probability of permanent data loss.

CHAPTER 7

Related Work

There has been a significant amount of previous work in utilizing unused resources for a variety of purposes, which can be broadly classified into three categories—energy management, performance improvement, and fault-tolerance enhancement. Next, we will discuss each of these areas in more detail.

7.1 Energy

7.1.1 Microprocessor

Recent research has demonstrated that a significant amount of energy can be saved by exploiting the power management capabilities of unused hardware. In the early 90's, Weiser *et al.* [125] first demonstrated the effectiveness of using Dynamic Voltage Scaling (DVS) to reduce power dissipation of microprocessors. DVS allows system software to direct microprocessor to dynamically change its operating frequency and voltage. Almost all mobile microprocessors support this feature. Some earliest work [10, 54, 96, 125] using DVS techniques simply used average processor utilization to direct voltage and frequency scaling. Later work [48, 79] improved these techniques by applying prediction techniques and soft deadlines in their DVS algorithms. This allowed DVS to be applied to general-purpose systems while maintaining good interactive performance. DVS techniques are finally applied to real-time systems [55, 90, 97, 99], where tasks have more strict execution behavior and deadlines.

7.1.2 Disk

Due to having higher rotational speed and heavier platters, new disk drives are consuming an ever-increasing amount of energy. Numerous energy conserving techniques [20, 21, 39, 53, 58, 73, 76, 126] have been proposed by exploiting disk idleness. In these techniques, power is reduced by switching disks to one of the lower energy modes after observing a certain period of disk inactivity. However, due to increasing utilization of disk drives, most idle periods are becoming too small to be exploited in such ways [57, 132]. Gurumurthi *et al.* [57] proposed a Dynamic RPM (DRPM) disk design. In their simulation, this design was shown to be effective in saving energy even when the disk is busy. However, as DRPM disks are not currently pursued by any disk manufacturers, we cannot yet benefit from this new architecture. In this thesis, we lowered energy consumption by reducing seek distance. This is similar to the DRPM design, where power can be reduced even when disk is busy. However, unlike DRPM, our work in FS^2 does not suffer any performance degradation. On the contrary, FS^2 actually improves performance while saving energy.

Techniques to conserve energy in multi-disk arrays have also been proposed. Popular Data Concentration (PDC) [98] and Massive Array of Idle Disks (MAID) [14] have been implemented to use either caching or data migration to increase disk traffic skewness among the disks in the array. This skewness allows some disks to have longer idle periods than others so they can be exploited for energy saving purpose. In [38], traditional RAID systems are studied in detail in the aspects of energy and performance by varying various RAID parameters.

7.1.3 Memory

Among power-management techniques proposed for main memory, there are currently two main approaches—hardware- and software-controlled. Among the software-controlled approach, Delaluz *et al.*[40] proposed a compiler-directed approach, where power-state transition instructions are automatically inserted into compiled code based on offline code profiling. The major drawback of this approach is that compiler only works with one program at a time and has no information about other processes that may be present at runtime. Therefore, it has to be conservative in managing power or else it can trigger large performance and energy overheads when used in a multitasking system. Nevertheless, this technique may be applicable in embedded systems where workloads are more deterministic. Delaluz *et al.*[42] later demonstrated a simple scheduler-based power-management policy. The basic idea is similar to our PAVM approach, but is of much more limited scope. First, we put much more effort into having the underlying physical page allocator allocate pages by collaborating with the VM through a NUMA management layer so that the memory footprint of each process is reduced, whereas they rely on the default behavior of the page allocator and VM. As we have seen in Section 2.4, a substantial amount of power-saving opportunities still remained unexploited even with our rudimentary implementation of PAVM, let alone when pages are randomly allocated by the default page allocator. In [25], it was also noted that the default page-allocation behavior has a detrimental effect on the memory footprint of processes. Second, we have explored more advanced techniques such as aggregation of shared pages and page migration which are necessary for reducing memory footprints when complex sharing between processes in real systems is involved. Page migration techniques [43, 74] have been extensively explored in NUMA systems to reduce I/O latencies. We used a similar migration technique in this thesis but for the purpose of reducing energy consumption. Finally, they determined the active ranks by using page faults and repeated scans of processes' page tables. Although this ensures only the truly active ranks are detected, it is intrusive and involves high operational overheads. In contrast, we take every precaution to avoid performance overheads and hide any avoidable latencies in our design.

Among the hardware-controlled approach, Lebeck *et al.* [25, 45] 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 [46], they used stochastic Petri Nets to explore more complex policies. Delaluz *et al.* took a similar approach in [40], where they studied various flavors of threshold predictors and evaluated their energy implications. The software-directed hardware technique proposed in this thesis is orthogonal to the techniques described above and can be used to improve the prediction accuracy in some of these previously-proposed threshold prediction mechanisms. However, unlike these previous approaches, the techniques proposed in this thesis are specifically designed and optimized for a multitasking environment, as are most of today's systems. In other research contexts, using software and hardware collaboration [9, 22, 77, 110] has also been shown to be beneficial in terms of improving performance and security, and providing new functionalities.

We have also shown that memory traffic reshaping can complement these existing hardware-controlled techniques by creating larger idle periods from smaller ones, thus allowing power management to become more effective in reducing power. Similar techniques have been proposed by [132] in the context of disk drives.

7.1.4 Wireless Device

Due to the rapid increase in the number of mobile devices, e.g., laptops, PDAs, and cellphones, wireless communication is becoming an important area. Wireless devices can consume a significant amount of power [47], and therefore, can severely limit the battery lifetime of mobile systems. For an IEEE 802.11 WLAN device, it can be in one of several states: transmit, receive, idle, and doze. It has been shown that by taking advantage of the doze mode adaptively at appropriate moments [24, 112, 118], battery energy can be conserved. Energy can also be conserved by applying Transmit Power Control [27, 30, 32, 70], which is orthogonal to the techniques that make use of the doze mode.

7.1.5 System

As opposed to saving energy for individual components, there are also some proposals [49, 80, 131] that explored system-level approaches to extend/target the battery lifetime of the system. Flinn *et al.*'s work on Odyssey [49] changes the fidelity of data objects in response to the available resources to meet user-specified goals of battery duration. Zeng *et al.* proposed ECOSystem [131] to provide a model for fair allocation of energy resources to competing applications. In [80], Lu *et al.* monitor processes' requests to hardware components. The operating system can slow down or shut off hardware components to conserve power depending on which process is currently executing.

7.2 Performance

7.2.1 Microprocessor

The most widely-known work that takes advantage of idle processor cycles to do more work is the SETI@Home project [111]. SETI stands for Search for Extra-Terrestrial Intelligence, which, as its name suggests, is used to detect life outside of Earth by analyzing radio signals collected by the Arecibo radio telescope. In this project, a massive amount of data is sent to 5.2 million participants around the world connected by the Internet and processed by their unutilized CPU cycles without interfering with the normal tasks of the participants.

7.2.2 Disk

File System: FFS [34] and its successors [11, 122] improved disk bandwidth and latency by placing related data objects (e.g., data blocks and inodes) near each other on disk. Although it can be effective in initial data placement, it takes no notice of dynamic access patterns. Therefore, its improvement is fairly limited [2, 104]. Based on FFS, Ganger *et al.* [52] optimized performance for small objects by improving their adjacency rather than just locality, to make better use of disk bandwidth. Unfortunately, they also suffer the same set of problems as FFS.

In Log-structured File System (LFS) [103], disk write performance can be significantly improved by delaying and writing in large segments. Tuning LFS by putting active segments on high-bandwidth outer cylinders can further improve write performance [124]. However, due to segment-cleaning overheads and its inability to improve read performance, there is no clear indication that it can outperform file systems in the FFS family.

The closest related works to FS^2 are the Hot File Clustering used in the Hierarchical File System (HFS) [51] and the Smart File System [114]. These file systems monitor file access patterns at runtime and move frequently accessed files to a reserved area on disk. FS^2 has several major advantages over these. First, the use of block granularity allows us to optimize for all files instead of only the smaller ones. It also gives us opportunities to place meta-data and data blocks *adjacent* to one another as opposed to just *close-by*. As indicated in [52], this can significantly improve performance. Second, moving data can potentially break sequentiality. We use data replication so both sequential and random disk I/Os can benefit. Third, using a fixed location on disk to rearrange disk layout will only have benefit under a light load [71]. Using free disk space that may be at arbitrary locations, we show that even under heavy loads, significant benefits can still be attained while keeping replication overhead low. Fourth, reserving a fixed amount of disk space to rearrange disk layout is intrusive, and this can be prevented in our approach by using only the available disk space.

Adaptive Disk Layout: Early adaptive disk layout techniques have been mostly studied either theoretically [128] or via simulation [23, 91, 104, 123]. It has been shown for random disk access, *organ pipe* heuristic, which places the most frequently-accessed data in the middle of the disk, is optimal [128]. However, as real workloads do not generate random disk accesses, the organ pipe placement is not always optimal, and for various reasons, far from practical. To adapt the organ pipe heuristic to realistic workloads, Vongsathorn and Carson [123] proposed *cylinder shuffling*. In their approach, the access frequency of each cylinder is maintained, and based on their frequencies, cylinders are reordered using the organ pipe heuristic at the end of each day. Similarly, Ruemmler and Wilkes [104] suggested shuffling in units of blocks instead of cylinders, and it was shown to be more effective. In all these techniques, it was assumed that the disk geometry is known and can be easily simulated. This might be true at that time, but due to increasingly complex hard drive designs and fiercer competition between disk manufacturers, disk geometry is now mostly hidden. Of course, there are tools [1, 44, 107, 119] that can be used to extract the physical disk geometry experimentally, but such an extraction usually takes a long time. Furthermore, the extracted information can be too tedious to be used at runtime.

Akyurek and Salem [2] were the first who actually implemented block shuffling in a real system. Other than its use of block granularity instead of file granularity, this technique is very similar to HFS and Smart File System, and therefore, suffers the same pitfalls as HFS and Smart File System.

I/O Scheduling: Disk performance can also be improved by means of better I/O scheduling. Seek time can be reduced by using SSTF and SCAN [18], C-SCAN [108], and LOOK [85]. Rotational delay can be reduced by using the approaches as described in [64, 68, 101, 109]. An anticipatory I/O scheduler [67] is used to avoid hasty I/O scheduling decisions by introducing additional wait time. These techniques are orthogonal to the adaptive disk layout techniques.

7.2.3 Graphics Processor

Today's GPUs are highly parallel vector processors that can handle texture mapping and polygon rendering at a blazing speed. In fact, their core frequency has been doubling roughly every 6 months [33], even faster than CPU's rate of improvements. However, GPUs are utilized only by a small number of 3-D applications, so most of the time, they just sit idle in the background. Various interest groups are pushing for GPU manufacturers towards more general-purpose GPU architectures with easier programming models. This will ideally allow GPUs to be used for general-purpose computation, but up to date, only a very small number of specialized scientific computations has been compiled to run on GPUs [26, 120]. This is still a research area with many low-hanging fruits. Furthermore, video memory has become an ample resource in today's video cards due to the recent drop in memory price. Video cards with half of a gigabyte of memory are fairly common, which is comparable to the size of main memory. It is not difficult to imagine system software that can opportunistically take advantage of the idling video memory to augment the size of main memory when running memory-intensive applications.

7.3 Fault-Tolerance

7.3.1 Disk

An adequate level of fault-tolerance against data loss and data corruption is a must in mission-critical environments. Redundant Array of Inexpensive Disks (RAID) [94] is the most widely used storage volume manager that makes tradeoffs between the storage system's performance, fault-tolerance, cost, and energy consumption. Data loss can be deterred by using either mirroring (RAID-0) or parity information (RAID-5), or a combination of both depending on the performance and fault-tolerance requirement of a particular system. However, in a RAID system, when improving one dimension, one or more other dimensions are usually degraded. For example, if we want to upgrade an existing RAID system so that it can tolerate more failures, then we would need to buy more disks just so we can store the same amount of data as before. This increases hardware cost, administration cost, and energy cost. These tradeoffs between various dimensions in a RAID system are studies here [3, 7, 13, 37, 127, 130]. For backup systems, not only it requires additional equipments to hold backup data, but it also gives additional chores to system administrators.

The Elephant Filesystem [31] uses free disk space to automatically store old versions of files to avoid explicit user-initiated version control. In their system, a cleaner is used to free older file versions to reclaim disk space, but it is not allowed to free *Landmark* files. In FS^2, all free space used for duplication can be reclaimed as replicas are used only for enhancing performance and invalidating them will not compromise the correctness of the system.

Additionally, distributed storage systems such as Farsite [28], Pastiche [16], and Samsara [17] exploit free disk capacity in participating clients to distribute data across geographically different locations to implement highly available, reliable, and secure data storage systems.

CHAPTER 8

Conclusions and Future Work

8.1 Conclusions

The aim of this thesis is to show how unused hardware resources can be effectively utilized to the benefit of users while keeping all the additional complexity due to harnessing such resources hidden from them. One main focus is to exploit idle resources for energy savings.

To reduce power dissipated by the main memory, we first implemented a purely software-controlled power-management technique. This technique is implemented both on a real system and a simulated system, and we were able to demonstrate a significant amount of power savings. Besides the energy benefit, this technique is platform independent and can be easily ported to different machine and memory architectures. However, having power-control and monitoring mechanisms implemented in the system software inheritantly limited the effectiveness of this approach, and as a result, many power-saving opportunities cannot be exploited. To improve upon this technique, we then implemented a software-directed hardware power-management technique, which takes advantage of (i) finer-grained power-control mechanisms available at the hardware level and (ii) system information available at the operating system level. In addition to these two techniques, we also demonstrated the usefulness of actively reshaping memory traffic in reducing memory's power dissipation and how this technique can be used to complement some existing hardware power-management techniques.

In the area of magnetic disks, we have shown the usefulness of free disk capacity in FS^2. The use of free disk space allows a flexible way to improve disks' I/O performance and reduce their power dissipation, while being nonintrusive to users. We have also demonstrated the potential usefulness of using replication and unused disk capacity to improve data fault-tolerance in a detailed analysis of disk failure characteristics and strategies to better place data and replicas on disk.

8.2 Future Work

In the area of memory power-management, our main focus is on uniprocessor systems, with an explicit assumption that there is only one running process at any instant of time. PAVM is specifically designed and optimized for such systems. It will be interesting to consider extending PAVM to systems with multiple processing units, e.g., multi-core and multi-processor, of which many of today's systems are equipped. To implement PAVM in such systems, we would need to consider a combination of running processes instead of a single process, and thus, task scheduling will play an important role in power management.

In FS^2, there are still many interesting areas to explore. For example, replication can be done either at the file system level or at the block device level. If it is done at the file system level, implementation is much easier, but it will be very file-system-specific and need to be re-implemented for each file system we want to extend and evaluate. If it is done at the block device level, we can generally apply it to any file system. The problem with this approach is that the block device driver does not know whether a disk block is free or used, as this information is available only at the file system level. To have replication done at the block device level, it would require the file system to export some information to the the block device driver.

Several other design aspects of FS2 are also interesting to explore in the future. First, we have only implemented a working 1-replica system, but there are scenarios where an n-replica system, where $n > 1$, might perform better. Second, we have only explored replication techniques, but migration sometimes can be more appropriate. Further investigation is needed to determine under which scenario is migration or replication the better method to reorganize data layout. Third, extending the techniques we have developed for single-disk systems to multi-disk systems can potentially improve existing multi-disk solutions, e.g., RAID, in terms of performance, fault-tolerance, and energy-efficiency.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] M. Aboutabl, A. Agrawala, and J. Decotignie. Temporally determinate disk access: An experimental approach. In *Technical Report, CS-TR-3752*, 1997.

[2] Sedat Akyurek and Kenneth Salem. Adaptive block rearrangement. *Computer Systems*, 13(2):89–121, 1995.

[3] E. Anderson, R. Swaminathan, A. Veitch, G. Alverez, and J. Wilkes. Selecting RAID levels for disk arrays. In *Conference on File and Storage Technologies (FAST)*, pages 189–201, 2002.

[4] APC—American Power Conversion. Determining total cost of onwership for data centers and network room infrastructure, December 2003.

[5] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of the 1998 ACM Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 151–160, 1998.

[6] Fred Beck. Energy smart data centers: Applying energy efficient design and technology to the digital information sector. In *Renewable Energy Policy Project*, November 2001.

[7] Dina Bitton and Jim Gray. Disk shadowing. In *Proceedings of the International Conference on Very Large Data Bases*, pages 331–338, 1998.

[8] Pat Bohrer, Mootaz Elnozahy, Tom Keller, Mike Kistler, Charles Lefurgy, Chandler McDowell, and Ram Rajamony. The case for power management in web servers. In *Power Aware Computing*, January 2002.

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

[10] T. D. Burd and R. W. Brodersen. Energy efficient 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.

[11] R. Card, T. Ts'o, and S. Tweedle. Design and implementation of the second extended filesystem. In *First Dutch International Symposium on Linux*, 1994.

[12] Data Centre. http://www.max-t.com/downloads/whitepapers/sl edgehammerpower-heat20411.pdf.

[13] Peter M. Chen and Edward K. Lee. Stripping in a RAID 5 disk array. In *Proceedings of the ACM SIGMETRICS*, 1995.

[14] D. Colarelli and D. Grunwald. Massive arrays of idle disks for storage archives. In *Proceedings of the 15th Supercomputing (SC)*, 2002.

[15] Transactional Processing Performance Council. http://www.tpc.org/tpcw.

[16] Landon Cox, Christopher D. Murray, and Brian Noble. Pastiche: Making backup cheap and easy. In *The 5th Proceedings of Operating Systems Design and Implmentation*, 2002.

[17] Landon Cox and Brian Noble. Samsra: Honor among thieves in peer-to-peer storage. In *The 19th Symposium of Operating Systems Principles*, 2003.

[18] P. J. Denning. Effects of scheduling on file memory operations. In *AFIPS Spring Joint Computer Conference*, pages 9–21, 1967.

[19] John Douceur and William Bolosky. A large-scale study of file-system contents. In *ACM SIGMETRICS Performance Review*, pages 59–70, 1999.

[20] 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.

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

[22] 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.

[23] Robert English and Stephnov Alexander. Loge: A self-organizing disk controller. In *Proceedings of the Winter 1992 USENIX Conference*, 1992.

[24] E.S.Jung and N.H.Vaidya. An energy efficient mac protocol for wireless lans. In *IEEE INFOCOM*, pages 1756–1764, 2002.

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

[26] Aaron Lefohn *et al.* Glift: Generic, efficient, random-access gpu data structures. In *ACM Transaction on Graphics*, 2006.

[27] A.E.Gamal *et al.* Energy-efficient scheduling of packet transmission over wireless networks. In *IEEE INFOCOM*, pages 1773–1782, 2002.

[28] Atul Adya *et al.* Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *The 5th Proceedings of Operating Systems Design and Implementation*, 2002.

[29] B. L. Worthington *et al.* Online extraction of SCSI disk drive parameters. In *Proceedings of the ACM SIGMETRICS*, pages 146–156, 1995.

[30] Daji Qiao *et al.* Miser: An optimal low-energy transmission strategy for IEEE 802.11a/h. In *ACM MobiCom*, 2003.

[31] Douglas Santry *et al.* Deciding when to forget in the elephant file system. In *ACM Symposium on Operating Systems Principles*, pages 110–123, 1999.

[32] J. Gomez *et al.* Conserving transmission power in wireless ad hoc networks. In *IEEE ICNP*, pages 24–34, 2001.

[33] John Owens *et al.* A survey of general-purpose computation on graphics hardware. In *Eurographics*, 2005.

[34] M. K. McKusick *et al.* A fast file system for unix. *ACM Transactions on Computing Systems (TOCS)*, 2(3), 1984.

[35] Muthian Sivathanu *et al.* Life or death at block level. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 379–394, 2004.

[36] P. Bohrer *et al.* Mambo — a full system simulator for the powerpc archtecture. *ACM SIGMETRICS Performance Evaluation Review*, 31(4), 2004.

[37] P. Chen *et al.* Raid: High-performance, reliable secondary storage. *ACM Computing Survyes*, 26(2):145–185, 1994.

[38] S. Gurumurthi *et al.* Interplay of energy and performance for disk arrays running transaction processing wokloads. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2003.

[39] T. Heath *et al.* Application transformations for energy and performance-aware device management. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, pages 121–130, 2002.

[40] 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.

[41] V. Delaluz *et al.* Hardware and software techniques for controlling dram power modes. *IEEE Transactions on Computers*, 50(11):1154–1173, 2001.

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

[43] W. J. Bolosky *et al.* Numa policies and their relation to memory architecture. In *Proceedings of the Fourth International Conference on Architectural Support for Programming languages and Operating Systems*, pages 212–221, 1991.

[44] Zoran Dimitrijevic *et al.* Diskbench: User-level disk feature extraction tool. In *Technical Report, UCSB*, 2004.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

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

[50] Linux Filesystem Performance Comparison for OLTP. http://oracle.com/technology/tech/linux/pdf/linux-fs-performance-comparison.pdf.

[51] HFS Plus Volume Format. http://developer.apple.com/technotes/tn/tn1150.html.

[52] Greg Ganger and Frans Kaashoek. Embedded inodes and explicit groups: Exploiting disk bandwidth for small files. In *USENIX Annual Technical Conference*, pages 1–17, 1997.

[53] R. A. Golding, P. Bosch, C. Staelin, T. Sullivan, and J. Wilkes. Idleness is not sloth. In *USENIX Winter*, pages 201–212, 1995.

[54] 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.

[55] 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.

[56] PC Guide. http://www.pcguide.com/ref/power/index.htm.

[57] S. Gurumurthi, A. Sivasubramaniam, K. Kandemir, and H. Franke. DRPM: Dynamic speed control for power management in server class disks. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2003.

[58] D. Helmbold, D. Long, T. Sconyers, and B. Sherrod. Adaptive disk spin-down for mobile computers. *ACM/Balazer Mobile Networks and Applications (MONET) Journal*, 5(4):285–297, 2000.

[59] Mike Hopkins. The onsite energy generation options. In *The Data Center Journal*, Feburary 2004.

[60] H. Huang, W. Hung, and K. G. Shin. Fs2: Dynamic disk replication in free disk space for improving disk performance and energy consumption. In *Symposium of Operating Systems Principles*, pages 263–276, 2005.

[61] H. Huang, C. Lefurgy, T. Keller, and K. G. Shin. Memory traffic reshaping for energy-efficient memory. In *International Symposium on Low Power Electronics and Design*, pages 393–398, 2005.

[62] 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.

[63] H. Huang, K. G. Shin, C. Lefurgy, K. Rajamani, T. Keller, E. V. Hensbergen, and F. Rawson. Cooperative software-hardware power management for main memory. In *Power-Aware Computing Systems*, pages 61–77, 2004.

[64] L. Huang and T. Chiueh. Implementation of a rotation latency sensitive disk scheduler. In *Technical Report, ECSL-TR81*, 2000.

[65] IBM DB2. http://www-306.ibm.com/software/data/db2.

[66] Intel. http://www.intel.com/design/mobile/perfbref/250725.htm.

[67] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *18th ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[68] D. Jacobson and J. Wilkes. Disk scheduling algorithms based on rotational position. In *HP Technical Report, HPL-CSP-91-7*, 1991.

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

[70] J.P.Ebert and A. Wolisz. Combined tuning of RF power and medium access control for wlans. In *Mobile Networks and Applications*, pages 417–426, 2001.

[71] Richard King. Disk arm movement in anticipation of future requests. In *ACM Transaction on Computer Systems*, pages 214–229, 1990.

[72] Donald E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, 1968.

[73] 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.

[74] R. P. Larowe and C. S. Ellis. Experimental comparison of memory management policles for numa multiprocessors. In *ACM Transactions on Computer Systems*, 1991.

[75] 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.

[76] 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.

[77] 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.

[78] Sai-Lai Lo. Ivy: a study on replicating data for performance improvement. In *HP Technical Report, HPL-CSP-90-48*, 1990.

[79] 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.

[80] 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.

[81] Y. H. Lu, L. Benini, and G. De Michelli. Power-aware operating systems for interactive systems. *IEEE transactions on very large scale integration (VLSI) systems*, 10(2), 2002.

[82] C. Lumb, J. Schindler, and G. Ganger. Freeblock scheduling outside of disk firmware. In *Conference on File and Storage Technologies (FAST)*, pages 275–288, 2002.

[83] C. Lumb, J. Schindler, G. Ganger, D. Nagle, and E. Riedel. Towards higher disk head utilization: Extracting free bandwidth from busy disk drives. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2000.

[84] Aqeel Mahesri and Vibhore Vardhan. Power consumption breakdown on a modern laptop. In *Power-Aware Computer System*, December 2004.

[85] A. G. Merten. Some quantitative techniques for file organization. In *Ph.D. Thesis*, 1970.

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

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

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

[89] Bob Moore. http://www.energyusernews.com/cda/articleinformation/features/bnp_features_item/0,2584,82916,00.html.

[90] 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.

[91] David Musser. Block shuffling in loge. In *HP Technical Report CSP-91-18*, 1991.

[92] MySQL. http://www.mysql.com.

[93] Douglas Orr, Jay Lepreau, J. Bonn, and R. Mecklenburg. Fast and flexible shared libraries. In *USENIX Summer*, pages 237–252, 1993.

[94] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of ACM SIGMOD*, pages 109–116, 1988.

[95] David Patterson. Latency lags bandwidth. In *Communications of the ACM*, 2004.

[96] 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.

[97] 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.

[98] E. Pinheiro and R. Bianchini. Energy conservation techniques for disk array-based servers. In *Technical Report, DCS-TR-525*, 2003.

[99] 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.

[100] Power, Heat, and Sledgehammer, 2004.

[101] Lars Reuther and Martin Pohlack. Rotational-position-aware real-time disk scheduling using a dynamic active subset (DAS). In *IEEE Real-Time Systems Symposium*, 2003.

[102] Rik V. Riel. http://www.surreal.com.

[103] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *ACM Transactions on Computer Systems*, pages 26–52, 1992.

[104] C. Ruemmler and J. Wilkes. Disk shuffling. In *HP Technical Report, HPL-CSP-91-30*, 1991.

[105] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, 1994.

[106] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen Keckler, and Charles Moore. Exploiting ilp, tlp, and dlp with the polymorphous trips architecture. In *ISCA*, 2003.

[107] J. Schindler and G. Ganger. Automated disk drive characterization. In *Technical Report CMU-CS-99-176*, 1999.

[108] P. H. Seaman, R. A. Lind, and T. L. Wilson. An analysis of auxiliary-storage activity. *IBM System Journal*, 5(3):158–170, 1966.

[109] M. Seltzer, P. Chen, and J. Ousterhout. Disk scheduling revisited. In *USENIX Winter*, pages 313–324, 1990.

[110] 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.

[111] S. Shostak. Sharing the universe: Perspectives of extraterrestrial life. In *Berkley Hills Books*, 1998.

[112] 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.

[113] SQL Server. http://www.microsoft.com/sql/default.mspx.

[114] C. Staelin and H. Garcia-Molina. Smart filesystems. In *USENIX Winter*, pages 45–52, 1991.

[115] Standard Performance Evaluation Corporation. http://www.specbench.org/jbb2000/.

[116] Standard Performance Evaluation Corporation. http://www.specbench.org/osg/cpu2000/.

[117] Internet World Statistics. www.internetworldstats.com/stats.htm.

[118] 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.

[119] N. Talagala, R. Arpaci-Dusseau, and D. Patterson. Microbenchmark-based extraction of local and global disk characteristics. In *Technical Report CSD-99-1063*, 1999.

[120] Chris Thompson, Sahngyun Hahn, and Mark Oskin. Using modern graphics architectures for general-purpose computing: A frameword and analysis. In *The 35th International Symposium on Microarchitecture*, 2002.

[121] D. A. Thompson and J. S. Best. The future of magnetic data storage technology. *IBM Journal of Research Development*, 44(3), 2000.

[122] Stephen Tweedie. Journaling the linux ext2fs filesystem, 1998.

[123] P. Vongsathorn and S. D. Carson. A system for adaptive disk rearrangement. *Software Practice Experience*, 20(3):225–242, 1990.

[124] J. Wang and Y. Hu. Profs – performance-oriented data reorganization for log-structured file system on multi-zone disks. In *9th International Symposium on Modeling, Analysis and Simulation on Computer and Telecommunication Systems*, pages 285–293, 2001.

[125] 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.

[126] A. Weissel, B. Beutel, and F. Bellosa. Cooperative i/o — a novel i/o semantics for energy-aware applications. In *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[127] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The hp autoraid hierarchical storage system. In *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pages 90–106, 2001.

[128] C. K. Wong. Algorithmic studies in mass storage systems. In *Computer Sciences Press*, 1983.

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

[130] Xiang Yu, Benjamin Gum, Yuqun Chen, Randolph Y. Wang, Kai Li, Arvind Krishnamurthy, and Thomas E. Anderson. Trading capacity for performance in a disk array. In *Proceedings of Symposium on Operating Systems Design and Implementation*, pages 243–258, 2000.

[131] 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.

[132] Qingbo Zhu, Francis M. David, Christo F. Devaraj, Zhenmin Li, Yuanyuan Zhou, and Pei Cao. Reducing energy consumption of disk storage using power-aware cache management. In *The Tenth International Symposium on High Performance Computer Architecture (HPCA-10)*, February 2004.