NORTHWESTERN UNIVERSITY

Transparent Memory Hierarchy Compression and Migration

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Electrical and Computer Engineering

By

Lei Yang

EVANSTON, ILLINOIS

December 2008

ABSTRACT

Transparent Memory Hierarchy Compression and Migration

Lei Yang

Embedded systems are ubiquitous. Although many aspects of embedded system design and synthesis have received significant research attention, comparatively less attention has been given to new ideas in memory hierarchy design. This dissertation presents several new operating system and architecture techniques that use elements of the virtual and physical memory system to improve the functionality, power consumption, and performance of embedded systems such as multimedia devices and wireless sensor network nodes. In particular, the techniques presented in this dissertation explore the use of software and hardware compression of physical and virtual memory to improve performance and functionality of uniprocessor and multiprocessor systems.

We describe an operating system controlled, on-line memory compression framework for embedded systems that support virtual memory but have tight physical RAM constraints. We

present a new software-based memory compression algorithm and a method of adaptively managing the uncompressed and compressed memory regions during application execution. We also describe the use of software on-line memory compression in embedded systems that do not support virtual memory, e.g., wireless sensor network nodes. Specifically, we focus on a new compression algorithm that is developed for compressing sensor data.

We present a hardware-based, adaptive cache and memory link compression framework to improve the performance of applications with large working data sets. We propose efficient organization scheme to manage compressed cache lines, and present detailed hardware design and synthesis results of the area and performance overheads of the required compression and decompression hardware. In addition, we present a cooperative cache compression and migration technique to improve in chip multiprocessor throughput without increasing area, or to reduce area without degrading throughput. We propose an adaptive control policy integrating the two techniques and permitting run-time adaptation to workload.

Acknowledgements

It is my pleasure to express my gratitude to these people who have contributed, in many different ways, to make my success a part of their own.

First, I wish to express my deepest gratitude to my advisor, Professor Robert Dick, for his guidance, insights and attention to details throughout my Ph.D. study. We closely collaborated on all the work presented in the body of this dissertation. I am very fortunate to have Professor Dick as my advisor, for his insights into the essentials of problems and his stimulating suggestions, for his encouragement and patience when I am stuck during my research, and for his invaluable support and advices both at the professional level and at the personal level.

I am sincerely grateful to Doctor Haris Lekatsas and Doctor Srimat Chakradhar, for their helpful advice and discussions on the operating system controlled on-line compression work described in Chapter 3. Much of this work was done during my term as an employee of NEC Laboratories America.

I am sincerely grateful to Professor Li Shang, from the University of Colorado at Boulder, for his advice on the hardware-based cache and memory compression work described in Chapter 6, and the chip multiprocessor cooperative cache compression and migration work described

in Chapter 7. I especially appreciate the time and efforts he spent on the numerous meetings and discussions we have.

I would also like to extend my gratitude and appreciation to my exam committee, Professor Gokhan Memik, Professor Peter Dinda, and Professor Russ Joseph, for generously spending their time and efforts on my research and for kindly being the first readers of this dissertation. Professor Gokhan Memik and Professor Peter Dinda also gave me numerous helpful suggestions on the human and application driven processor frequency scaling work described in Chapter A.

I was glad to have the opportunity to collaborate and discuss research with Lan Bai, Xi Chen, Hui Ding, Jia Wang, David Bild, Jack Cosgrove, Stephen Tarzia, Lide Zhang, Yan Gao, and Alex Shye. I would also like to thank my friends at Northwestern University, who gave me help and support at various times, and made my graduate life easier and joyful.

I am deeply grateful to the Graduate School of Northwestern University, for awarding me the prestigious Presidential Fellowship. I was fortunate enough to meet stellar scholars from across the university, to have interdisciplinary interactions with them, and to learn about cutting-edge research in other fields.

My special and sincere appreciation goes to my wonderful husband, Hui Ding, for being patient, supportive and helping at ever stage of my Ph.D. study. It is his love, understanding, encouragement, and support that made me through the hardest times of my graduate life.

Finally, my forever gratitude goes to my parents. Without their unconditional love, support and encouragement, I would never have achieved what I have in my life.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Embedded systems are ubiquitous. Although many aspects of embedded system design and synthesis have received significant research attention, comparatively less attention has been given to new ideas in memory hierarchy design for embedded systems. While designing and exploiting memory hierarchy is essential to modern computer architectures, operating systems, and applications, most past work has focused on only a few uses of memory hierarchy, e.g., improving performance and, more recently, power consumption via conventional caches; improving the convenience for application developers via swapping; and improving application stability by using virtual memory to create isolated data storage regions.

In this dissertation, we explain how novel memory hierarchies can help solve several pressing problems in embedded system design, and describe several new virtual and physical memory hierarchy design techniques that have the potential to dramatically improve the functionality and performance of modern embedded systems, e.g., portable consumer electronics and wireless sensor network nodes. In particular, the techniques presented in this dissertation explore

the use of software and hardware compression of physical and virtual memory to improve performance of uniprocessor and multiprocessor systems. We demonstrate that compression of one level in the memory hierarchy can expand the capacity of that level, thereby reducing accesses to the next level of hierarchy and improving performance.

In Chapter 3, we describe a novel on-line memory compression framework designed for embedded systems. This framework, named CRAMES, is a software-controlled OS module that provides a compressed virtual swap device and manages a new virtual layer in the memory hierarchy. This technology increases embedded system functionality by doubling the amount of RAM available on portable embedded systems, without requiring any changes to hardware or applications, and with small performance and power consumption penalties.

In Chapter 4, we describe our techniques to improve the performance of CRAMES. We first describe a new, fast, high-quality memory compression algorithm for use in high-performance on-line memory compression. This algorithm, named pattern-based partial match (PBPM), exploits frequent patterns that occur within each word of memory and takes advantage of the similarities among words by keeping a small two-way, hashed, set-associative dictionary that is managed with a least-recently used (LRU) replacement policy. In comparison with algorithms that are commonly used in on-line memory compression, our new algorithm has a competitive compression ratio but is twice as fast with significantly lower memory overhead. We then discuss several practical issues in compressed memory management, and describe the dynamic and adaptive memory management techniques we developed for on-line memory compression.

In Chapter 5, we describe a new memory expansion technique, named MEMMU, for use in wireless sensor networks. MEMMU provides application developers with access to more usable RAM and requires no or minor changes to application code and no changes to hardware. This work was done in collaboration with other researchers. In particular, Lan Bai was the leader on developing compile-time transformation and run-time library support to automatically manage on-line migration of data between compressed and uncompressed memory regions. The author of this dissertation focused on developing the Delta compression algorithm, an efficient software-based compression algorithm that is well suited for compressing sensor data. Therefore, the focus of this chapter is on the Delta compression algorithm, and the compile-time and run-time techniques are only briefly summarized to give readers a context.

In Chapter 6, we present a hardware-based, unified memory hierarchy compression technique to improve the performance of applications with large working data sets. Data in on-chip caches are adaptively compressed to benefit applications that require large cache without adding unnecessary overhead to applications that do not. In addition, data that are transferred to off-chip main memory are compressed to reduce processor–memory communication latency and bandwidth requirements. We propose a pair-matching scheme to store compressed cache lines, which is demonstrably more efficient than previous segmentation-based techniques. We also present a highly-efficient hardware compression algorithm that has been designed for fast cache compression. This work was done in collaboration with other researchers. In particular, Xi Chen was the leader on implementing, synthesizing, and evaluating the compression and decompression hardware.

In Chapter 7, we present a cooperative on-chip cache compression and migration technique to improve chip multiprocessor throughput without increasing area, or to reduce area without degrading throughput. We propose an adaptive control policy integrating the two techniques and permitting run-time adaptation to workload. The feasibility, area, and performance overheads of the required control, compression, and decompression hardware were evaluated via detailed design and synthesis. The performance impact was explicitly modelled and the area overhead was found to be small compared to processor and cache area.

In Chapter 8, we summarize the contributions of the work presented in the main body of this dissertation, and conclude. In Chapter A, we present a dynamic CPU power reduction technique that adapts CPU voltage and frequency to the performance requirements of individual users and applications.

Chapter 2

The Processor–Memory Bottleneck

Computer designers are now faced with an increasing *processor–memory performance gap* [**1**, **2**]: the rate of improvement in microprocessor performance exceeds the rate of improvement in memory performance. Moreover, the disparity between the two is continuously growing. Therefore, the processor–memory performance gap is now a primary bottleneck to improve computer system performance. The ongoing move to chip-level multiprocessors (CMPs) is further increasing the problem; more processors require more accesses to memory, but the performance of the processor–memory bus is not keeping pace.

The processor–memory performance gap is one of the fundamental problems that motivate the techniques presented in the following chapters of this dissertation, especially the hardware-based cache and memory compression techniques. In this section, we first give an overview of the memory hierarchy of computer systems. We then describe the cause of the processor–memory gap, the reason why it has become a primary bottleneck to improve computer systems performance, and the current research efforts on solving this problem.

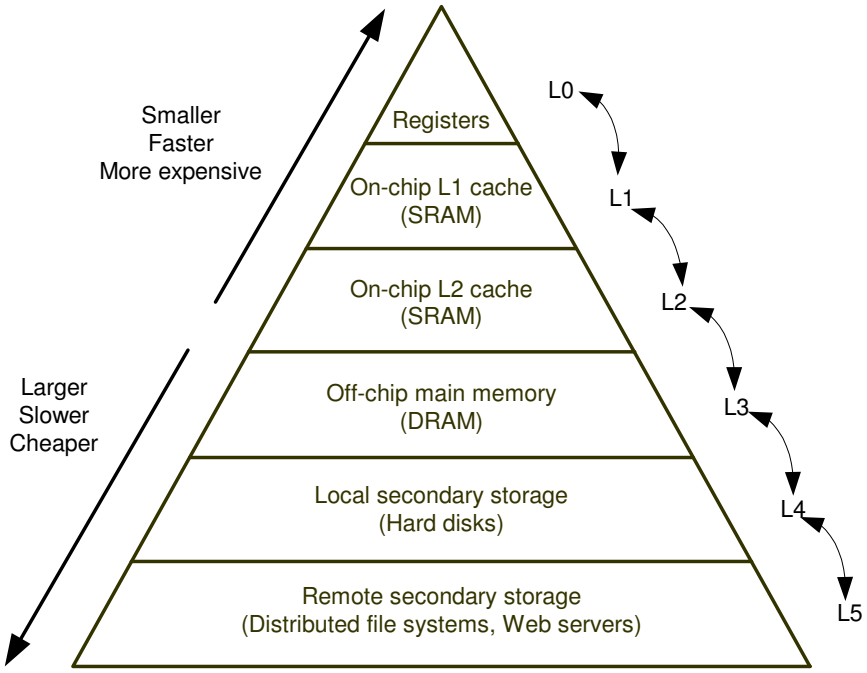

FIGURE 2.1. Memory hierarchy.

2.1. Memory Hierarchy

The memory system is the repository for all code and data used by and produced by the processor. The phenomenal increase in microprocessor performance places significant demands on the memory system. Ideally, a perfect memory system is a large, flat storage system that immediately supplies data that the processor requests. However, this ideal memory is not practically implementable, as the three factors of memory (capacity, speed, and cost) are in direct opposition. An economical solution is to hierarchically organize the memory system into several levels, each smaller, faster, and more expensive per byte than the next. The goal is to provide a

memory system with cost almost as low as the cheapest level and performance almost as high as the fastest level.

The concept of memory hierarchy takes advantage of the principle of locality. Briefly, this principle states that:

- *Temporal locality*: Recently accessed data will be accessed again soon.
- *Spatial locality*: Data adjacent to recently accessed data will be accessed soon.

Another principle is that smaller is faster, i.e., smaller pieces of hardware will generally be faster than larger pieces. This simple principle is particularly applicable to memories built from similar technologies for two reasons. First, larger memories have more signal delay and require more levels to decode addresses to fetch the required data. Second, in most technologies we can obtain smaller memories that are faster than larger memories. This is primarily because the designer can use more power per memory cell in a smaller design. The fastest memories are generally available in smaller number of bits per chips at any point in time, and they cost substantially more per byte. For these reasons, the memory hierarchies of modern general purpose computers generally contain the following, as shown in Figure 2.1.

- *Registers*: These are the small and fast storage buffers within the processor. The compiler is responsible for managing their use, i.e., deciding which values should be kept in the available registers during different execution phases of the program.
- *Caches*: These are small and fast memories generally located on the same chip as the processor, which holds the most recently accessed code or data. Caches are generally made from static random access memory (SRAM). They typically exploit the principle

of spatial locality of reference by prefetching a fixed amount of data that are located close to the referenced value. They can vary widely in size and organization, and there may be more than one level of cache in the hierarchy. The widening gap between the processor and memory has lead to the use of multi-level caches. Modern computers usually have a two-level cache hierarchy, where the level closer to the processor is designated as level one (L1) cache and the next level is designated as level two (L2) cache. Although adding levels in the hierarchy is straightforward, it increases latency and complicates design.

- *Main memory*: This is the next level down in the hierarchy, which is generally located on a different chip from the processor. Main memory interfaces with both caches and hard disk. It is generally made from dynamic random access memory (DRAM). Compared to caches, main memory has relatively-large storage capacity, but longer access latencies. However, high density and therefore low cost has made DRAM the most common choice for main memory.
- *Virtual Memory*: As applications provide more and more functionalities and become more and more sophisticated, a greater number of memory locations than physically available in the main memory, i.e., a larger address space, is required. The entire address space of the running application is stored on large magnetic or optical disks, which creates a virtually larger main memory. The most frequently used sections of the address space are kept in the main memory. When the processor requests data that

are stored on the disk, they are swapped in the main memory, in exchange of the data that are already in memory but are not currently being used.

2.2. Processor–Memory Gap

There are a number of reasons for the growing disparity between the performance of processor and memory. The division of the semiconductor industry into microprocessor and memory camps is perhaps the primary one. Fast increases in clock frequencies, pipelining, superpipelining, superscalar execution, very long instruction word (VLIW), explicitly parallel instruction computing (EPIC), simultaneous multithreading (SMT), and various other techniques are improving the microprocessor performance at a rate of 60% per year. In contrast, the performance of DRAM has been improving at less than 10% per year. Although each is improving exponentially, the exponent for microprocessors is substantially larger than that for DRAM. The difference between diverging exponentials is also exponential.

Due to the growing memory access latencies (relative to processor speed), any request that misses in the caches may take hundreds of processor cycles to satisfy. As fast as the processor may be, it must wait for the data to be transferred from memory to complete the instruction generating the miss. Thus, the performance of the system can be dominated by memory performance. Consequently, the disparity between processor and memory speed has become a primary obstacle to improve computer system performance, and will be a greater issue in the future.

The key solution to the processor–memory gap is to reduce the average memory access latency. Memory hierarchy provides reduced average memory access latency, and hence there are a lot of research efforts on efficient implementations of memory hierarchy. This involves primarily improving each of the following three factors: cache hit latency (T_{hit}), cache miss penalty (T_{miss}) and cache miss rate (R_{miss}) [**1**], as shown in Equation 1.

$$T_{memory} = T_{hit} + R_{miss} \times T_{miss} \tag{1}$$

More specifically, techniques have been developed to reduce miss rate by increasing the dimensions of caches and their blocks, increasing cache associativity, inserting victim and/or pseudo-associative caches, hardware pre-fetching, compiler-controlled prefetching, etc.. To reduce miss penalty, techniques like read priority over write on miss, sub-block placement, early restart and critical word first on miss, lockup-free caches, and multi-level caches have been proposed. To reduce hit time, researchers have proposed to use simple and small caches, and pipelined writes.

However, the above techniques that reduce memory access latency may increase the requirement for the processor–memory bus bandwidth, because they speedup the instruction processing rate, and may require transferring more data than are actually used by the processor, thereby increasing the absolute amount of memory traffic. Bandwidth constraints increase the response times for processor requests. Thus, the two factors of memory latency and memory

bandwidth are closely related. Improvement in memory bandwidth is critically necessary to support aggressive memory latency techniques.

Increases in VLSI integration density and the increasing importance of power consumption are leading to the use of chip-level multiprocessor (CMP) architectures. The move to CMPs substantially increases capacity pressure on the on-chip memory hierarchy. The 2006 ITRS Roadmap [**3**] predicts that transistor speed will continue to grow faster than DRAM speed and pin bandwidth. This means that cache miss costs will continue to increase. However, increases in on-chip cache size may block the addition of processor cores, thereby preventing the growth of the overall system performance.

Chapter 3

On-Line Memory Compression for Embedded Systems

Memory is a scarce resource during embedded system design. Increasing memory often increases packaging costs, cooling costs, size, and power consumption. In this chapter, we present the design and implementation of CRAMES, a novel and efficient software-based RAM compression technique for embedded systems. The goal of CRAMES is to dramatically increase effective memory capacity without hardware or application design changes, while maintaining high performance and low energy consumption. To achieve this goal, CRAMES takes advantage of an operating system's virtual memory infrastructure by storing swapped-out pages in compressed format. It dynamically adjusts the size of the compressed RAM area, protecting applications capable of running without it from performance or energy consumption penalties. In addition to compressing working data sets, CRAMES also enables efficient in-RAM filesystem compression, thereby further increasing RAM capacity.

CRAMES was implemented as a loadable module for the Linux kernel and evaluated on a battery-powered embedded system. Experimental results indicate that CRAMES is capable of doubling the amount of RAM available to applications running on the original system hardware. Execution time and energy consumption for a broad range of examples are rarely affected. When physical RAM is reduced to 62.5% of its original quantity, CRAMES enables the target embedded system to support the same applications with reasonable performance and energy consumption penalties (on average 9.5% and 10.5%), while without CRAMES those applications either may not execute or suffer from extreme performance degradation or instability.

The rest of this chapter is organized as follows. Section 3.1 begins with a brief introduction of the problem motivation and background, and continues with an overview of the CRAMES framework. Section 3.2 summarizes related techniques and their contributions. Section 3.3 describes the proposed memory compression technique and elaborates on the tradeoffs involved in the design of CRAMES. This section also proposes important design principles for software-based RAM compression techniques. Section 3.4 discusses the implementation of CRAMES as a Linux kernel module. Section 3.5 presents the experimental set-up, describes the workloads, and presents the experimental results in detail. Finally, Section 3.6 summarizes this chapter.

3.1. Introduction

The complexities and resource demands of modern embedded systems such as *personal digital assistants* (PDAs) and smart phones are constantly increasing. In order to support applications such as 3-D games, secure Internet access, email, music, and digital photography,

the memory requirements of embedded systems have grown at a much faster rate than was originally anticipated by their designers. For example, the total RAM and flash memory requirements for applications in the mobile phone market are doubling or tripling each year [**4**]. Although memory price drops with time, adding memory frequently results in increased embedded system packaging cost, cooling cost, size, and power consumption. For example, the HP iPAQ hx2755 PDA has a price 20% higher than its predecessor, the iPAQ hx2415. With the exception of a slight increase in CPU frequency (520 MHz for hx2415 and 624 MHz for hx2755), the hx2755 differs from the hx2415 only by making 2.2 times as much memory available to the user [**5**]. In addition, as embedded systems support new applications, their working data sets often increase in size, exceeding original estimates of memory requirements. Redesigning hardware and applications is not desirable as it may dramatically increase time-to-market and design costs.

The on-line memory compression technique described in this Chapter was originally motivated by a specific engineering problem faced by a corporation during the design of an embedded system for secure network transactions. After hardware design, the memory requirements of the embedded system's applications overran the initial estimate. There were two ways to solve this problem: redesign the embedded system hardware, thereby dramatically increasing time-to-market and cost, or make the hardware function as if it had been redesigned without actually changing it. The second approach was chosen, resulting in the technique described in this chapter. Note that, even for embedded systems capable of functioning on their current hardware platforms, it is often desirable to increase the number of supported applications if the cost of

doing so is small. In addition, the proposed technique has the potential to allow the hardware complexity and cost of some embedded systems to be reduced, without sacrificing the ability to run the desired applications.

3.1.1. Problem Background

The desire to minimize embedded system memory requirements has led to the design of various memory compression techniques. Memory compression for embedded systems is a complex problem that differs from the typical file compression problem. It can be divided into two categories: executable code compression and data compression, each of which is described below.

Embedded systems that support code compression decompress code during execution. Since code does not change during application execution[1], compression may be done off-line and can be slow. Decompression is done during application execution and therefore must be very fast. This means that, in code compression techniques, the performance of compression and decompression algorithms may differ. Another important characteristic of runtime code decompression is the need for random access during decompression: code does not execute in a sequential manner. Most code compression techniques are hardware-based, i.e., they are implemented with, and rely on, special-purpose hardware.

Although code compression was shown to be useful in reducing embedded system memory requirements, in many embedded systems it is more important to compress working data sets. The code segment is often small and can be stored in flash memory. Therefore code can *execute*

[1]The exception to this is self-modifying code, which is rarely used today.

in place (XIP), i.e., execute directly from flash memory without being copied to RAM. As a result, in many modern embedded systems, e.g., PDAs and smart phones, the majority of RAM is used for storing data, not code.

While code compression techniques are mostly hardware-based solutions, data set compression techniques are usually *operating system* (OS) based. Data compression presents more design challenges than code compression. In addition to the random access requirement of code compression, data must be written back to memory. Some algorithms that are suitable for code compression, i.e., those that allow slow compression but provide fast decompression, are inappropriate for data compression. In addition, allocation of compressed pages must be carefully managed to minimize memory, time, and energy use. Furthermore, locating a compressed page of data in memory must be fast to ensure good performance.

There are also techniques that compress main memory with a mix of both data and code. For example, a hardware compression/decompression unit can be inserted between the cache and main memory. Data in main memory are all stored in compressed format, while data in the cache are not compressed.

In summary, despite the existence of data set compression techniques in the literature, few, if any, have seen use in commercial products; practical memory compression still faces a number of problems, e.g., smart selection of pages to compress, careful scheduling of compression and decompression to minimize performance and power impact, elegantly dealing with the memory fragmentation problem introduced by compression, and avoiding the addition of special-purpose hardware, etc.

3.1.2. CRAMES Overview

We propose a software-based RAM compression technique, named *CRAMES* (Compressed RAM for Embedded Systems), that increases effective memory capacity without requiring designers to add physical memory. RAM compression for embedded systems is a complex problem that raises several questions. Can a RAM compression technique allow existing applications to execute without performance and energy consumption penalties? Can new applications with working data sets that were originally too large for physical memory be automatically made to execute smoothly? What compression algorithm should be used, and when should compression and decompression be performed? How should the compressed RAM area be managed to minimize memory overhead? How should one evaluate RAM compression techniques for use in embedded systems?

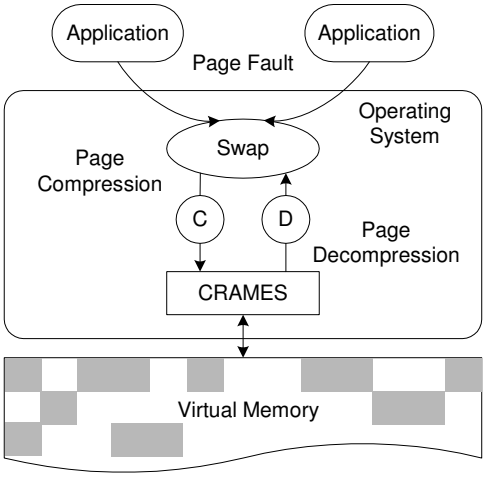

FIGURE 3.1. Overview of CRAMES.

In order to minimize performance and energy consumption impact, CRAMES takes advantage of the OS virtual memory swapping mechanism to decide which data pages to compress

and when to perform compression and decompression. Figure 3.1 illustrates the data path of CRAMES in an operating system. Note that virtual memory contains both uncompressed areas (white) and compressed areas (gray). CRAMES requires an MMU. However, no special-purpose hardware is required.

In addition, multiple compression algorithms and memory allocation methods were experimentally evaluated for CRAMES to efficiently compress and manage data; the most promising ones were selected. CRAMES dynamically adjusts the size of the compressed area during operation based on the amount of memory required, so that applications capable of running without memory compression do not suffer performance or energy consumption penalties as a result of its use. In addition to data set compression, CRAMES is also effective for in-RAM filesystem compression, thereby further expanding system RAM.

CRAMES has been implemented as a loadable Linux kernel module for maximum portability and modularity. Note that the technique can easily be ported to other modern OSs. The module was evaluated on a battery-powered PDA running an embedded version of Linux called Embedix [**6**]. This embedded system's architecture is similar to that of modern mobile phones. CRAMES requires the presence of an MMU. However, no other special-purpose hardware is required. MMUs are becoming increasingly common in high-end embedded systems. We evaluate our technique using well-known batch benchmarks as well as interactive applications with *graphical user interfaces* (GUIs). A PDA user input monitoring and playback system was designed to support the creation of reproducible interactive GUI benchmarks. Our results show

that CRAMES is capable of dramatically increasing memory capacity with small performance and power consumption costs.

3.2. Related Work

This section discusses related work in the memory and filesystem compression domain. Recent memory compression research has focused on two topics: code compression and data compression. Code compression techniques are often hardware-based, with the goal of reducing system memory requirements. In contrast, data compression techniques are often software-based, targeting at improving system performance by reducing disk I/O. There are also hardware-based techniques that compress both code and data to reduce RAM requirements. In addition, filesystem compression is another area of interest that exhibits similar problems to the ones addressed in this chapter. The primary goal of filesystem compression is to save disk space and furthermore, to reduce disk I/O by transferring compressed data.

3.2.1. Compress Both Code and Data

IBM MXT [7] is a technique for hardware-based main memory compression. In this technique, a large, low-latency, shared cache sits between the processor bus and a content-compressed main memory. A hardware compression unit is inserted between the cache and main memory. The authors reported a typical main memory compression ratio between 16.7% and 50%, as measured in real-world system applications. *Compression ratio* is defined as compressed memory size divided by original memory size. This compression ratio is consistent with that achieved by CRAMES.

Benini and Bruni [**8**] also proposed to insert a hardware compression/decompression unit between the cache and RAM, so that data in RAM are all stored in compressed format, while data in the cache are not compressed. Kjelso, Gooch, and Jones [**9**] proposed a hardware-based compression scheme designed for in-memory data. Their X-match algorithm, which uses a small dictionary of recently-used words, is designed for hardware implementation. The above approaches may reduce embedded system RAM requirements and power consumption. However, they require changes to the underlying hardware and thus cannot be easily incorporated into existing processors and embedded systems.

3.2.2. Compress Only Code

Early techniques for reducing the memory requirements of embedded systems were mostly hardware-based, i.e., they were implemented with, and relied on, special-purpose hardware. *Code compression* techniques [**10**, **11**, **12**, **13**] store instructions in compressed format and decompress them during execution. In these techniques, compression is usually done off-line and can be slow, while decompression is done during execution by special hardware and must be very fast. Other techniques focus on modifying the instruction set, which led to the design of denser instruction sets geared for the embedded market, e.g., the Thumb architecture [**14**].

3.2.3. Compress Only Data

Most previous work on software-based on-line data compression falls into two categories: compressed caching and swap compression. The main goal of compressed caching and swap

compression is to improve system performance. These techniques target general-purpose systems with hard disks.

3.2.3.1. *Compressed Caching*

Compressed caching [**15**, **16**, **17**, **18**] was proposed by a number of researchers to solve the data memory compression problem. The objective is to improve system performance by reducing the number of page faults that must be serviced by hard disks, which have much longer access times than RAM. Early work by Douglis [**15**] proposed a software-based compressed cache, which uses part of the memory to store data in LZRW1 compressed format. A study on compressed caching by Kjelso, Gooch, and Jones [**19**] used simulations to demonstrate the efficacy of compressed caching. They addressed the problem of memory management for variable-size compressed pages. Their experiments also used the LZRW1 compression algorithm. Furthermore, Russinovich and Cogswell [**16**] presented a thorough analysis of the compression algorithms used in compressed caching. Wilson, Kaplan, and Smaragdakis [**17**] also used simulations to prove a consistent benefit from the use of compressed virtual memory. In addition, they proposed a new compression algorithm suited to compressing in-memory data representations.

3.2.3.2. *Swap Compression*

Swap Compression [**20**, **21**, **22**, **23**] compresses swapped out pages and stores them in a software cache in RAM. Cortes, Eles, and Peng [**21**] explored the introduction of a compressed swap cache in Linux for improving swap performance and reducing memory requirements.

Their work targeted general-purpose machines with large hard drives, for which saving space in memory was not a design goal.

Roy and Prvulovic [**22**] proposed a memory compression mechanism for Linux with the goal of improving performance. Again, their approach targets systems with hard disks. They reported speed-ups from 5% to 250% depending on the application. They did not consider the use of this technique in systems that do not contain disks, i.e., most embedded systems.

Rizzo et. al. [**20**] proposed a RAM compression algorithm based on Huffman coding. Their approach is useful for disk-less systems; a swap area is introduced in RAM to hold compressed pages. They compared the compression ratio of their algorithm with several well-known algorithms and demonstrated desirable results. However, the compressed *Swap-on-RAM* architecture was not implemented in any OS. The impact on memory reduction and power consumption were not evaluated on embedded systems.

3.2.4. Compressed Filesystems

Compressed filesystems are typically used for reducing disk or RAM disk (i.e., an in-RAM block device that acts as if it were a hard disk) space requirements, and hence are usually off-line and read-only. There has been some work by researchers and the Linux community on introducing compressed filesystems into the Linux kernel.

Cramfs [**24**] is a read-only compressed Linux filesystem. It uses Zlib to compress a file one page at a time and allows random page access. The meta-data is not compressed, but is expressed in a condensed form to minimize space requirements. Since Cramfs is read-only,

one must first create a compressed disk image with the `mkcramfs` utility off-line before the filesystem is mounted. Cramfs is currently being used on a variety of embedded systems.

Cloop [**25**] is a Linux kernel module that enables compressed loopback [**26**] filesystem support. The module provides a filesystem-independent, transparently decompressed, and read-only block device, i.e., a device that stores and accesses data in fixed-size blocks. A user can mount a compressed filesystem image like a block device. Data is automatically decompressed when the device is addressed. Like Cramfs, users must first create a compressed image and mount it in read-only mode. Cloop uses the Zlib compression algorithm.

CBD [**27**] is a Linux kernel patch that adds a compressed block device designed to reduce the size of filesystems. CBD is a disk-based read-only compressed block device that is very similar to Cloop. A compressed filesystem image needs to be created off-line. Writes to CBD are locked in the buffer cache of memory and are never sent to the physical device. CBD also uses the Zlib compression algorithm.

3.2.5. Other Related Work

There exist other software-based memory compression techniques that cannot easily be included in any of the main categories listed above. *RAM Doubler* [**28**] is a technique that expands the memory available to Mac OS via three methods. First, it attempts to locate small chunks of RAM that applications are not actively using and makes that memory available to other applications. Second, RAM Doubler attempts to locate and compress pages that are not likely to be accessed in the near future. Finally, if the first two attempts fail, the system swaps rarely-accessed data to disk. Although RAM Doubler allows more applications to run simultaneously,

applications with memory footprints that exceed physical memory still cannot run. In contrast, CRAMES uses compression to increase available memory, allowing applications to run to completion when their memory requirements exceed the physical memory.

Heap compression for memory-constrained Java environments [**29**] enables the execution of Java applications using a smaller heap footprint on embedded Java virtual machines (JVMs). This work introduced a new garbage collector, the Mark-Compact-Compress collector (MCC), that compresses objects when heap compaction is not sufficient for creating space for the current allocation request. This technique is useful for reducing the size of the heap segment of an embedded Java application. However, the stack segment is not compressed. CRAMES is able to compress the application stack segment as well as the heap segment, allowing it to further increase available RAM. More importantly, CRAMES works with applications implemented in any programming language. For example, if activated on a system running a Java interpreter, the interpreter and application data will automatically be compressed.

3.2.6. Summary and CRAMES Contributions

In summary, despite the existence of memory compression schemes, few have seen use in commercial embedded systems for one or more of the following reasons: (1) they assume off-line compression and thus can not handle dynamic data memory, (2) they require redesign of the target embedded system and the addition of special-purpose hardware, or (3) their performance and energy consumption impacts have not been considered, or are unacceptable, for typical disk-less embedded systems.

CRAMES makes the following main contributions: (1) unlike previous work, it handles both on-line data memory compression and in-RAM filesystem compression; (2) it requires no special hardware and thereby requires no system redesign; (3) it requires no change to applications; (4) the compression algorithm and memory allocation method are carefully selected to minimize performance and energy consumption overheads; and (5) CRAMES targets disk-less embedded systems. CRAMES is portable and can be used on a variety of platforms. Moreover, the technique is general enough to permit its use with any modern OS supporting virtual memory. CRAMES dramatically increases the available RAM to embedded systems at small performance and power consumption costs (please refer to Section 3.5).

3.3. The CRAMES Architecture

This section describes the CRAMES architecture. CRAMES divides the RAM of an embedded system into two portions: one containing compressed data pages and the other containing uncompressed data pages as well as code pages. We call the second area the *main memory working area*. Consider a disk-less embedded system in which the memory usage of one memory-intensive process (or several such processes) increases dramatically and exceeds system RAM. If no memory compression mechanism is used, the process may not proceed; there is no hard disk to which it may swap out pages to provide more RAM. However, with CRAMES, the kernel may compress and move some of the pages within the main memory working area to the compressed area so that the process may continue running. When data in a compressed page is later required by a process, the kernel will quickly locate that page, decompress it, and copy

it back to the main memory working area, allowing the process to continue executing. With CRAMES, applications that would normally be unable to run to completion correctly operate on an embedded system with limited RAM.

3.3.1. Design Principles

The goal of CRAMES is to significantly increase available memory with minimal performance and energy penalties, and without requiring additional hardware. We follow these principles to achieve this goal:

(1) **Carefully select and schedule pages for compression**. This is essential to guarantee correct operation. The selected pages must be compressed when the memory usage of active processes exceeds the main memory working area. One may view the uncompressed memory area as a cache for the compressed memory area. Therefore, frequently accessed pages should be placed in the uncompressed area rather than the compressed area to minimize access time. Compression and decompression must be carefully scheduled to avoid application termination due to memory exhaustion, which may happen when the amount of free memory falls below a predefined threshold or when a memory request cannot be satisfied.

(2) **Use a performance and energy efficient compression algorithm with low compression ratio and memory overhead.** Compression ratio gives a measure of the compression achieved by a compression algorithm on a page of data. It is defined as compressed page size divided by original page size. Therefore, a low compression

ratio indicates better performance than a high one. Using a high-quality compression algorithm is crucial to ensure that CRAMES can dramatically increase the amount of perceived memory with small performance and energy consumption penalties. Compressing/decompressing pages and moving them between the uncompressed memory and compressed memory consumes time and energy. The compression algorithm used in an embedded system must have excellent performance and low energy consumption. The algorithm must provide a low compression ratio, with small working memory requirements, to substantially increase the amount of usable memory, thereby enabling new applications to run or allowing the amount of physical memory in the embedded system to be reduced while preserving functionality. However, trade-offs exist between compression speed and compression ratio. As shown in Section 3.3.2.2, slower compression algorithms usually have better compression ratios, while faster compression algorithms have poorer compression ratios. In addition, algorithms with lower compression ratio have shorter latencies to move pages to the compressed area due to their smaller sizes.

(3) **Organize the compressed area in non-uniform-size slots.** Since sizes of compressed pages vary widely, efficiently distributing and locating data in the compressed memory area is challenging. Even if memory is generally organized into identically-sized pages, the compressed area may not be organized into uniform-size slots because the sizes of compressed pages vary widely. Thus, compression transforms the easy problem of finding a free page in an array of uniform-size pages into the hard problem of

finding an arbitrary-size range of free bytes in an array of bytes. This problem is closely related, but not identical, to the classical memory allocation problem. The compressed memory manager must be fast and high quality, i.e., it must minimize waste resulting from fragmentation.

(4) **Dynamically adjust the size of the compressed area.** The compressed area must be large enough to provide applications with additional memory, when necessary. However, it should stay out of the way when the applications do not require additional memory to avoid performance and energy consumption penalties. This can be achieved by using a compressed memory area just barely large enough to execute the currently running applications. In other words, the compressed area should dynamically change its size based on the amount of memory required by the currently executing applications.

(5) **Minimize the memory overhead.** Additional space in memory is required to index and tag the compressed pages, allowing them to be located in the future. Moreover, compressed data pages may vary in size and cut memory into chunks with different sizes. The classic memory allocation internal fragmentation problem is potentially relevant. CRAMES must try to minimize the memory overhead of compression, fragmentation, and indexing compressed pages to ensure an improvement in physical memory capacity.

3.3.2. Design Overview

This section provides an overview of CRAMES. Three components are closely related: OS virtual memory swapping, block-based data compression, and kernel memory allocation. After

giving a brief overview of these components, we describe the design of CRAMES in accordance with the design principles described in Section 3.3.

3.3.2.1. *CRAMES and Virtual Memory Swapping*

When a system with virtual memory support is low on memory (either when the amount of free memory falls below a predefined threshold or when a memory request cannot be satisfied), the least recently used data pages are swapped out from memory to, conventionally, a hard disk. Swapping allows applications, or sets of applications, to execute even when the size of RAM is not sufficient. CRAMES takes advantage of swapping to decide which pages to compress and when to perform compression and decompression; the compressed pages are then swapped out to a special *compressed RAM device*. Figure 3.2 illustrates the logical structure of the swapping mechanism on the compressed RAM device. RAM is divided into two parts: the uncompressed area (white) and the compressed swap area (grey). Neither part is contiguous, i.e., each consists of a set of non-uniform size chunks. The uncompressed areas swap out compressed pages to the compressed swap areas, which are linked together.

The compressed RAM device should vary its memory usage over time according to memory requirements. Unlike conventional swap devices, which are typically implemented as disk partitions or files within a filesystem on a hard disk, the compressed RAM device is not a contiguous region with fixed size; instead, it is a linked list of compressed RAM areas (as shown in Figure 3.2). Whenever the compressed RAM device is not large enough to handle a new write request, it requests more memory from the kernel. If successful, the newly allocated chunk of memory is linked to the list of existing compressed swap areas; otherwise, the combined data

set of active processes is too large even after compression, making it necessary for the kernel to kill one or more of the processes. Recall that a request to swap out a page is generated when physical memory has been nearly exhausted. If attempts to reserve a portion of system memory for the compressed memory device were deferred until this time, there would be no guarantee of receiving the requested memory. Therefore, the compressed swap device starts with a small, predefined size but expands and contracts dynamically.

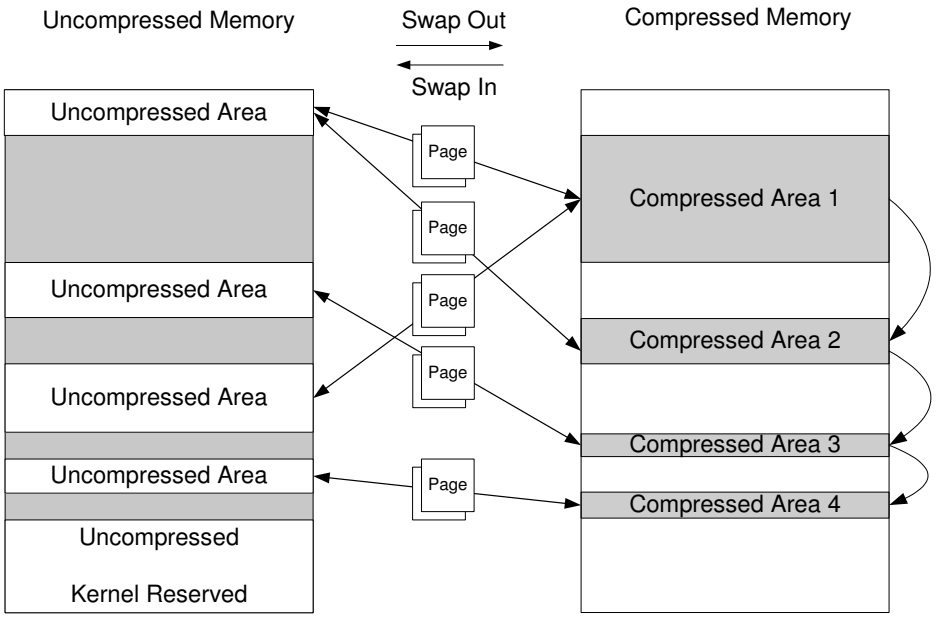

FIGURE 3.2. Logical structure of swapping on compressed RAM device.

Since a copy of a program's text segment is kept in its executable file, a text page need not be copied to the swap area or written back to the executable file because it may not be modified. Therefore, swapping is not useful for compressing code pages. It would potentially be beneficial to compress executable files that are stored in flash memory or electrically programmable read-only memory (ROM) filesystem by (partially) copying them to RAM on execution. However,

this can be accomplished with existing techniques and tools, e.g., JFFS2 [**30**] with demand paging. These techniques may be used in combination with CRAMES.

3.3.2.2. *CRAMES and Block-based Data Compression*

To ensure good performance for CRAMES, appropriate compression algorithms must be identified and/or designed. Classical data compression is a mature area; a number of algorithms exist that can effectively compress data blocks, which tend to be small in size, e.g., 4 KB, 8 KB, or 16 KB. This is important for CRAMES because it performs compression at the page level. We evaluated existing data compression algorithms that span a range of compression ratios and execution times: bzip2, zlib (with level 1, 9, and default), LZRW1-A, LZO [**31**], and RLE (Run Length Encoding).

Figure 3.3 illustrates the compression ratios and execution times of the evaluated algorithms and Table 3.2 gives their memory requirements. For these comparisons, the source file for compression is a 64 MB swap data file from a workstation running SuSE Linux 9.0, which is later divided into uniform-sized blocks to perform block-based compression. Although bzip2 and zlib have the best compression ratios, their execution times are significantly longer than LZO, LZRW1-A, and RLE. In addition, the memory overheads of bzip2 and zlib are sufficient to starve applications in many embedded systems. Among these candidates, LZO has the best all-around performance. It has a low compression ratio and low working memory requirements as well as allowing fast compression and fast decompression. Therefore, LZO appears to be an appropriate compression algorithm for dynamic data compression in low-power embedded systems and we further evaluated the performance of LZO when used in CRAMES on real

TABLE 3.1. Evaluated Compression Algorithms

Application	**Algorithm**	**Characteristics**
bzip2	Burrows-Wheeler Transform	Compression ratios within 10% of the state-of-the-art algorithms; relatively slow compression; decompression is faster than compression.
zlib	LZ-family	Fast compression/decompression; good compression ratio.
LZRW1-A	LZ-family	Very fast compression/decompression; good compression ratio; very low working memory requirement.
LZO	LZ-family	Very fast compression; extremely fast decompression; favors speed over compressibility; low working memory requirement.
RLE	Run-Length Encoding	Very simple, and extremely fast, poorer compression ratio for most data; no working memory requirement.

TABLE 3.2. Memory Overhead of Compression Algorithms

	bzip2	zlib	LZO	LZRW1-A	RLE
Compression	7600 kB	256 KB	64 KB	16 KB	0
Decompression	3700 kB	44 KB	0	16 KB	0

embedded systems. In Chapter 4, we will describe the design of a new, faster on-line memory compression algorithm.

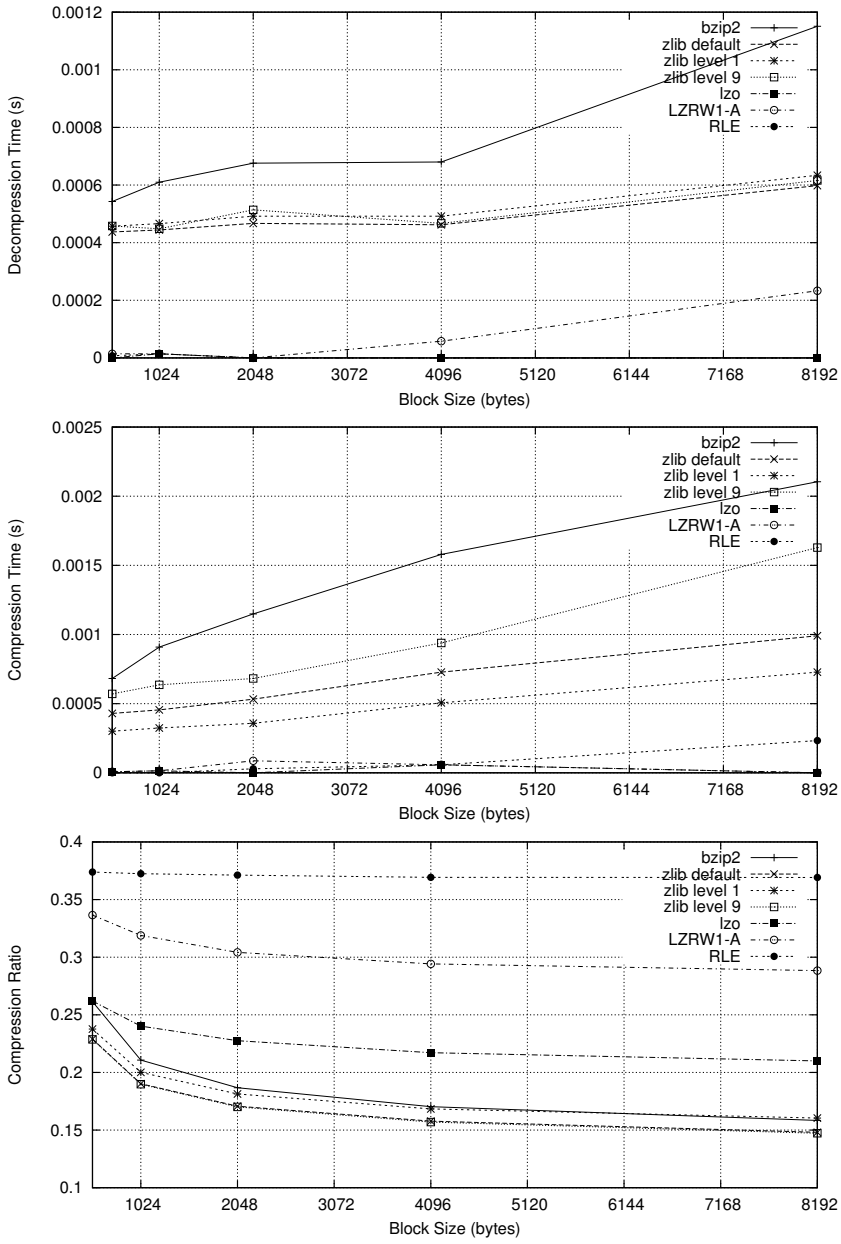

FIGURE 3.3. Comparison of evaluated compression algorithms.

3.3.2.3. *CRAMES and Kernel Memory Allocation*

In addition to scheduling compression and using an appropriate block compression algorithm, CRAMES must efficiently organize the compressed swap device to enable fast compressed page access and minimal memory waste. More specifically, the following problems must be solved: (1) efficiently allocating or locating a compressed page in the swap device, (2) mapping between the virtual locations of uncompressed pages and actual data locations in the compressed swap device, and (3) maintaining a linked list of free slots in the swap device that are merged when appropriate.

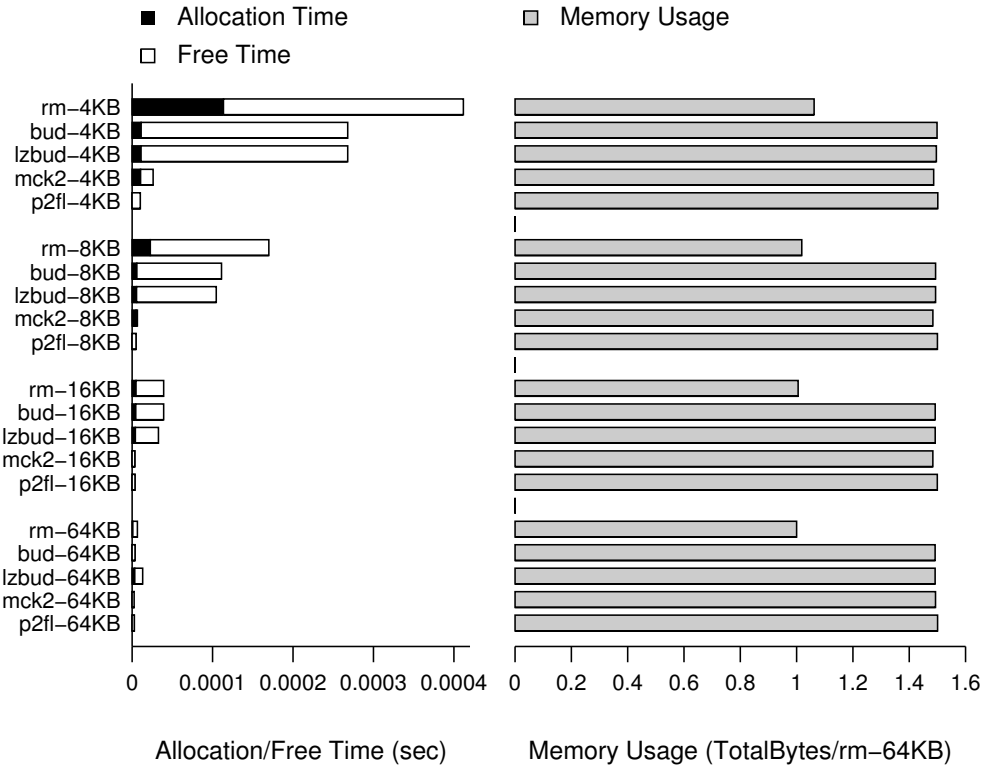

FIGURE 3.4. Memory usage of evaluated memory allocation methods.

The compressed RAM device memory management problem is related to the *kernel memory allocation* (KMA) problem. The virtual memory management system must maintain a map from virtual pages to the locations of pages in physical memory, allowing it to satisfy requests for virtually contiguous memory by allocating several physically non-contiguous pages. In addition, the kernel maintains a linked list of free pages. When a process requires additional pages, the kernel removes them from the free list; when the pages are released, the kernel returns them to the free list.

The CRAMES memory manager builds upon methods used in KMA. In order to identify the most appropriate memory allocation method for the RAM compression problem, the following five memory allocators were implemented and applied to requests generated from the 64 MB swap data file that was used to evaluate compression algorithms: Resource Map Allocator (rm), Simple Power-of-Two Freelists (p2fl), McKusick-Karels Allocator [**32**] (mck2), Buddy System [**33**] (bud), and Lazy Buddy Algorithm [**34**, **35**] (lzbud). As observed for block-based compression algorithms, there is a tradeoff between algorithm quality and performance, i.e., techniques with excellent memory utilization achieve it at the cost of allocation speed and energy consumption.

Figure 3.4 illustrates the impact of chunk size on allocation/free time and total memory usage, including fragmentation and bookkeeping overheads, for each of the five memory allocators. For example, `rm-4KB` stands for Resource Map allocator with a chunk size of 4 KB. Recall that the CRAMES memory manager requests memory from the kernel in linked chunks

in order to dynamically increase and decrease the size of the compressed memory area. Although Resource Map requires the most time when the chunk size is smaller than 16 KB, its execution time is as good as, if not better than, the other four allocators when the chunk size is larger than 16 KB. In addition, Resource Map always requires the least memory from the kernel. Therefore, Resource Map is a good choice for CRAMES when the chunk size is larger than 16 KB; it is the default memory allocation method. In our experiments described in Section 3.5, the chunk size is set to be 64 KB. Note that for embedded system memory sizes less than or equal to 16 KB, faster allocators with good memory usage ratios may be considered, e.g., the McKusick-Karels allocator.

During the implementation of CRAMES, we designed a user-space simulator to assist code debugging and performance profiling. Profiling results indicated that compression and decompression are responsible for the main performance penalty in CRAMES. The overhead resulting from Resource Map memory management was no more than 1/10 of that from compression and decompression. In addition, to determine the impact of fragmentation during the evaluation of the Resource Map allocator, we tested CRAMES on a workstation running various applications to trigger swapping. We observed that the fragmented memory in CRAMES is constantly under 5%. We believe that fragmentation is not a problem for CRAMES with the Resource Map allocator. To summarize, although there are many recent advances of modern memory allocators, the performance of Resource Map is adequate for this application and using faster allocators would not improve overall performance of CRAMES.

3.3.3. Using CRAMES with the Filesystem

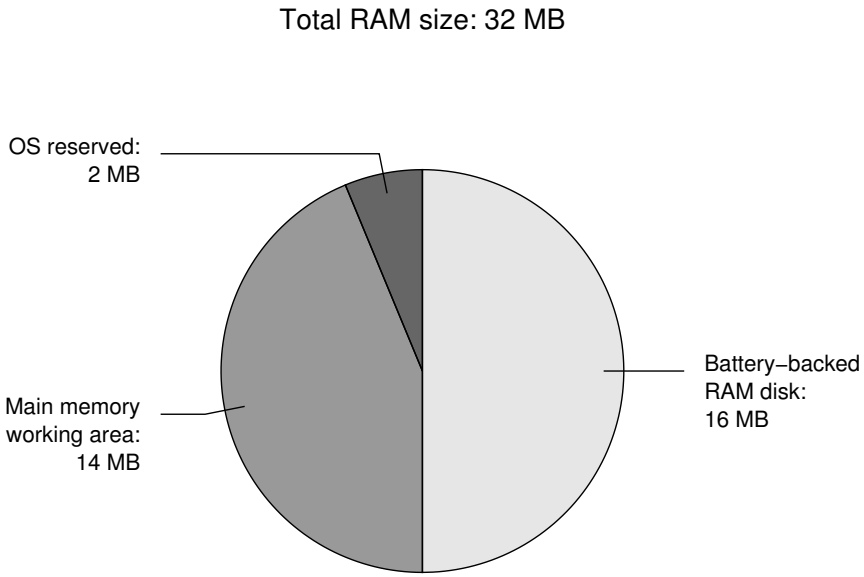

FIGURE 3.5. RAM regions in a Sharp Zaurus SL5600.

As illustrated in Figure 3.5, in a typical embedded system (Sharp Zaurus SL-5600 PDA) with 32 MB RAM, only 14 MB of memory are available for user applications and system background processes. A significant portion (50% or 16 MB) of RAM is used to create a battery-backed RAM disk for the filesystem. A RAM disk is a common RAM device without compression. In this chapter, the RAM disk is usually referred to as a filesystem holder, while the RAM device is used to indicate a swap device.

Although the design of compressed filesystems has been studied extensively in recent years, no solution exists for readable/writable RAM disks. For example, Cramfs [**24**] is a compressed

filesystem targeting embedded systems, but it is read-only. JFFS2 [**30**] is a compressed, readable, and writable filesystem, but it is for use with flash memory rather than RAM disks. Therefore, it is desirable for a memory compression technique to support the compression of RAM disks with filesystems in addition to the compression of data in main memory. Although not designed with filesystem compression as its primary goal, CRAMES supports compressed RAM disks containing arbitrary filesystem types.

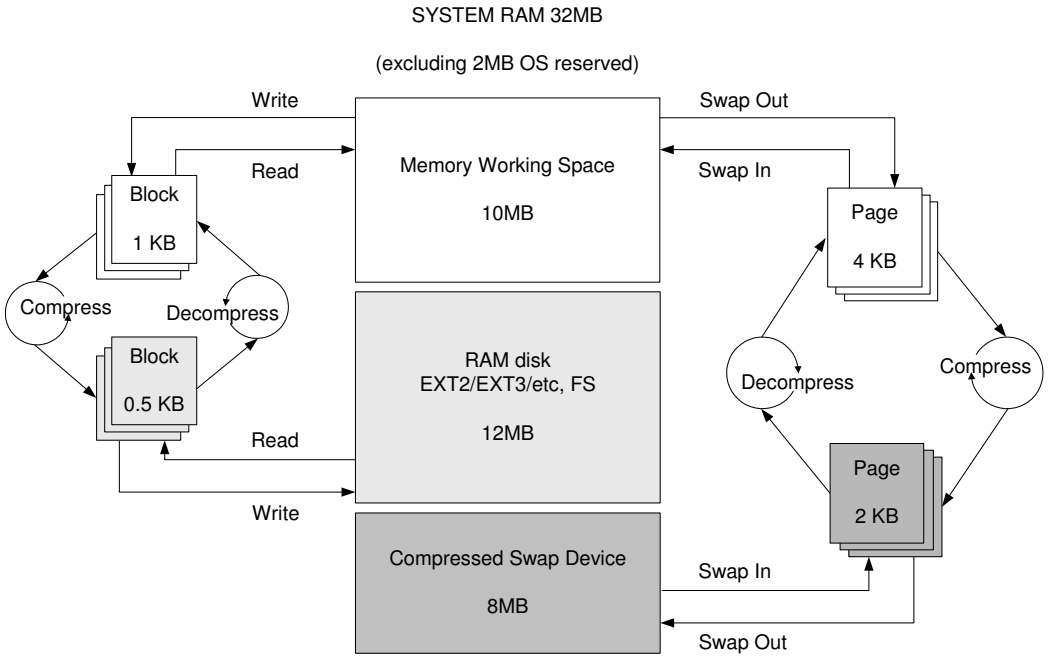

FIGURE 3.6. A possible system RAM partition.

Figure 3.6 illustrates a possible new RAM partitioning scheme, using CRAMES on an embedded system with its 32 MB of RAM originally partitioned in approximately the same way as previously described for the Sharp Zaurus SL-5600. The gray areas in the figure correspond to compressed memory and the white areas correspond to uncompressed memory. Darker colored

areas have lower compression ratios, i.e., better compression. Without compression, 30 MB of memory are available (2 MB are reserved for the OS kernel). These 30 MB of RAM are divided into two parts: 14 MB of main memory working area and a 16 MB RAM disk for filesystem storage. When CRAMES is used, the 30 MB of RAM is divided into three parts: a 10 MB main memory working area, a 12 MB compressed RAM device for filesystem storage, and an 8 MB compressed RAM device for swapping. Suppose the average memory compression ratio for the swap device is 50%. The maximum capacity this device can provide is $8 \div 0.5 = 16\,\mathrm{MB}$. The maximum total memory addressable is therefore $10 + 16 = 26\,\mathrm{MB}$. If the average compression ratio for the RAM disk is 60%, the total filesystem storage available for the system is $12 \div 0.6 = 20\,\mathrm{MB}$. Thus, the maximum RAM capacity of the system is expanded to $26 + 20 + 2 = 48\,\mathrm{MB}$ with little degradation in performance or power consumption (as shown in Section 3.5). This example assumes fixed-size compressed RAM devices to simplify the explanation; however, CRAMES supports dynamic, automatic compressed RAM resizing.

3.4. Implementation of CRAMES

This section describes implementation details. CRAMES has been implemented and evaluated as a loadable module for the Linux 2.4 kernel. The module is a special block device (i.e., a random access device that stores and retrieves data in blocks) using system RAM. It may serve as both a swap device and a storage area for filesystems. Although the block size for a swap device is 4 KB (the page size in Linux), the block size may vary (e.g., 4 KB, 16 KB, or 64 KB)

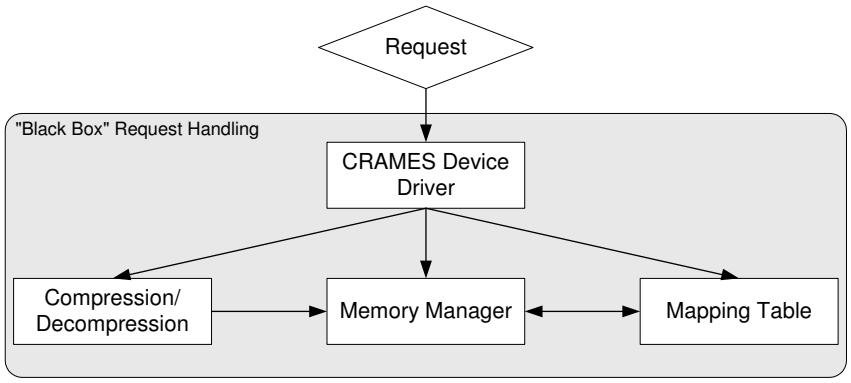

FIGURE 3.7. CRAMES device request handling.

when it is used as a filesystem storage area. This section describes the structure of a CRAMES device. It focuses on the use of CRAMES as a swap device.

3.4.1. CRAMES Request Handling

CRAMES is a special block device. A block device must register itself with the kernel to become accessible. During registration, it is necessary to report (1) block size and number of blocks[2], i.e., capacity, and (2) a request handling function that the kernel will call when there is a read/write request for this device. CRAMES reports an estimated maximum capacity to the kernel, although its actual memory requirement is usually substantially smaller. It enables on-the-fly data compression and decompression via its request handling procedure, which consists of four steps. For a write request, these steps are (1) compressing a block that is written to the device, (2) allocating memory for a compressed block and placing the compressed block in allocated memory, (3) managing the mapping table, and (4) attempting to merge free slots. For a read request, these steps are (1) locating a compressed block with an index number, (2)

[2]The kernel sets the block size of a block device to page size (often 4 KB) and adjusts the number of blocks accordingly when the device is used as a swap device.

decompressing a block that is read from the device, (3) releasing the memory occupied by this compressed block if it contains obsolete data, (4) managing the mapping table if memory is released, and (5) attempting to merge free slots if memory is released. Figure 3.7 depicts the logical structure of the CRAMES request handling procedure. A CRAMES device is like a black box to the system: compression, decompression, and memory management are all handled within the device.

3.4.1.1. *Mapping Table*

Data in a block device are always requested by block indices, regardless of whether the device is compressed. CRAMES creates the illusion that blocks are linearly ordered in the device's memory area and are equal in size. To convert requests for block numbers to their actual addresses, CRAMES maintains a *mapping table*, which may be direct-mapped or hashed. In a direct-mapped table, each entry is indexed by its block number. In a hash table, the key of each entry is a block number. The memory overhead of a direct-mapped table is higher because it may maintain block indices that are never used. However, searching in such a table is extremely fast. In contrast, a hash table minimizes the memory overhead by only keeping block indices that are actually accessed. However, the search time is longer. When evaluating CRAMES on a Sharp Zaurus SL-5600 PDA (see Section 3.5) we used a direct-mapped table because it is small (at most 16 KB) and fast.

Regardless of mapping table implementation style, the data field of each entry must contain the following information.

TABLE 3.3. Mapping Table in CRAMES

index	used	compressed	addr	size
0	1	0	0xa3081a3d	52
1	1	1	0xa3081a71	1090
...	...	...	...	...
4	1	1	0xa3081396	63
5	0	1	0xa3081004	910
6	0	1	0xa30813d9	1684
...	...	...	...	...
128	1	1	0xa30a3faa	80

- `Used` indicates whether it is a valid swapped-out block. This field is especially important for CRAMES to decide whether a compressed block may be freed.
- `Compressed` indicates whether a swapped-out block is in compressed format. When compression fails due to internal errors in the compression algorithm or the compressed size exceeds the original block size, CRAMES aborts compression and stores the original block. This field is necessary to guarantee correctness, even though such cases are rare. Another option would be to add additional information at the beginning of a compressed block to indicate whether it is compressed or not. However, doing this would require modification of compression algorithms, and therefore should be avoided.
- `Addr` records the actual address of that block.
- `Size` keeps the compressed size of a block.

Table 3.3 is a mapping table trace collected from experiments. In the Linux kernel, swapping is performed at the page level. Therefore, once a block device is used as a swap device with the command `mkswap`, the kernel swap daemon first sets the block size of the device to page size, i.e., 4 KB. From the swap daemon's perspective, each swap area consists of a sequence of pages. The first page of a swap area is used to persistently store information about the swap area and is also compressed in CRAMES. Starting from page 1, pages are used by the kernel swap daemon to store swapped-out pages that are compressed by CRAMES.

3.4.1.2. *Memory Manager*

Section 3.3.2.3 reveals that the KMA techniques can help in building an efficient memory allocator for CRAMES. Recall that the CRAMES memory allocator must efficiently handle three problems: (1) locating compressed blocks during reads, (2) finding free locations to store newly compressed blocks during writes, and (3) merging free slots to enable memory reuse.

The CRAMES memory manager must optimize the conflicting objectives of performance and allocation efficiency. Based on the experimental evaluation described in Section 3.3.2.3, CRAMES uses Resource Map as its default allocator to provide the best performance. Regardless of the KMA technique used in the CRAMES memory allocator, it is necessary to determine when a compressed block may safely be discarded from the device. For a compressed swap device, a compressed page may be freed under two circumstances: (1) the current request is a *read* and the requested page only belongs to one running process or (2) the current request is a *write* and the requested page has been written in previous requests.

It is straightforward for CRAMES to free a page if that page belongs to only one process and that process has just read the page back to main memory. However, complications arise when a page is shared by multiple processes. In this case, a page can be swapped in by one process and still reside in the swap area for other processes that share it. After a read request, CRAMES must first check the usage count of the page just read, to see whether after this read no other process will expect the page to reside in the swap area. If the page has no additional references, the CRAMES memory manager can proceed to free it. Likewise, a write request to a previously-written page indicates that the kernel swapper knows this page contains data from a terminated process and therefore can be overwritten by a new swapped-out page. Consequently, CRAMES can safely free the old page and allocate memory for the new compressed page.

3.4.1.3. *Request Handling Flow*

Figure 3.8 graphs the flow of the CRAMES request handling procedure for a compressed swap device. Unlike a RAM device, a given page need not always be placed at the same fixed offset. For example, when the driver receives a request to `read page 7`, it checks mapping table entry `tbl[7]`, gets the actual address from `addr` field, checks the `compressed` field to determine whether the page is compressed and, if it is, gets the compressed page size from the `size` field. Page 7 is then decompressed. Subsequently CRAMES checks the usage counter for page 7 to decide whether to free this page after this read. Finally, the request returns successfully.

Handling write requests is more complicated. When the driver receives the request to `write to page 7`, it first checks the mapping table entry `tbl[7]` to determine whether the `used` field

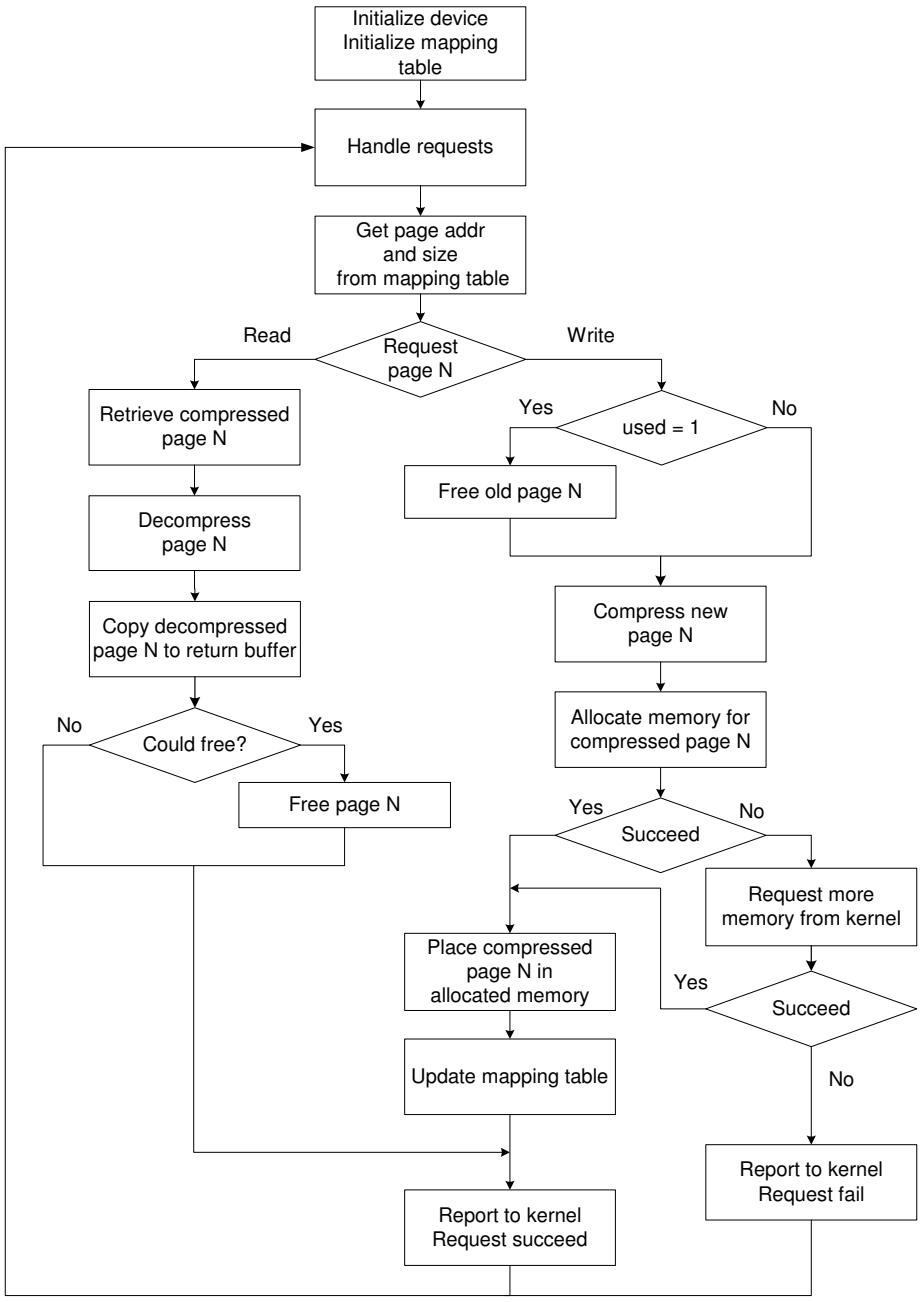

FIGURE 3.8. Handling request in CRAMES device.

is 1. If so, the old page 7 may safely be freed. After this, the driver compresses the new page

7, requires the CRAMES memory manager to allocate a slot of the compressed size for the new page 7, and places the compressed page 7 into the slot.

3.4.2. CRAMES and RAM Disk Comparison

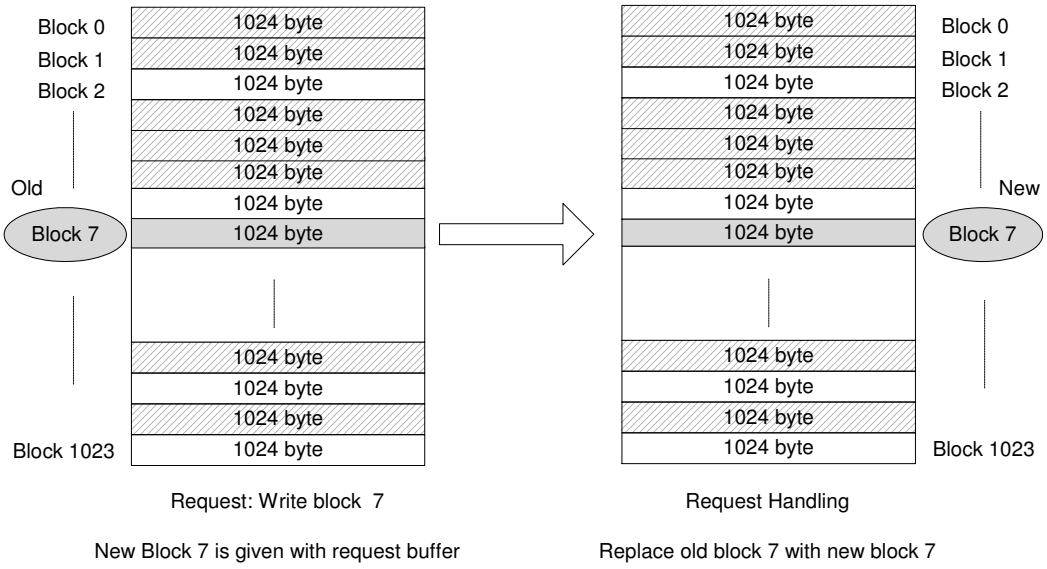

FIGURE 3.9. Linear, fixed-size blocks in RAM disk.

Figure 3.9 illustrates the logical structure and request handling of a RAM disk. As shown in the figure, the virtually contiguous memory space in a RAM disk is divided into fixed-size blocks. Shaded areas in the device memory represent occupied blocks and white areas represent free blocks. Upon initialization, a RAM disk requests a virtually contiguous memory region from the kernel. This memory region is then divided into uniform fixed-size blocks. When the RAM disk receives a read request for a block, it first locates the block by its index and then copies the data in that block to the request buffer. When it receives a write request, it first

locates that block in the same way, then replaces the data in that block with the data in the request buffer.

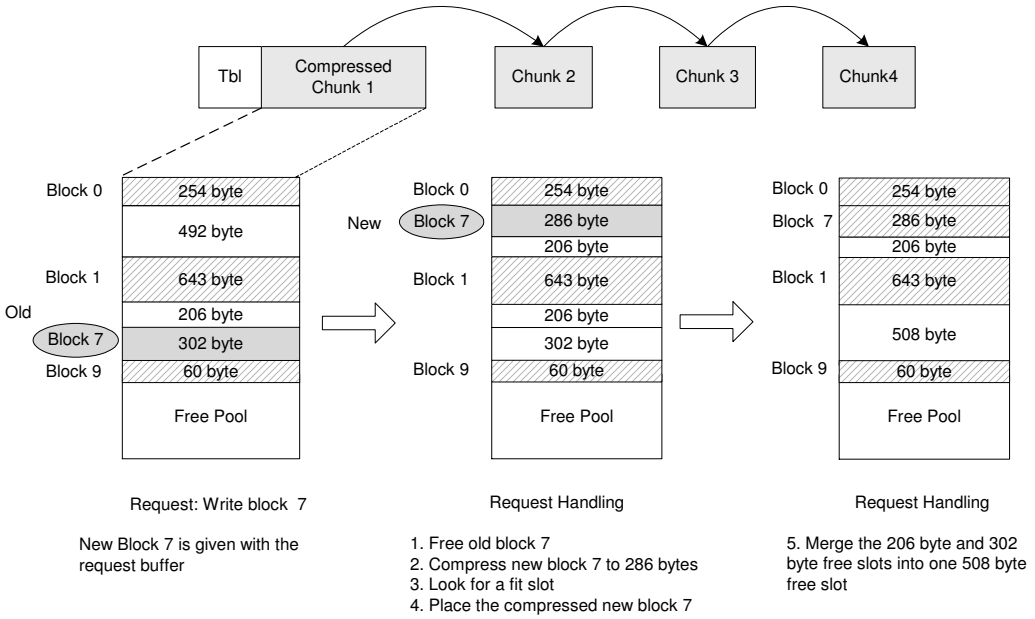

FIGURE 3.10. Compressed blocks in CRAMES device.

Figure 3.10 illustrates the logical structure and request handling of a CRAMES device. The memory space in a CRAMES device consists of several virtually contiguous memory chunks. Each chunk is divided into blocks with potentially different sizes. Shaded areas represent occupied areas and white areas represent free areas. Upon initialization, a CRAMES device requests a small contiguous memory chunk in the kernel virtual memory space. It requests additional memory chunks as system memory requirements grow. These compressed memory chunks are maintained in a linked list. Each chunk cannot be divided uniformly because the sizes of compressed blocks may differ due to the dependence of compression ratio on the specific data in each block. When all compressed blocks in a compressed chunk are free, CRAMES frees the

entire chunk to the system. Therefore, the size of a CRAMES device dynamically increases and decreases during operation, adapting to the data memory requirements of the currently running applications. This dynamic adjustment allows CRAMES to support applications that would not be capable of running without the technique but avoids performance and power consumption penalties for (sets of) applications that are capable of running without data compression. When a CRAMES device receives a read request for a block, it locates the block using its mapping table, decompresses it, and copies the original data to the request buffer. When it receives a write request for a block, it locates the block, determines whether the old block with the same index may be discarded, compresses the new block, and places it at a position decided by the CRAMES memory management system.

3.5. Evaluation and Experimental Results

This section presents performance and power consumption measurements of applications running on a Sharp Zaurus SL-5600 PDA, with and without CRAMES. This battery-powered embedded system runs an embedded version of Linux called Embedix [**6**]. It has a 400 MHz Intel XScale PXA250 processor [**36**], 32 MB of flash memory, and 32 MB of RAM. We replaced the SL-5600's battery with an Agilent E3611A direct current power supply. Current was computed by measuring the voltage across a 5 W, 250 mΩ, Ohmite Lo-Mite 15FR025 molded silicone wire element resistor in series with the power supply. Note that this resistor was designed for current sensing applications. Voltage measurements were taken using a National

TABLE 3.4. Performance, Power Consumption, and Energy Consumption for Filesystem Experiments

Benchmark	Time (s)		Power (W)		Energy (J)	
	without /	w. CRAMES	without /	w. CRAMES	without /	w. CRAMES
mke2fs	0.0451	0.0454	1.58	1.48	0.0713	0.0670
cp small file	0.0509	0.0469	1.57	1.63	0.0802	0.0763
cp large file	0.1688	0.2339	1.50	1.43	0.2536	0.3346
rm small file	0.0456	0.0500	1.49	1.48	0.0678	0.0738
rm large file	0.0447	0.0455	1.50	1.49	0.0669	0.0677
pack tree	3.8130	4.9336	1.92	1.92	7.3134	9.4965
unpack	0.2761	0.3109	1.43	1.47	0.3937	0.4571
cp tree	0.4597	0.4555	1.71	1.39	0.7844	0.6327
rm tree	0.2991	0.3071	1.46	1.48	0.4368	0.4560
find	0.2968	0.2893	1.50	1.39	0.4465	0.4025

Instruments 6034E data acquisition board attached to the PCI bus of a host workstation running Linux.

3.5.1. Using CRAMES for Filesystem on Zaurus

CRAMES was used to create a compressed RAM device for the EXT2 filesystem on a Zaurus SL-5600. We compared the execution time and energy consumption of this device with that of the EXT2 filesystem on a common RAM disk. For these comparisons we used common file operations such as `mke2fs`, `cp`, `rm`, and etc. as shown in Table 3.4. Note that each benchmark was executed five times; the average results are reported. We observed an average compression ratio of 63% for the CRAMES device. In addition, Table 3.4 illustrates that the increases in execution time and energy consumption were small: on average 8.4% and 5.2%,

respectively. These results indicate that CRAMES is capable of reducing the RAM requirement for embedded system filesystem storage with only small performance and energy penalties.

3.5.2. Using CRAMES for Swapping on Zaurus

In order to evaluate the effectiveness of CRAMES when it is used for swapping, we used two experimental setups. For the first set of experiments, we did not change the RAM size on Zaurus. We found that sets of applications that required too much RAM to execute on the unmodified Zaurus could execute smoothly when CRAMES was used. In these experiments, we also identified the performance, power consumption, and energy consumption impacts of CRAMES on existing applications that were able to run without compression. Our results show that the penalties for such applications were negligible.

For the second set of experiments, we did not introduce new applications to the system; instead, we artificially reduced the system RAM to different sizes to prove that with CRAMES the system could still support existing applications with small performance, power consumption, and energy consumption penalties, while without CRAMES these applications were either unable to execute or ran only with extreme performance degradation and instability, i.e., no response or system crash.

3.5.2.1. *Evaluating CRAMES with the original RAM size*

The benchmarks used to evaluate CRAMES contain three applications from the Mediabench benchmark suite [**37**], one matrix multiplication program with different matrix sizes, ten common GUI applications provided with Qtopia [**38**] for Zaurus PDAs, and combinations of

these applications running simultaneously. In order to consistently evaluate the behavior of an unmodified PDA and a PDA using CRAMES when running interactive applications, we wrote software to monitor user input and repeat it with identical timing characteristics. This technique replaces the OS touchscreen device with a *named pipe* or *FIFO* (first in first out) controlled by a program that reads from the raw touchscreen. It stores user input events and timing information in a file. The contents of this file are later replayed to the touchscreen device in order to simulate identical user interaction. This allows us to consistently reproduce user input, enabling the consistent use of benchmarks containing GUIs.

Benchmarks applications were tested with and without CRAMES. Each application was executed five times; the average results are reported. Applications can be grouped into three categories: (1) applications with small working data sets, i.e., adpcm, mpeg2, jpeg, Hancom Word, Hancom Sheet, and calculator; (2) applications with working data sets nearly as large as physical memory, but still (barely) able to run without CRAMES, i.e., 500 by 500 matrix multiplication, Opera, Primtest, and Quasar; and (3) applications with working data sets too large to fit into physical memory, i.e., simultaneously running Qpera and Quasar as well as simultaneously running large matrix multiplication and Media Player. Table 3.5 and Figure 3.11 show that, for the first and second category, using CRAMES seldom results in any performance, power, or energy penalties because no or few pages are swapped out. For the third category, it is not possible to compare with the performance, power, and energy of the original embedded system. The applications in this category simply cannot run without using CRAMES.

TABLE 3.5. Performance, Power Consumption, Energy Consumption, and Compression Ratio for Swapping Experiments

Num.	Application	Description	Size (KB)		Time (s)		Power (W)		Energy (J)		Swap	Comp
			Data	Code	w.o.	CRAMES	w.o.	CRAMES	w.o.	CRAMES	(bytes)	ratio
1	Adpcm	MB: Speech compression	24	4	1.31	1.30	2.11	2.09	2.75	2.72	0	n.a.
2	Mpeg2	MB: Video CODEC	416	48	76.60	76.76	2.42	2.43	185.72	186.55	0	n.a.
3	Jpeg	MB: Image encoding	176	72	0.22	0.21	2.14	2.02	0.48	0.42	0	n.a.
4	Address Book	GUI: Address book	32	8	30.63	30.61	1.51	1.59	46.14	48.72	0	n.a.
5	Hancom Word	GUI: Office tool	32	8	32.97	32.98	1.54	1.55	50.70	51.26	0	n.a.
6	Hancom Sheet	GUI: Office tool	32	8	28.85	28.75	1.69	1.72	48.77	49.55	0	n.a.
7	Calculator	GUI: Calculator	32	8	33.19	33.21	1.59	1.54	52.89	51.07	0	n.a.
8	Asteroids	GUI: Fighting game	1,004	64	30.79	30.81	1.72	1.79	53.01	55.28	0	n.a.
9	Snake	GUI: Game	692	32	31.75	31.73	1.54	1.53	48.76	48.69	0	n.a.
10	Go	GUI: Chess game	508	80	31.02	31.02	1.52	1.51	47.02	46.79	0	n.a.
11	Matrix (500)	Matrix Multiplication	2,948	4	43.02	41.41	2.28	2.27	98.27	94.07	129,461	0.33
12	Opera Browser	GUI: Web browser	1,728	3,972	29.65	29.65	1.78	1.69	52.86	50.16	454,585	0.40
13	Primtest	GUI: Java Multi-thread	2,848	1,364	27.77	27.79	2.06	2.11	57.30	58.52	497,593	0.39
14	Quasar	GUI: Java Multi-thread	4,192	1,364	47.16	47.10	2.01	2.03	94.63	95.43	449,224	0.43
15	Opera & Quasar	GUI & GUI combination	6,104	5,336	n.a.	47.12	n.a.	2.09	n.a.	98.68	992,561	0.40
16	Matrix (800) & Media Player	Batch & GUI combination	11,600	168	n.a.	83.77	n.a.	3.27	n.a.	273.55	832,642	0.34

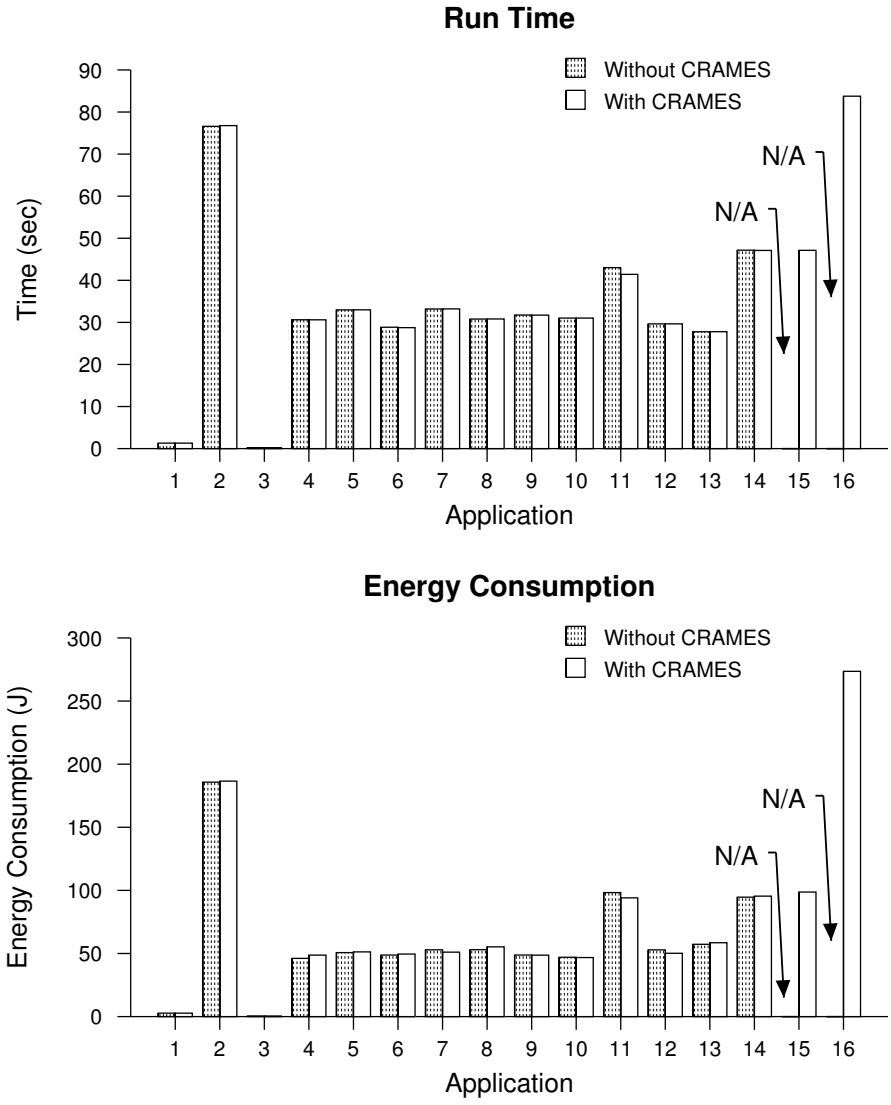

FIGURE 3.11. Performance and energy impact of using CRAMES for swapping.

These results indicate that CRAMES has the ability to increase available memory, thereby allowing an embedded system to support applications that would otherwise be unable to run.

Moreover, its impact on the performance and energy consumption of applications capable of running on the original system is negligible.

3.5.2.2. *Evaluating CRAMES by constraining the memory size*

One of the biggest contribution of CRAMES is that it allows applications to run on a system with less RAM. In order to evaluate the impact of using CRAMES to reduce physical RAM, we artificially constrained the memory size of Zaurus. With reduced physical RAM, we measured and compared the run times, power consumptions, and energy consumptions of the four batch benchmarks, i.e., three applications from MediaBench and one matrix multiplication application. For these experiments we didn't use the GUI applications and the user interface playback system described in Section 3.5.2.1. When physical RAM is reduced to a very low value, e.g., 20 MB, the applications suffer severe performance degradation due to kernel page reclamation. Therefore, the timing information is no longer accurate and the recorded user actions can no longer correctly control the applications. Moreover, Zaurus is designed to pop up warning windows indicating that the system is dangerously low on memory, interfering with the playback of GUI interaction traces. As a result, GUI applications do not allow fair performance comparisons and thus cannot be used as benchmarks for the experiments described in this section.

To constrain the size of system RAM on Zaurus, we used a simple kernel module that permanently reserved a certain amount of physical memory. The memory taken by a kernel module cannot be swapped out [**26**] and therefore is not compressed by CRAMES. This guarantees the fairness of our comparison.

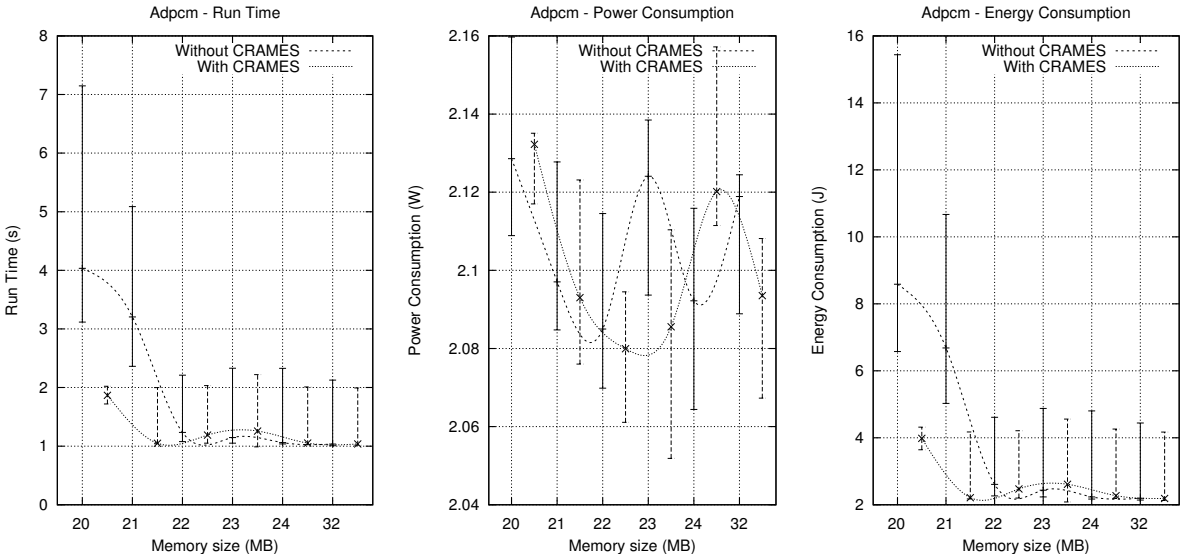

FIGURE 3.12. Performance and energy consumption of adpcm.

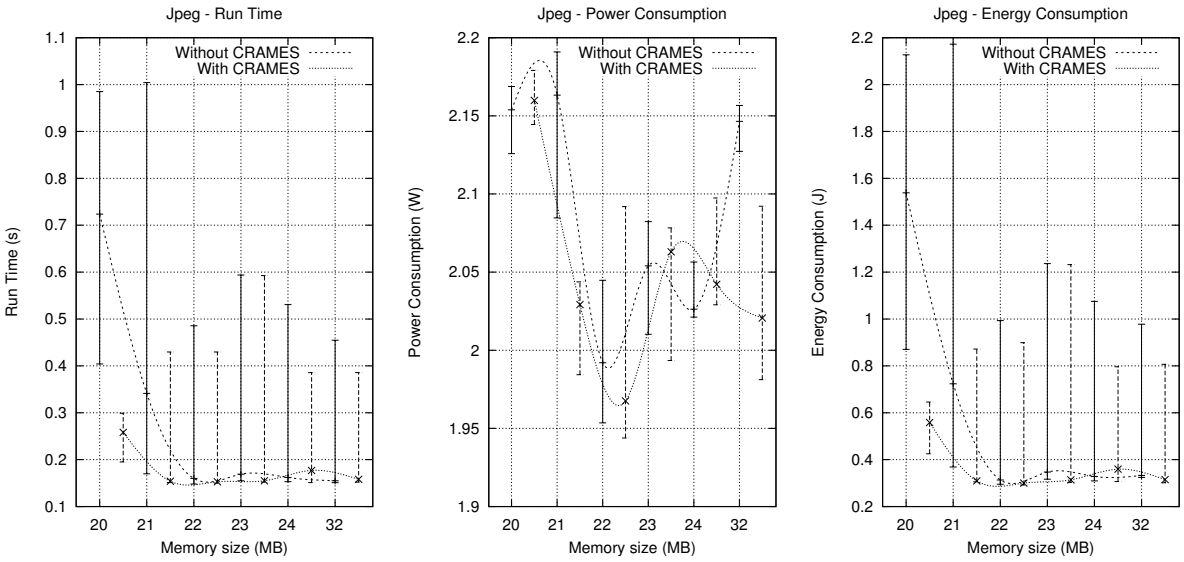

FIGURE 3.13. Performance and energy consumption of jpeg.

Figure 3.12, 3.13, 3.14, and 3.15 show the performance and energy consumptions of benchmarks adpcm, jpeg, mpeg2, and matrix multiplication. In our experiments, each benchmark was executed five times; the average results are reported. The vertical bars in these figures are

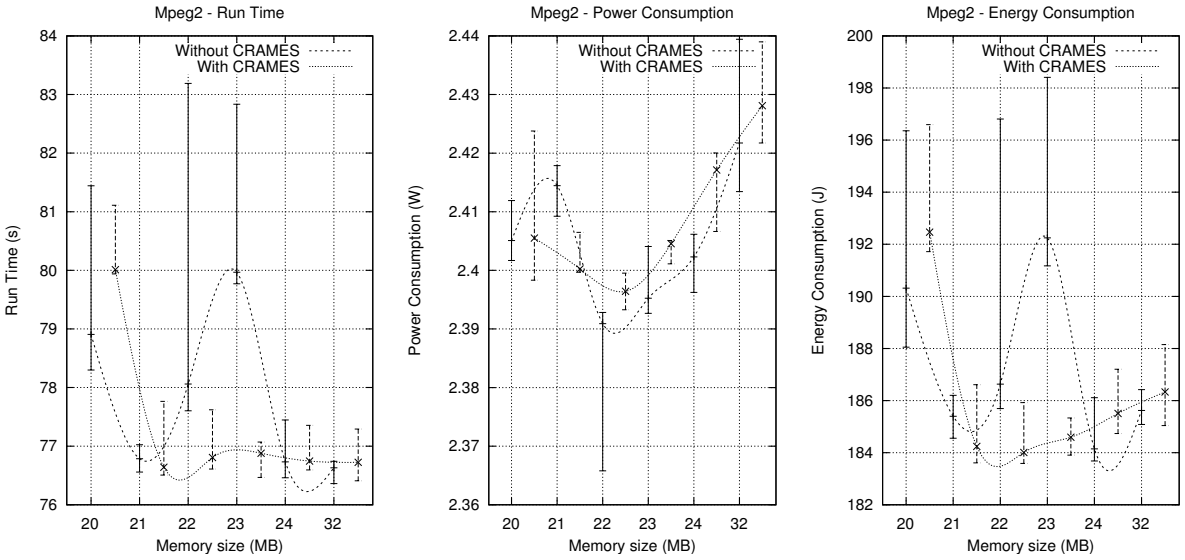

FIGURE 3.14. Performance and energy consumption of mpeg2.

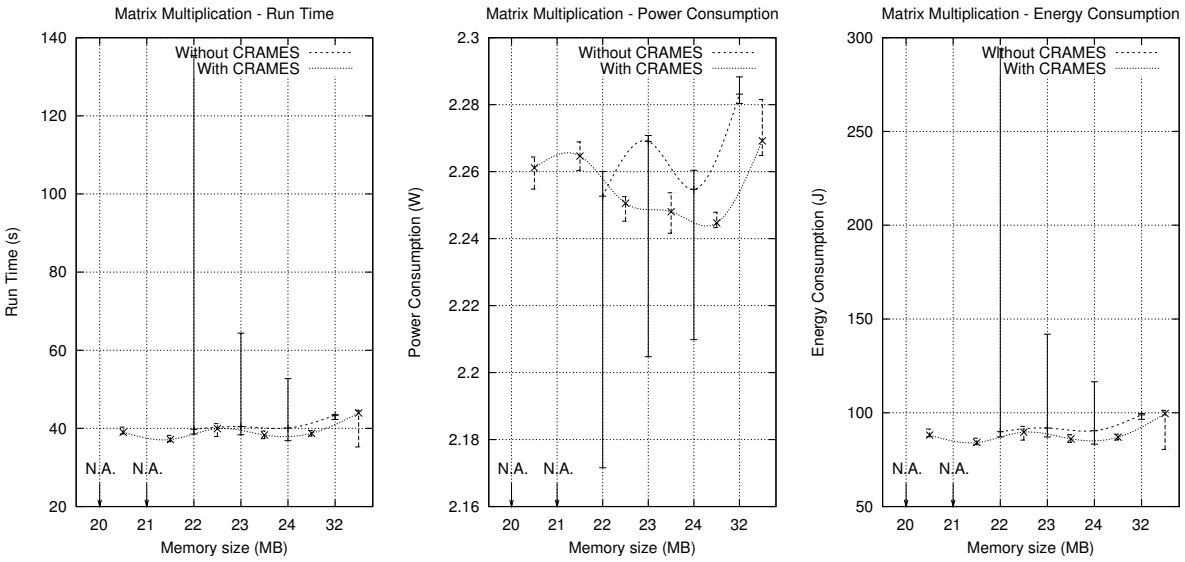

FIGURE 3.15. Performance and energy consumption of matrix multiplication.

the range of run times, power consumptions, and energy consumptions for each benchmark, while the dotted curves represent the median values. For example, when system RAM is 20 MB and no compression is used, the shortest, longest, and median run time of benchmark adpcm

are 3.12 seconds, 7.15 seconds, and 4.03 seconds, respectively. The dotted curves illustrates the trend of execution time, power consumption, and energy consumption of each benchmark under different system settings.

Figure 3.12, 3.13, and 3.14 show that when the system RAM is set to 20 MB or 21 MB, CRAMES dramatically improves the performance of benchmarks adpcm, jpeg, and mpeg2. More specifically, compared to the base case in which system RAM is 32 MB and no compression is used, when system RAM is reduced to 20 MB and CRAMES is not present, these three benchmarks exhibit an average performance penalty of 160.3% and a worst-case performance penalty of 268.7%. In contrast, when CRAMES is present, the average and worst-case performance penalties of these benchmarks are 9.5% and 29%, respectively. As explained in Section 3.3.3, a significant portion of RAM on Zaurus is used as a battery-backed RAM disk. Therefore only 14 MB of memory are available to the applications and system background processes. When the memory size is reduced by 12 MB or 11 MB, the system is dangerously low on memory. Without compression, the kernel must rely on page reclamation from buffer caches to get enough memory to run the applications. This process may take a very long time and therefore introduces large performance and energy consumption penalties. However, when CRAMES is present, the kernel may provide more memory for applications via compression and therefore maintains good performance and low energy consumption. When the system RAM is higher than 24 MB, the performance of the applications with or without compression is very close. i.e., no latency is observed.

Figure 3.15 shows that, without CRAMES, 512 by 512 matrix multiplication simply would not execute when the system RAM is set to 20 MB or 21 MB. However, when CRAMES is present, it was able to execute. More interestingly, we observe that when CRAMES is present the performance actually improves by 8% on average compared to the base case for which the system has 32MB of memory and CRAMES is not present. This phenomenon can be explained as follows. Unlike the other three applications, the memory requirement of 512 by 512 matrix multiplication exceeds the available memory when the system has 32 MB of RAM and no compression. Without compression, the kernel must reclaim memory from buffer caches by either using clean pages or evicting dirty pages. However, CRAMES starts compressing data as soon as the free memory available in the system becomes dangerously low. Therefore, before the matrix multiplication program starts, the free memory in the system has already increased due to pre-compression. When the application starts running, the kernel needs to do little reclamation, resulting in improved performance. CRAMES does not improve the performance of the other three applications, i.e., Adpcm, Jpeg, and Mpeg2. Although CRAMES responds to reduced RAM by pre-compressing pages, the performance of these applications is not affected much because their memory requirements are smaller, i.e., little RAM reclamation is needed even without CRAMES.

We also ruled out another possible cause of the performance improvement: cache effects. It is conceivable that, on a system with a physically-tagged cache, CRAMES might shuffle the matrix pages within physical RAM, reducing conflict misses on some architectures. However,

the XScale PXA250 processor used in the Zaurus has a data cache that is virtually addressed and virtually tagged [**36**].

3.6. Conclusions

In this chapter, we have presented a software-based RAM compression technique, named CRAMES, for use in low-power, disk-less embedded systems. CRAMES has been implemented as a Linux kernel module and evaluated on a typical disk-less embedded system with a representative set of batch as well as GUI applications. Experimental results indicate that CRAMES is capable of doubling the amount of memory available to applications, with negligible performance and energy consumption penalties for existing applications, without adding RAM or hardware to the target system. When system RAM is reduced from 32 MB to as low as 20 MB, CRAMES allows all batch benchmarks to execute with on average 9.5% increase in execution time. However, without CRAMES these benchmarks either cannot execute, become unstable, or suffer from extreme performance and energy consumption penalties. In addition to on-line working data sets compression, CRAMES supports in-RAM compression of arbitrary filesystems type. For experiments with the EXT2 filesystem, CRAMES increased available storage by at least 40%, with small performance and energy consumption penalties (on average 8.4% and 5.2%, respectively). We conclude that CRAMES is an efficient software solution to the RAM compression problem for embedded systems in which application memory requirements exceed physical RAM. Moreover, it allows hardware designs to be optimized for the typical memory requirements of applications while also supporting (sets of) applications with larger data sets.

Chapter 4

Towards Higher Performance On-Line Memory Compression

In the previous chapter, we described our software-based memory compression framework CRAMES. Via on-line memory compression, CRAMES is capable of increasing functionality of embedded systems without changes to hardware or applications by making better use of physical memory. CRAMES takes advantage of an operating system's virtual memory infrastructure by storing swapped-out pages in compressed format. It dynamically adjusts the size of the compressed RAM area, protecting applications capable of running without it from performance or energy consumption penalties. CRAMES has been implemented as a loadable module for the Linux kernel and evaluated on a battery-powered Sharp Zaurus PDA. It is currently being evaluated on a smartphone prototype system for a major embedded device manufacturer.

Recall that in order to evaluate the effectiveness of CRAMES on Zaurus, we used two experimental setups and performed extensive experiments. For the first set of experiments, we did not change the RAM size on Zaurus. We found that CRAMES was capable of doubling the

amount of RAM available to applications; sets of applications that required too much RAM to execute on the unmodified Zaurus could execute smoothly when CRAMES was used. Moreover, the performance, power consumption, and energy consumption impacts of CRAMES were negligible for existing applications that were able to run without compression.

For the second set of experiments, we did not introduce new applications to the system; instead, we artificially reduced the system RAM to different sizes. We found that with CRAMES, the system could still support existing applications while without CRAMES these applications were either unable to execute or ran only with extreme performance degradation and instability, i.e., no response or system crash. Compared with the unmodified system, using CRAMES to cut physical memory to 40% increased application execution time by 9.5% on average and by 29% in the worst case. This performance penalty may prevent CRAMES from being used for many embedded systems.

In this chapter, we describe two techniques to further improve the performance of on-line software-based memory compression: (1) a very fast, high-quality compression algorithm for working data set pages; and (2) an adaptive compressed memory management technique that predictively allocates memory for compressed data. In comparison with algorithms commonly used in on-line memory compression, our new compression algorithm has a competitive compression ratio but is twice as fast. The adaptive memory management scheme effectively responds to the predicted needs of applications and prevents on-line memory compression deadlock, permitting reliable and efficient compression for a wide range of applications.

The rest of this chapter is organized as follows. Section 4.1 summarizes existing RAM compression algorithms in literature. Section 4.2 presents the proposed software-based memory compression algorithm. Section 4.3 describes our method of adaptively managing the uncompressed and compressed regions of memory. Section 4.4 presents the experimental set-up, describes the workloads, and explains the experimental results in detail. Finally, Section 4.5 summarizes this chapter.

4.1. Existing Memory Compression Algorithms

Compression techniques, both lossy and lossless, are widely used in all fields of information processing. The compression ratio and compression/decompression speed vary greatly, depending on the technique used and the type of data which is acted upon. On-line memory compression apparently requires lossless compression algorithms. Unfortunately, many existing algorithms are not suitable for this application. There are three main reasons for this. First, the algorithm must perform well operating on data blocks of only a few kilobytes in size. Existing algorithms are generally only suitable for larger amounts of data as they tend to produce significant data expansion (up to 150%) during the early stages of the compression process, and hence perform poorly on small data blocks. Second, to minimize performance penalty, the compression and decompression speed must be extremely fast. Many existing algorithms like gzip and bzip2 provide very good compression ratio but are too slow to be used in on-line memory compression. The last point is, for some techniques that require hardware-based compression and decompression unit, most current algorithms tend not to perform as well as could

be hoped from the view of both throughput and latency, although they are capable of hardware implementation.

Rizzo [**20**] tried to analyze the frequent patterns that present in memory data. He concluded that zero-valued data are the most frequent and that the next frequent symbols are much more difficult to determine, and very content-dependent. He therefore proposed an software-based algorithm that compresses in-RAM data by only exploiting the high frequency of zero-valued data. In order to improve the operating speed, the algorithm uses W-bits Huffman encoding, where W is the size of a machine-word. In the next section, we will show our analysis of in-RAM data and present other frequent patterns we have identified, in addition to zero-valued data.

Kjelso et al. [**9**] designed the X-Match hardware compression algorithm that maintains a dictionary of data previously seen and attempts to match the current data element with an entry in the dictionary, replacing it with a shorter code referencing the match location. The dictionary is maintained using a move to front strategy to exploit locality in the input data. A data element can have all characters match a dictionary entry (full match), or it can have at least any two of the characters match exactly with a dictionary entry (partial match), with the characters that do not match being transmitted literally. Data elements which do not produce a match are transmitted in full prefixed by a single bit.

IBM's MXT technology [**7**] used an efficient hardware-based main memory compression algorithm by implementing a gate-intensive, parallelized derivative of the Lempel-Ziv (LZ77)

sequential algorithm [**39**]. With this implementation, the uncompressed data block is partitioned into n equal parts, each operated on by an independent compression engine, but with shared dictionaries. Their results show that parallel compressors with cooperatively constructed dictionaries have compression efficiency essentially equivalent to that of the sequential LZ77 method.

Wilson et al. [**17**] presented a software-based algorithm, called WKdm, which also used a small dictionary of recently seen words and attempts to fully or partially match incoming data with an entry in the dictionary. In their algorithm a 32-bit word may not match a dictionary entry, or match only in the upper 22 bits, or match a whole 32-bit pattern. As a special case, the word is first checked to see if it is all zeroes, i.e., matches a full-word zero. Similar to Rizzo's algorithm, the WKdm algorithm exploits the highest frequent pattern (all zero value); and like the X-Match algorithm, the WKdm algorithm attempts to fully or partially match input data with dictionary entries to exploit similarities among words. However, it does not identify and take advantage of other frequent patterns that exhibit in the word itself and therefore does not generate a very satisfactory compression ratio.

4.2. Pattern-Based Partial Match Compression

A fast compression algorithm with good compression ratio is the first step toward a high-performance memory compression system. In Section 3.3.2.2, we presented the experimental evaluation of several existing data compression algorithms. Among those algorithms, LZO appears to be most appropriate for on-line memory compression because of its good all-around

performance. Therefore, the original implementation of CRAMES used the LZO algorithm to compress data in memory. Although LZO is significantly faster than many other general-purpose compression algorithms, e.g., LZW series algorithms, gzip, and bzip2, it is not designed for memory compression and therefore does not fully exploit the regularities of in-RAM data. In addition, LZO requires 64 KB of working memory for compression [1], a significant overhead on many memory-constrained embedded systems. We believe better results are possible for the on-line memory compression application. In this section, we analyze the regularities of in-RAM data and describe a new algorithm, named pattern-based partial match (PBPM), that is extremely fast and well-suited for memory compression.

4.2.1. Overview of PBPM

The PBPM algorithm is designed based on the observation that frequently-encountered data patterns can be encoded with fewer bits to save space. It explores frequent patterns that occur within each word of memory and takes advantage of the similarities among words by keeping a small two-way, hashed, set associative dictionary that is managed with a least-recently used (LRU) replacement policy.

Scanning through the input data a word (32 bits) at a time, PBPM exploits patterns that occur frequently within each word of memory and searches for complete and partial matches with dictionary entries to take advantage of the similarities among words. More specifically, (1) some patterns that are very frequent are encoded using special bit sequences that are much shorter than the original data, (2) patterns that do not fall into the above category and are found

[1] There is also a compression level which requires 8 KB for compression but offers poorer compression ratio.

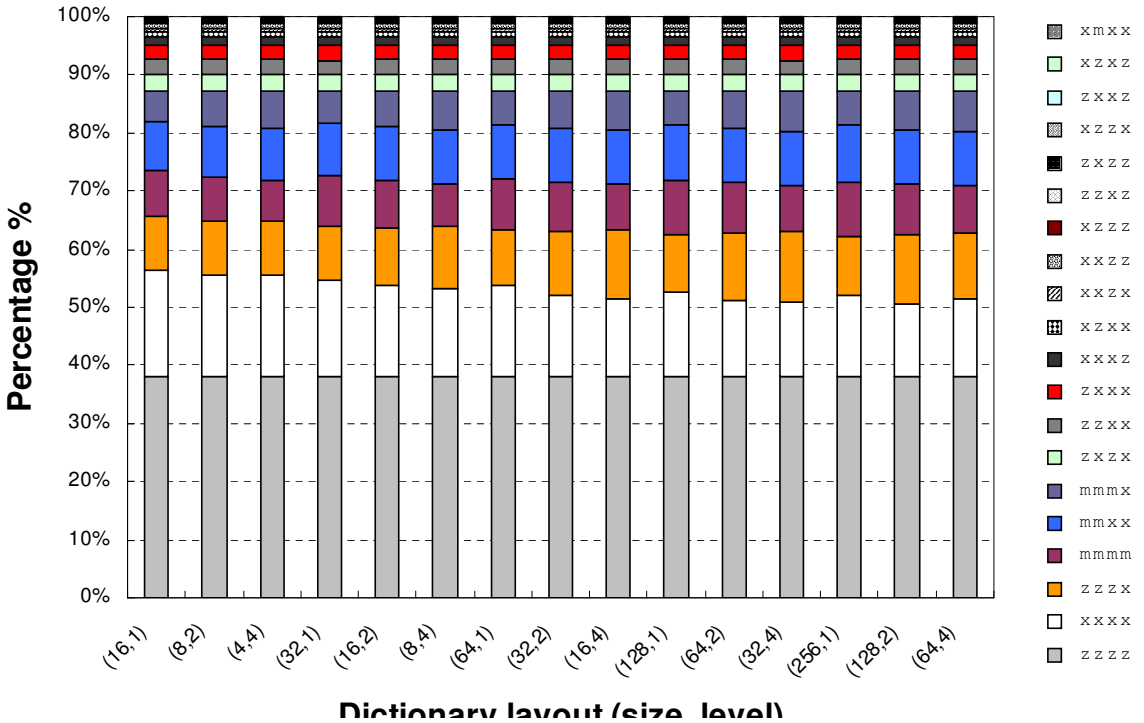

FIGURE 4.1. Frequent pattern histogram.

in a small dictionary are encoded using the index of their location in dictionary, and finally (3) patterns that do not frequently occur and cannot be found in the dictionary are stored in dictionary while the word contents are output.

PBPM has a compression ratio that is competitive with the best compression algorithms of the Lempel-Ziv family [**39**] while exhibiting lower runtime and memory overhead, making it ideal for on-line software memory compression.

4.2.2. In-RAM Data Patterns

Unlike general-purpose algorithms designed for text data, a special-purpose algorithm designed for in-RAM data compression must fully exploit the regularities present in memory.

In-RAM data frequently have certain patterns. For example, pages are usually zero-filled after being allocated. Therefore, runs of zeroes are commonly encountered during memory compression. Numerical values are often small enough to be stored in 4, 8, or 16 bits, but are normally stored in full 32-bit words. Furthermore, numerical values tend to be similar to other values in nearby locations. Likewise, pointers often point to adjacent objects in memory, or are similar to other pointers in nearby locations.

In order to develop a reasonable set of frequent patterns, we experimented with a 64 MB swap data file from a workstation running SuSE Linux 9.0. Various applications were executed to exhaust physical memory and trigger swapping. Figure 4.1 shows the relative frequencies of patterns we evaluated. Below we specify the conventions in describing the data and patterns, as well as the dictionary management scheme we considered.

We consider each 32-bit word (four bytes) as an input, and represent them with four symbols, each of which represents a byte. A 'z' represents a zero byte, an 'x' represents an arbitrary byte, and an 'm' represents a byte that matches with a dictionary entry. Following this convention, 'zzzz' indicates an all-zero word, while 'mmmx' indicates a partial match with one dictionary entry for which only the lowest byte differs.

To allow fast search and update operations, we maintain a hash-mapped dictionary. More specifically, the third byte of a word is hash-mapped to a 256 entry hash table, the contents of which are random indices that are within the range of the dictionary. Based on this hash function, we only need to consider four match patterns: 'mmmm' (full match), 'mmmx' (highest three bytes match), 'mmxx' (highest two bytes match), and 'xmxx' (only the third byte matches). Note

that neither the hash table nor the dictionary need be stored with the compressed data. The hash table is static and the dynamic dictionary is regenerated automatically during decompression.

We experimented with different dictionary sizes/layouts, e.g., 16-entry direct-mapped and 32-entry two-way set associative, etc. A direct hash-mapped dictionary has the advantage of supporting fast search and update: only a single hashing operation and lookup are required per access. However, it has tightly limited memory. For each hash target, only the most recently observed word is remembered; the victim to be replaced is decided entirely by its hash target. In contrast, if a dictionary is maintained with move-to-front strategy, its LRU entry is selected as the victim. Unfortunately, searching in such a dictionary is slow. A set associative dictionary can enjoy the benefits of both LRU replacement and speed. When a search miss followed by a dictionary update occurs, the oldest of the dictionary entries sharing one hash target index is replaced.

As Figure 4.1 illustrates, zero words, ‘zzzz’, are the most frequent compressible pattern (38%), followed by one byte positive sign-extended words ‘zzzx’ (9.3%). ‘zxzx’ has a frequency of 2.8%. Other zero-related patterns are infrequent. As the dictionary size increases, dictionary match (including partial match) frequencies do not increase much. While a set associative dictionary usually generates more matches than a direct hash-mapped dictionary with the same overall size, a four-way set associative dictionary works no better than a two-way set associative dictionary.

TABLE 4.1. Pattern Encoding in PBPM

Code	Pattern	Output	Size (bits)	Frequency
00	zzzz	00	2	38.0%
01	xxxx	01BBBB	34	21.6%
10	mmmm	10bbbb	6	11.2%
1100	zzzx	1100B	12	9.3%
1101	mmxx	1101bbbbBB	24	8.9%
1110	mmmx	1110bbbbB	16	7.7%
1111	zxzx	1111BB	20	3.1%

4.2.3. The PBPM Compression Algorithm

The PBPM compression and decompression algorithms are presented in Algorithm 1. Based on our analysis of frequent patterns for in-RAM data and different dictionary management schemes, we selected the most frequent patterns and the most effective dictionary layouts in PBPM. The patterns and coding schemes are summarized in Table 4.1, which also reports the actual frequency of each pattern observed in our swap data file when other infrequent patterns are ignored. In Algorithm 1 and column 'Output' of Table 4.1, 'B' represents a byte and 'b' represents a bit.

PBPM maintains a small two-way set associative dictionary *DICT*[] of 16 recently-seen words. An incoming word can fully match a dictionary entry, or match only the highest three bytes or two bytes of a dictionary entry. These patterns occurred frequently during swap trace analysis. Although it would be possible to consider non-byte-aligned partial matches [**7**, **17**], we have experimentally determined that byte-aligned partial matches are sufficient to exploit the partial similarities among in-RAM data while permitting more efficient implementation. The

PBPM algorithm compresses and decompresses 32-bit words. The compressor scans through a page (usually 4KB), reads each word, and determines the first of the following criterion the word meets:

(1) Is it a '0000'?

(2) If not, is it a '000x'?

(3) If not, is it a '0x0x'?

If the word does not meet any of these criteria, the compressor checks whether the word fully or partially matches a dictionary entry. If it is a partial match, this word is inserted into the dictionary location indicated by hashing on its third byte. The decision to base hashing on the third byte was made to achieve decent hashing quality with low computational overhead. Note that the victim to be replaced is decided by its age. If there is no match at all, the word is also inserted to the dictionary according to the same replacement policy. Correspondingly, the decompressor reads through the compressed output, decodes the format based on the patterns given in table 4.1, and adds entries to the dictionary upon a partial match or dictionary miss. Therefore, the dictionary can be re-constructed during decompression and does not need to be stored together with the compressed data. The experimental evaluation of our PBPM algorithm is presented in Section 4.4.

4.3. Adaptive Compressed Memory Management

Efficient management of compressed memory regions is critical to the performance and compression ratio of on-line memory compression techniques. In this section, we present an

Algorithm 1 PBPM (a) compression and (b) decompression

```
Require: IN, OUT word stream
Require: TAPE, INDX bit stream
Require: DATA byte stream
for word in range of IN do
    if word = zzzz then
        TAPE ← 00
    else if word = zzzx then
        TAPE ← 1100
        DATA ← B
    else if word = zxzx then
        TAPE ← 1111
        DATA ← BB
    else
        mmmm ← DICT[hash(word)]
        if word = mmmm then
            TAPE ← 10
            INDX ← bbbb
        else if word = mmmx then
            TAPE ← 1110
            INDX ← bbbb
            DATA ← B
            Insert word to DICT
        else if word = mmxx then
            TAPE ← 1101
            INDX ← bbbb
            DATA ← BB
            Insert word to DICT
        else
            TAPE ← 01
            DATA ← BBBB
            Insert word to DICT
        end if
    end if
end for
OUT ← Pack(TAPE,DATA,INDX)
```

```
Require: IN, OUT word stream
Require: TAPE, INDX bit stream
Require: DATA byte stream
Unpack(OUT)
for code in range of TAPE do
    if code = 00 then
        OUT ← zzzz
    else if code = 1100 then
        B ← DATA
        OUT ← zzzB
    else if code = 1111 then
        BB ← DATA
        OUT ← zBzB
    else if code = 10 then
        bbbb ← INDX
        OUT ← DICT[bbbb]
    else if code = 1110 then
        bbbb ← INDX
        mmmm ← DICT[bbbb]
        B ← DATA
        OUT ← mmmB
        Insert mmmB to DICT
    else if code = 1101 then
        bbbb ← INDX
        mmmm ← DICT[bbbb]
        BB ← DATA
        OUT ← mmBB
        Insert mmBB to DICT
    else if code = 01 then
        BBBB ← DATA
        OUT ← BBBB
        Insert BBBB to DICT
    end if
end for
```

adaptive memory management scheme that predictively allocates memory for compressed data to improve the effectiveness of memory compression and preemptively avoid memory exhaustion. Experimental results show that our new pre-allocation method is able to further increase

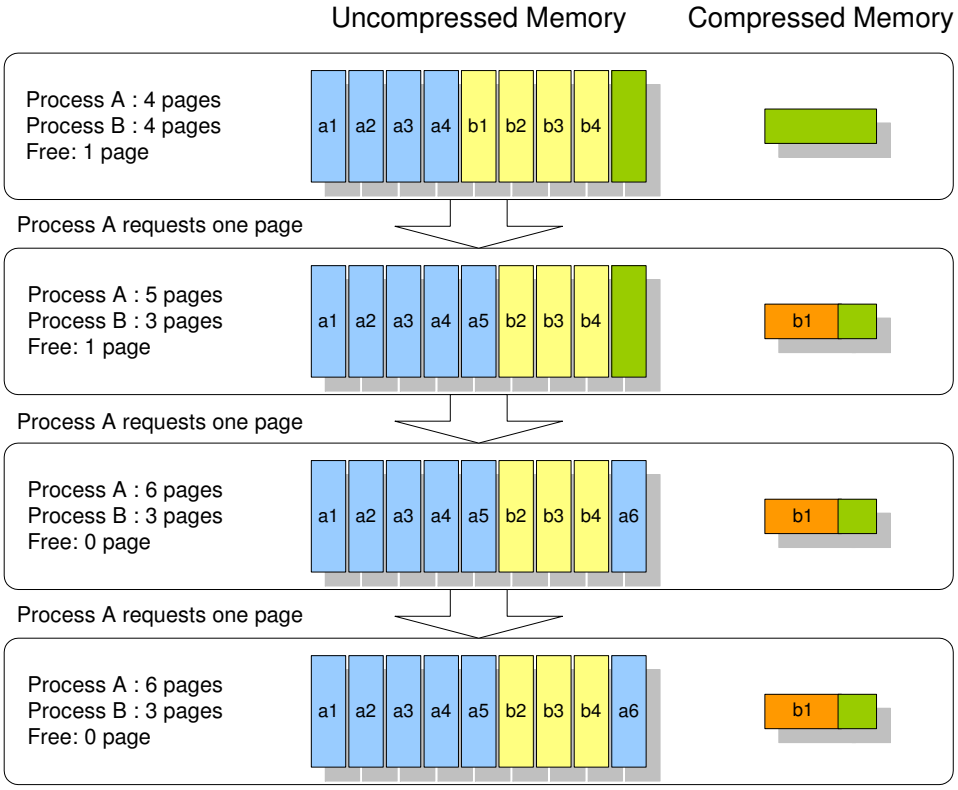

FIGURE 4.2. On-Line memory compression deadlock.

available memory to applications by up to 13% compared to the same system without pre-allocation.

As described in Section 3.3, CRAMES compresses the swapped-out data a page (4,096 bytes) at a time and stores them in a special compressed RAM device. Upon initialization, the compressed RAM device only requests a small memory chunk (usually 64 KB); as system memory requirements grow, CRAMES requests additional memory chunks. The compressed RAM device also dynamically decreases its size when compressed pages are freed. A compressed page may be freed under two circumstances: (1) the page only belongs to one running process

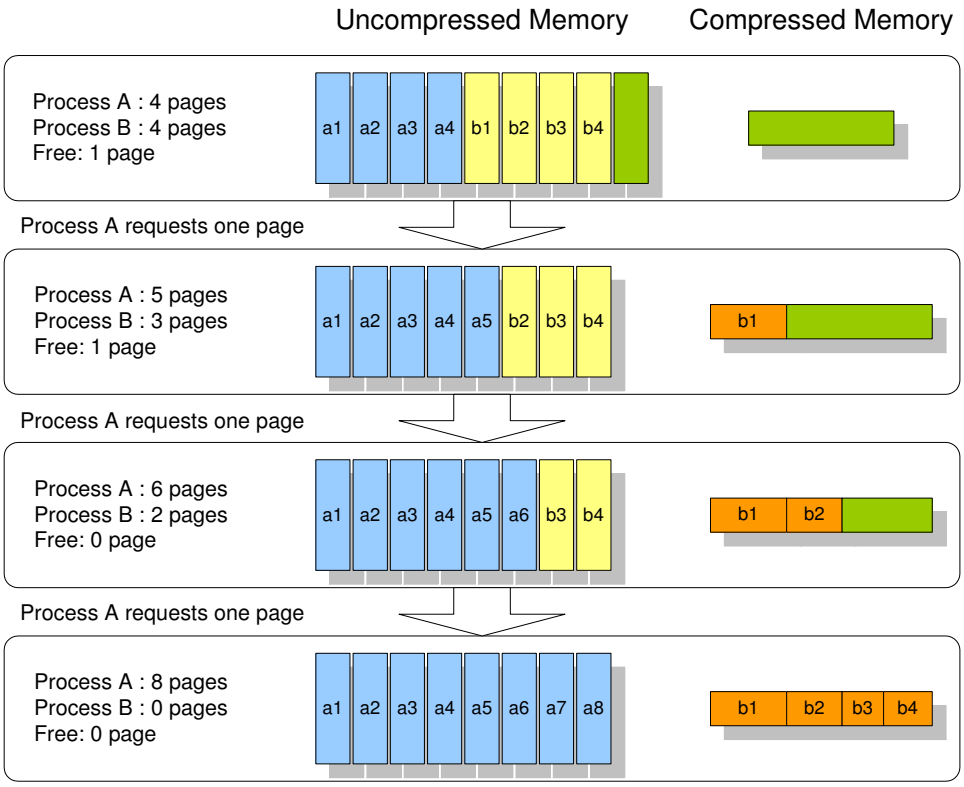

FIGURE 4.3. Deadlock avoided.

and is swapped in by the kernel or (2) the page belongs to a terminated process and the kernel attempts to overwrite this page with a newly swapped out page. When all compressed blocks in a compressed chunk are free, CRAMES frees the entire chunk to the system. Therefore, the size of a compressed RAM device dynamically adjusts during operation, adapting to the data memory requirements of the currently running applications.

The above dynamic memory allocation strategy works well when the system is not under extreme memory pressure. However, it suffers from a performance penalty when the system is dangerously low on memory: applications and CRAMES compete for remaining physical memory and there is no guarantee that an allocation request will be satisfied. If physical memory

is nearly exhausted, an application may be unable to allocate additional memory. If CRAMES were able to allocate even a page of physical memory for its compressed memory region, it would be able to swap out (on average) two pages, allowing the application to proceed. However, CRAMES is in contention with the application, resulting in a scenario we call *on-line memory compression deadlock*.

We explain the on-line memory compression deadlock with the following example. In Figure 4.3, assuming in an embedded system with physical RAM of ten page frames, process A and process B each has four pages in the uncompressed area. CRAMES allocates one page from the kernel for future compression usage, leaving only one free uncompressed page available to processes A and B. (1) The working data set of process A starts to grow and it first requests one more page. Since the free available memory in the system is now only one page, kernel starts swapping and page `b1` from process B gets swapped out and is compressed by CRAMES. After compression, the size of `b1` reduced to 80% of its original size. (2) Process A requests one more page. Kernel continues its attempt to swap out pages from process B. Unfortunately, since the compression ratio of the last swapped-out page `b1` was too high, none of the pages of process B can fit into the compressed area after compression. Therefore, kernel allocates the last free page in the uncompressed area to process A. (3) Process A requests one more page. If CRAMES is able to allocate even only one page from the uncompressed area, it could possibly compress multiple pages and break the contention. However at this moment no more free page is available because process A previously got the last free page. This results in a deadlock scenario where

the allocation request from process A is in contention with allocation request from CRAMES and none of the requests will be successful.

To avoid on-line memory compression deadlock, a compressed RAM device needs to be predictive during its requests for additional memory, i.e., it cannot wait until no existing chunks can allocate a fit slot for the incoming data. This subtle issue comprises a difference between our technique and compressed caching as well as swap compression, in which hard drives serve as backing store to which pages can be moved as soon as (or even earlier than) the compressed area is stuffed, so that the compressed memory is always available to applications.

We propose the following scheme to prevent on-line memory compression deadlock. CRAMES monitors the compressed area utilization and requests the allocation of a new memory chunk based on the saturation of current memory chunks. When the total amount of memory in the compressed area is above a predefined *fill ratio*, CRAMES requests a new chunk from kernel. This request may also be denied if the system memory is dangerously low. However, even if this first request is denied, subsequent invocations of CRAMES will generate additional requests. After low-memory conditions cause applications to swap out pages to the compressed RAM device, more memory will be available and the preemptive compressed RAM device allocation requests will finally be successful.

As illustrated in Figure 4.3, for the previous example, after the first page `b1` from process B is swapped out and compressed, CRAMES should realize that the total compressed area is 80% full and should request more pages from the kernel immediately. Since this allocation request occurs before process A, the last free page in the uncompressed area is guaranteed to be

allocated to CRAMES. Afterwards, when process A requests more memory, CRAMES would be able to compress the rest of the pages from process B because the available memory in the compressed area is now increased, and thus all the subsequent allocation requests from process A are successful.

We have experimented with different fill ratios and found that 7/8 is sufficient to permit successful requests of compressed RAM device memory allocation in all tested applications. This ratio also results in little memory waste. Evaluation of this method on a portable embedded system is presented in Section 4.4.2.

4.4. Evaluation and Experimental Results

In this section, we describe the evaluation methodology and results of the techniques proposed for high-performance on-line memory compression. More specifically, the following questions are experimentally evaluated:

(1) Does PBPM provide a competitve compression ratio yet have even lower performance costs than existing algorithms?

(2) Does adaptive memory management enable CRAMES to provide more memory to applications when system RAM is tightly constrained?

(3) What is the overall performance of CRAMES using PBPM and adaptive memory management?

To evaluate the proposed techniques, we used a Sharp Zaurus SL-5600 PDA. This battery-powered embedded system runs an embedded version of Linux, has a 400 MHz Intel XScale

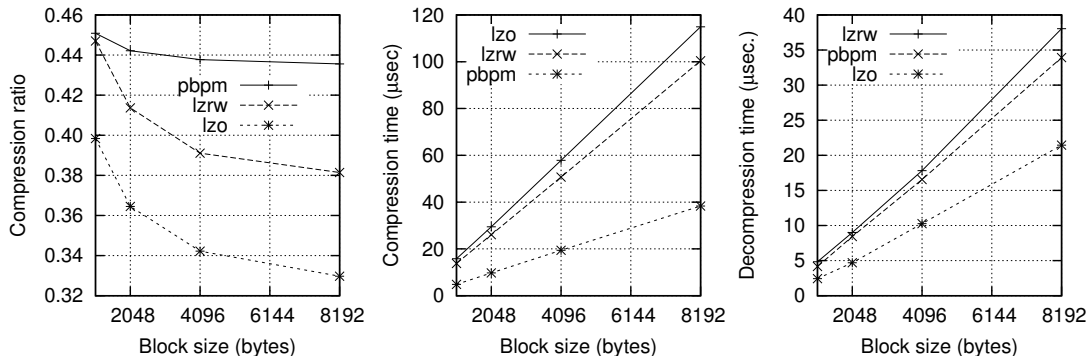

FIGURE 4.4. Compression ratios and speeds of PBPM, LZO, and LZRW.

PXA250 processor, 32 MB of flash memory, and 32 MB of RAM. In our current system configuration, 12 MB of RAM are used for uncompressed, battery-backed filesystem storage and 20 MB are available to kernel and user applications.

4.4.1. Quality and Speed of the PBPM Algorithm

We evaluated the compression ratio and speed of the PBPM algorithm compared to two other compression algorithms that have been used for on-line memory compression: LZO and LZRW. Figure 4.4.1 illustrates the compression ratios (compressed block size divided by original block size) and execution times of evaluated algorithms. For these comparisons, the source file for compression is the swap data file (divided into uniform-sized blocks) used to identify the frequent patterns in memory. The evaluation was performed on a Linux Workstation with a 2.40 GHz Intel Pentium 4 processor. Note that OS-controlled on-line memory compression is a symmetric application, i.e., a memory page is decompressed exactly once every time it is compressed. Therefore, the overall, symmetric, performance of a compression algorithm is the critical performance metric. Overall, PBPM achieves a 200% speedup over LZO and LZRW.

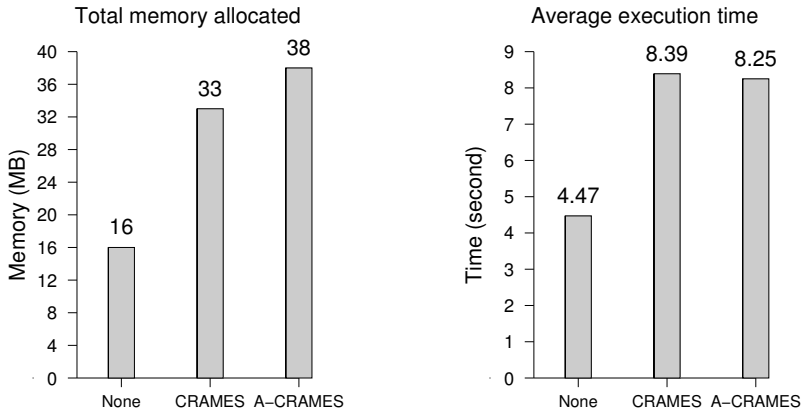

FIGURE 4.5. Performance of A-CRAMES.

Yet the compression ratio achieved by PBPM is competitive to that of LZO and LZRW. We believe that PBPM is especially suitable for on-line memory compression because of its extremely fast symmetric compression and good compression ratio.

4.4.2. Effectiveness of Adaptive Memory Management

In order to determine the effectiveness of adaptive memory management in providing more memory to applications under significant memory pressure, we designed the following experiments. We wrote a 'memeater' program that continuously requests 1 MB of memory at a time and fills the allocated memory with random numbers (including zero runs with similar frequency to that observed in real swap traces), until an allocation request fails. Memeater was then executed on Zaurus under three different system settings: without using CRAMES (none), using CRAMES without adaptive memory management (CRAMES), and using CRAMES with adaptive memory management (A-CRAMES). Figure 4.4.2 presents the total memory allocated and average execution times under the three system settings (memeater was executed multiple

TABLE 4.2. Overall Performance of CRAMES

RAM	Adpcm			Jpeg			Mpeg2			Matrix Mul.		
(MB)	w.o.	LZO	PBPM	w.o.	LZO	PBPM	w.o.	LZO	PBPM	w.o.	LZO	PBPM
Execution Time (s)												
8	4.83	1.69	1.43	0.71	0.26	0.23	79.35	80.30	77.96	n.a	39.26	38.68
9	3.69	1.35	1.26	0.44	0.21	0.21	76.80	76.83	74.04	n.a	37.40	38.24
10	1.41	1.34	1.36	0.23	0.21	0.21	79.06	76.93	75.32	59.11	39.56	37.18
11	1.37	1.40	1.40	0.26	0.25	0.21	80.57	76.81	76.83	44.44	38.42	42.65
12	1.37	1.31	1.32	0.24	0.21	0.19	76.79	76.94	76.95	41.72	38.73	43.96
20	1.31	1.30	1.30	0.23	0.21	0.22	76.60	76.77	76.76	43.02	41.41	42.97
Power Consumption (W)												
8	2.13	2.13	2.13	2.15	2.16	2.15	2.41	2.41	2.51	n.a	2.26	2.29
9	2.10	2.10	2.13	2.15	2.02	2.07	2.41	2.40	2.50	n.a	2.26	2.29
10	2.09	2.10	2.09	2.00	1.99	2.04	2.39	2.40	2.48	2.24	2.25	2.29
11	2.12	2.09	2.13	2.05	2.04	2.07	2.40	2.40	2.50	2.26	2.25	2.29
12	2.09	2.13	2.11	2.03	2.05	2.10	2.40	2.41	2.55	2.25	2.25	2.29
20	2.11	2.09	2.18	2.15	2.02	2.24	2.42	2.43	2.57	2.28	2.27	2.29
Energy Consumption (J)												
8	10.34	3.60	3.04	1.51	0.56	0.49	190.99	193.42	195.71	n.a	88.74	88.62
9	7.75	2.84	2.68	0.94	0.42	0.43	185.38	184.55	185.10	n.a	84.70	87.64
10	2.94	2.79	2.85	0.47	0.42	0.42	188.62	184.34	186.42	131.05	88.99	85.01
11	2.89	2.93	2.97	0.54	0.52	0.44	193.10	184.69	191.94	100.01	86.38	97.79
12	2.86	2.79	2.79	0.49	0.43	0.41	184.45	185.74	196.33	93.65	86.94	100.81
20	2.75	2.72	2.82	0.48	0.43	0.49	185.72	186.56	197.26	98.27	94.07	98.39

times under each setting). Without CRAMES, the system was only able to provide 16 MB of memory to memeater. With CRAMES, 33 MB of memory were provided and the execution time was proportional to the amount of memory allocated, i.e., no delay was observed. Furthermore, when adaptive memory management is enabled (A-CRAMES), 38 MB of memory were allocated with no additional cost. These results support our claim in Section 4.3 that A-CRAMES helps to prevent on-line data compression deadlock.

4.4.3. Overall Performance of the Improved CRAMES

In Section 3.5, we demonstrated via experiments that on an embedded system with sufficient RAM to support its original (sets of) applications, CRAMES is capable of doubling the amount of available memory with negligible performance and energy consumption penalties for existing applications. This implies that with the help of CRAMES, the same hardware platform can easily support new, bigger (sets of) applications. On the other hand, this also implies that CRAMES allows applications to run on a system with less RAM. In other words, an embedded system with high-performance software memory compression could be designed with less RAM and still support desired applications with some performance and energy consumption penalties. In a system with substantial memory pressure, the improved CRAMES ensures that such penalties are minimal.

In order to evaluate the impact of using CRAMES to reduce physical RAM, we artificially constrained the memory size of a Zaurus with a kernel module that permanently reserves a certain amount of physical memory. With reduced physical RAM, we measured and compared the run times, power consumptions, and energy consumptions of four batch benchmarks, i.e., three applications from MediaBench [**37**] (Adpcm, Jpeg, Mpeg2) and a 512 by 512 matrix multiplication application. Figure 4.2 shows execution times, power consumptions, and energy consumptions of benchmarks running without compression, with LZO compression, and with PBPM compression under different memory constraints. Note that adaptive memory management was enabled in both LZO and PBPM compression to ensure fair comparison. In our experiments, each benchmark was executed multiple times; the average results are reported.

As shown in Figure 4.2, when system RAM was reduced to 8 MB, without CRAMES, all benchmarks suffered from significant performance degradation; the 512 by 512 matrix multiplication couldn't even execute due to memory constraints. However, with the help of CRAMES, all benchmarks were able to execute with only slight performance and energy consumption penalties. Compared with the base case in which system RAM is 20 MB and CRAMES is not used, PBPM compression results in an average performance penalty of 2.1% and a worst-case performance penalty of 9.2%. This represents a substantial improvement over LZO, for which the average performance penalty is 9.5% and the worst-case performance penalty can be as high as 29%.

4.5. Conclusions

High-performance OS controlled memory compression can assist embedded system designers to optimize hardware design for typical software memory requirements while also supporting (sets of) applications with larger data sets. In this chapter, we presented and evaluated a fast software-based compression algorithm for use in this application. This algorithm provides competitve compression ratio to existing algorithms used in on-line memory compression with significantly better symmetric performance. We also described an adaptive compressed memory management scheme to prevent on-line memory compression deadlock, permitting reliable and efficient on-line memory compression for a wide range of applications. Experimental results indicate that the improved CRAMES allows applications to execute with only slight penalties even when system RAM is reduced to 40% of its original size.

Chapter 5

Increase Usable Memory in MMULess Embedded Systems

Random access memory (RAM) is tightly-constrained in many embedded systems. This is especially true for the least expensive, lowest-power embedded systems, such as sensor network nodes and portable consumer electronics. The most widely-used sensor network nodes have only 4–10 KB of RAM and do not contain memory management units (MMUs). It is very difficult to implement increasingly complex applications under such tight memory constraints. Nonetheless, price and power consumption constraints make it unlikely that increases in RAM in these systems will keep pace with the requirements of applications.

We propose to increase the amount of usable memory in MMU-less embedded systems using a CRAMES-like architecture with the assistance of automated compile-time and run-time techniques. This architecture, named MEMMU, does not increase hardware cost, and are designed to require few or no changes to existing applications. We have developed a fast compression algorithm well suited to this application, as well as run-time library routines and compiler

transformations to control and optimize the automatic migration of application data between compressed and uncompressed memory regions. These techniques were experimentally evaluated on Crossbow TelosB sensor network nodes running a number of data collection and signal processing applications. The results indicate that available memory can be increased by up to 50% with less than 10% performance degradation for most benchmarks.

This chapter gives a brief background introduction of the automatic compile-time and run-time techniques, and focuses on the compression algorithm we developed for compressing sensor data. We refer interested readers to a previous publication [**40**] for a detailed discussion of the compiler and run-time techniques. The rest of this chapter is organized as follows. Section 5.1 introduces the problem background and addresses the importance of MEMMU. Section 5.2 introduces related techniques. Section 5.3 provides a high-level overview of the library and compiler techniques, and optimization schemes used in MEMMU. Section 5.4 presents the compression and decompression algorithms designed to automatically increase usable memory in sensor network nodes. Section 5.5 describes the evaluation set-up and work-loads, and briefly summarizes the experimental results. Finally, Section 5.6 concludes this chapter.

5.1. Introduction

Low-power, inexpensive embedded systems are of great importance in applications ranging from wireless sensor networks to consumer electronics. In these systems, processing power and physical memory are tightly limited due to constraints on cost, size, and power consumption. For example, eight-bit microcontrollers generally have no memory management units (MMUs). Sensor network nodes are among the embedded systems whose resource and power are most

tightly constrained. Although the proposed techniques may be used in any memory-constrained embedded system without an MMU, this article will focus on using them to increase usable memory in sensor network nodes with no changes to hardware and with no or minimal changes to applications.

Many recent ideas for improving the communication, security, and in-network processing capabilities of sensor networks rely on sophisticated routing [**41**], encryption [**42**], query processing [**43**], and signal processing [**44**] algorithms implemented on sensor network nodes. However, sensor network nodes have tight memory constraints. For example, the popular Crossbow MICA2, MICAz, and TelosB sensor network nodes have between 4 KB and 10 KB of RAM, a substantial portion of which is consumed by the operating system (OS), e.g., TinyOS [**45**] or MANTIS OS [**46**]. Tight constraints on cost and power consumption of sensor network nodes make it unlikely for the size of physical RAM to keep pace with the demands of increasingly sophisticated in-network processing algorithms.

In order to reduce cost, sensor network nodes typically avoid the use of dedicated dynamic random access memory (DRAM) integrated circuits; in extremely low price, low power embedded systems, RAM is typically on the same die as the processor. Unfortunately, it is not economical to fabricate the deep trench capacitors used for high-density RAM with the same process as processor logic. As a result, static random access memory (SRAM) is used in sensor network nodes. Unlike DRAM, SRAM generally requires six transistors per bit and has high power consumption. Increasing the amount of physical memory in sensor network nodes would increase die size, and hence cost, as well as power consumption. Some researchers

have proposed addressing memory constraints using hardware techniques such as compression units inserted between memory and processor. However, such hardware implementations typically have difficulty adapting to the characteristics of different application data. Moreover, they would increase the price of sensor network nodes either by requiring additional integrated circuit packages or by requiring microcontroller redesign. Barring new technologies that allow inexpensive, high-density, low-power, high-performance RAM to be fabricated on the same integrated circuits as logic, sensor network applications will continue to face strict constraints on RAM in the future.

Software techniques that use data compression to increase usable memory have some advantages over hardware techniques. They do not require processor or printed circuit board redesign and they allow the selection and modification of compression algorithms, permitting good performance and compression ratio (compressed data size divided by original data size) for the target application. However, software techniques that require the re-design of applications are unlikely to be used by anybody but embedded systems programming experts. Most sensor network application experts are not embedded system programming experts. If memory expansion technologies are to be widely deployed, they should not require changes to hardware and should require minimal or no changes to applications.

We propose a new memory expansion technique, named MEMMU, for use in wireless sensor networks. This technique uses compile-time transformation and run-time library support to automatically manage on-line migration of data between compressed and uncompressed memory regions in sensor network nodes. It provides application developers with access to more

usable RAM and requires no or minor changes to application code and no changes to hardware. The proposed technique requires no MMU and has other design features enabling its use in sensor network nodes with extremely tight memory and performance constraints. It has been optimized to minimize impact on performance and power consumption; experimental results indicate that in many applications, such as data sampling and audio signal correlation computation, its overhead is small.

5.2. Related Work

The proposed library and compiler techniques to increase usable memory were built upon work in the areas of on-line data compression, wireless sensor networks, and high-performance data compression algorithms. Please refer to Section 3.2 and Section 4.1 for the discussion of related techniques in on-line data compression and data compression algorithms.

5.2.1. Software Virtual Memory Management for MMU-Less Systems

Choudhuri and Givargis [**47**] proposed a software virtual memory implementation for MMU-less embedded systems based on an application level virtual memory library and a virtual memory aware assembler. They assume that secondary storage, e.g., EEPROM or Flash, is present in the system. Their technique automatically manages data migration between RAM and the secondary storage to provide applications access to more memory than provided by physical RAM. However, since access to such secondary storage is significantly slower than that to RAM, the performance penalty of this approach can be very high for some applications. In

contrast, MEMMU requires no secondary storage and its performance and power consumption penalties have been minimized via various optimization techniques.

5.2.2. Compression for Reducing Communication in Sensor Networks

In many sensor network applications, sensor nodes in the network must frequently communicate with each other or with a central server. Sensor nodes have limited power sources and wireless communication accelerates battery depletion [**48**]. In-network data aggregation [**49**,**50**] and data reduction via wavelets or distributed regression [**51**, **52**] can significantly reduce the volume of communicated data. However, these techniques are lossy, limiting their application. Recently, researchers have proposed to reduce the amount of data communication via compression [**53**, **54**] in order to reduce radio energy consumption. In contrast, MEMMU focuses on automated memory compression for functionality improvement instead of communication reduction.

5.3. Memory Expansion on Embedded Systems Without MMUs

This section provides a high-level overview of *MEMMU* (Memory Expansion on embedded systems without MMUs). The main goal of MEMMU is to provide application designers with access to more usable RAM than is physically available in MMU-less embedded systems without requiring changes to hardware and with minimal or no changes to applications. MEMMU achieves this goal via on-line compression and decompression of in-RAM data.

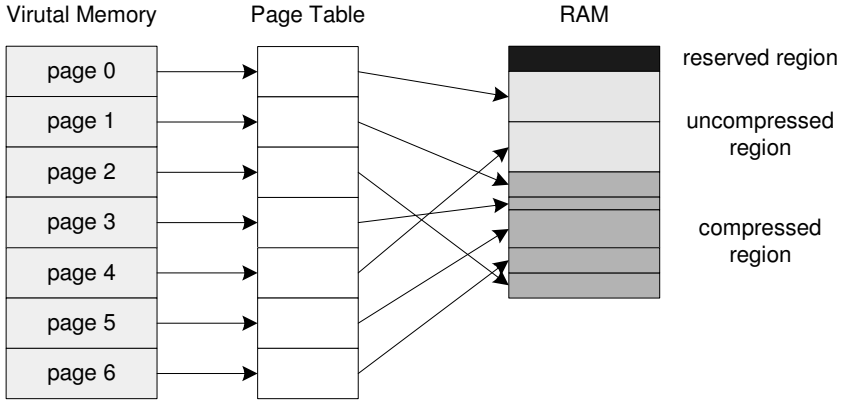

FIGURE 5.1. Memory layout of the MEMMU architecture.

MEMMU divides physical RAM into three regions: the reserved region, the compressed region, and the uncompressed region. The reserved region is used to store uncompressed data of the OS, including MEMMU. The compressed region and the uncompressed region are both used by applications. Application data are automatically migrated between these regions; the size of each region is decided by compile-time analysis of application memory requirements and estimated compression ratio. The compressed region can be viewed as a high-capacity but somewhat slower form of memory, and the uncompressed region can be viewed as a small, high-performance data cache.

Figure 5.3 illustrates the memory layout of an embedded system using MEMMU. From the perspective of application designers, all pages in the left-most *Virtual Memory* column are available. Memory objects in virtual memory are mapped to the uncompressed or compressed region via a software-maintained page table. A memory management mechanism was designed to manage data compression, decompression, and migration between two regions.

5.3.1. Handle-Based Access

Data elements are accessed via their virtual address handles. The mapping from virtual page to RAM is stored in a page table maintained as an array. When data are accessed via their virtual addresses within an application, MEMMU first determines the status of the corresponding virtual page based on the page table, and then decides whether to locate the page in compressed or uncompressed memory region. Recently accessed pages are always moved to the uncompressed region, if they previously present in the compressed region. Note that during this process victim pages in the uncompressed region may be selected to be compressed.

In order to make the procedure transparent to users, and to avoid increasing application development complexity, the routines for these operations are stored in a runtime library and compiler transformations are used to convert memory accesses within unmodified code to library calls.

5.3.2. Memory Management and Page Replacement

When the uncompressed memory region is filled by an application, its pages will be incrementally moved to the compressed region to make space available in the uncompressed region. When data in the compressed region are later accessed, they are decompressed and moved back to the uncompressed region. Ideally, pages that are unlikely to be used for a long time should be compressed to minimize the total number of compression and decompression events. MEMMU approximates this behavior via a least-recently used (LRU) victim page selection policy.

Managing the compressed region is complex since page sizes differ, however can be solved using similar methods used in CRAMES memory management. Please refer to Section 3.3.2.3 for a more illustrative discussion of the solutions.

5.3.3. Preventing Fragmentation

One potential problem with dynamic memory allocation is fragmentation. Fragmentation can prevent a newly compressed page from fitting in the compressed region even though the total available memory in that region is sufficient. MEMMU performs memory merging and coalescing to prevent fragmentation.

Free block merging takes place every time a page is decompressed and moved from the compressed region. If a free block is adjacent to its predecessor or successor, these adjacent blocks are merged. Coalescing occurs when the memory allocator fails to allocate a new block from the free list. In this case, MEMMU locates pages in order of increasing addresses and moves them to the top of the compressed region, or to the bottom of the most-recently moved pages. This process continues until all compressed pages have been moved. Upon completion, a single large free region remains.

5.3.4. Interrupt Management

The primary target platform for MEMMU is wireless sensor network nodes, which are typical memory-constrained, MMU-less embedded systems. On sensor nodes, hardware interrupts often take place when newly-sensed data arrive. Missing sampling events can be avoided as long as the sampling period is longer than the worst-case delay. However, arbitrarily reducing

sampling rate is not an acceptable solution because some applications may require high sampling rates and even infrequent events may occur during a page miss. To solve this problem, a *ring buffer* may be used; please refer to [**40**] for a more detailed discussion.

5.3.5. Optimization Techniques

In the previous sections, we described the basic design components of the MEMMU memory expansion system. Every memory access requires (1) a runtime handle check to ensure the address is in the uncompressed region, (2) an update to the LRU list, and (3) a virtual to physical address translation. This introduces high execution time overhead that is proportional to the total number of memory accesses. Hence, the basic solution is not practical for many real applications on embedded systems. However, the technique can be optimized to significantly reduce the number of runtime checks, LRU list updates, and address translations. Below are several such compile-time optimization techniques used in MEMMU.

- **Frequent references optimization**: If a small data structure is used very frequently in the application, it should be allocated to the reserved region at compile time to eliminate all handle checks and address translations.
- **Run-time handle check optimization:** If a sequence of memory references access a same page, only the first handle check is necessary since the referenced page is sure to be in the uncompressed region on subsequent accesses.
- **Loop transformation and compile-time elimination of inner-loop checks:** This optimization scheme can further reduce runtime handle checks by means of compile-time

loop transformations because access to an array in a loop usually uses an incremental memory reference pattern.

- **Handle check hoisting:** By hoisting handle checks, multiple handle checks inside a loop can be replaced with one handle check outside the loop. This optimization requires that the total size of the accessed pages is no larger than the size of the uncompressed region. It can be viewed as prefetching pages and locking them in the uncompressed region until a portion of code finishes execution.
- **Pointer dereferencing to reduce address translation:** There are usually dependencies among sequences of reference addresses. Many applications use a constant stride in memory reference sequences. If the physical address of memory object A is known, and object B is in the same page as A, the physical address of B can be determined by dereferencing and adding an offset to the pointer to A instead of computing results based on page table contents.

5.4. Delta Compression Algorithm

Many software-based memory compression algorithms are not appropriate for use on sensor network nodes due to large memory requirements or poor performance. For those with sufficiently low overhead, we found none that provides a satisfactory compression ratio for sensor data. The main reasons for this are that (1) zero words are rare in sensor data, (2) the similarities among sensor datum are not sufficient even though data often change gradually with time, and (3) the page size is usually significantly smaller in low-cost MMU-less devices than in other

Algorithm 2 Delta (a) compression and (b) decompression

Require: *IN, OUT, DATA* word stream
Require: *TAPE* delta stream
1: **for** $i \in \{1, \cdots, N\}$ **do**
2: $\delta \leftarrow$ *IN*[*i*] - *IN*[*i-1*]
3: **if** $log_2\delta \leq$ *MAXBITS* **then**
4: *TAPE*[*i*] $\leftarrow \delta$
5: **else**
6: *TAPE*[*i*] $\leftarrow$ *MAGIC_CODE*
7: *DATA*[*i*] $\leftarrow$ *IN*[*i*]
8: **end if**
9: *OUT* $\leftarrow$ pack*(TAPE, DATA)*
10: **end for**

Require: *IN, OUT, DATA* word stream
Require: *TAPE* delta stream
1: *DATA, TAPE* $\leftarrow$ unpack(*IN*)
2: **for** *TAPE*[*i*] in range of *TAPE* **do**
3: **if** *TAPE*[*i*] = *MAGIC_CODE* **then**
4: *OUT*[*i*] $\leftarrow$ *DATA*[*i*]
5: **else**
6: $\delta \leftarrow$ *TAPE*[*i*]
7: *OUT*[*i*] $\leftarrow$ *OUT*[*i-1*] + δ
8: **end if**
9: **end for**

embedded systems. We propose a memory compression algorithm that operates with very high performance on the 16-bit data generally found in the memory of MICAz and TelosB sensor network nodes. The average compression ratio for various types of sensor data is approximately 50%.

We developed a high-performance, lossless compression algorithm based on delta compression for use in sensor network applications. This algorithm exploits the similarities between adjacent data elements. Despite its simplicity, the algorithm has high performance and a good compression ratio for sensor data in which adjacent samples are often correlated.

To design an appropriate compression algorithm for sensor data, the regularities of the data must be well understood. For this purpose, we collected numerous types of sensor data, e.g., sound, light, and temperature, from Crossbow MICAz and TelosB sensor network nodes and analyzed their characteristics. Intuitively, sensor data are likely to stay similar during a certain period of time, and within a certain geographic range, hence showing high amounts of temporal

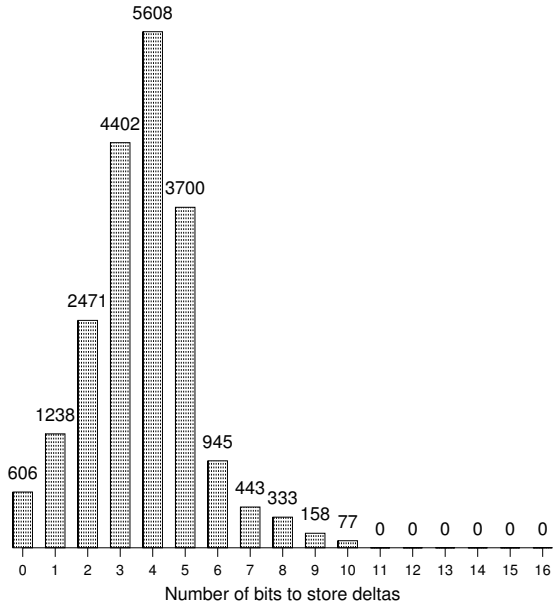

FIGURE 5.2. Histogram of compression bits.

and spatial locality. For example, in sensor network deployed for seabird habitat monitoring [**55**] sensor nodes may be placed in petrel nests in underground burrows. The temperature and humidity sensed from one sensor node usually changes smoothly during a day, except as a result of storms. In addition, the sensor data of temperature and humidity from adjacent burrows are likely to be similar; these data are usually transmitted within a cluster of nodes before they are sent to the base station. Thus, sensor nodes commonly contain highly-redundant data.

A delta-based compression algorithm exploits regularity in data: the difference between two adjacent data elements (delta) usually requires fewer bits to store than the original data. Our implementation of the delta compression and decompression algorithms are presented in Algorithm 2. The algorithms are based on the observation that the majority of the deltas can

be stored within a pre-defined $MAXBITS$; if the delta cannot be stored within $MAXBITS$, i.e., there is a sudden change in sensed data, the raw data are stored and a $MAGIC\_CODE$ is recorded to indicate this abnormality. The algorithm also adapts to the compressibility of pages by means of early termination. When the number of deltas that exceed $MAXBITS$ is above a certain threshold, causing the "compressed" page to exceed its original size before compression, the algorithm terminates and reports the compressed page size as zero, indicating that this page is not compressed.

In order to identify the $MAXBITS$ value that provides the best compression ratio, we statistically analyzed the sample sound data collected by the Crossbow MICAz sensor node. Since the analog-to-digital converter (ADC) on the MICAz generates a 10-bit output, the compression algorithm reads in 2 bytes (16 bits) at a time and computes the delta on a 2 bytes basis. Figure 5.4 shows that 95% of the deltas can be represented using six bits. Therefore, in our implementation, $MAXBITS$ is set to six. Please note that this value may vary depending on the underlying hardware of the sensor node, i.e., the bit width of the ADC.

5.5. Evaluation and Experimental Results

We evaluted MEMMU on a TelosB wireless sensor node. The TelosB is an MMU-less, low-power, wireless module with integrated sensors, radio, antenna, and an MSP430 microcontroller. It has 10 KB RAM and typically runs TinyOS. Five representative applications in the wireless sensor network domain were used as our benchmarks. These applications are: sound

filtering, image convolution, data sampling, covariance matrix computation, and correlation calculation.

The benchmarks are tested under three system settings: running the original applications without MEMMU, with an unoptimized version of MEMMU, and with an optimized version of MEMMU. Our experimental results show that using optimized MEMMU available memory can be increased by up to 50%, and that with the exception of the image convolution benchmark, the execution time overhead of all other four benchmarks are below 10%. For a more detailed presentation of the experiments and results, please refer to [**40**].

5.6. Conclusions

In this chapter, we have introduced MEMMU, an efficient software-based technique to increase usable memory in MMU-less embedded systems via automated on-line compression and decompression of in-RAM data. A number of compile-time and run-time optimizations are used to minimize its impact on the performance and power consumption of the target systems. An efficient delta-based compression algorithm was designed for sensor data compression. MEMMU was evaluated using a number of representative wireless sensor network applications. Experimental results indicate that the optimization technique effectively improve MEMMU's performance and that MEMMU is capable of significantly increasing usable memory with small performance and power consumption penalties.

Chapter 6

Hardware-Based Cache and Main Memory Compression

Microprocessor designers have been torn between tight constraints on the amount of on-chip cache memory and the high latency of off-chip memory, such as dynamic random access memory (DRAM). Accessing off-chip memory generally takes an order of magnitude more time than accessing on-chip cache memory, and two orders of magnitude more time than executing a register-only instruction.

This chapter addresses the increasingly-important issue of controlling off-chip communication in computer systems in order to maintain good performance and energy efficiency. We present a unified, adaptive on-chip cache and off-chip memory compression system, with the assistance of a lossless hardware compression algorithm that has been designed for fast on-line data compression, and cache compression in particular.

6.1. Introduction

As described in Chapter 2, microprocessor speeds have been increasingly faster than off-chip memory latency, raising an increasing gap between processor and memory. The ongoing move to chip-level multiprocessors (CMPs) is further increasing the problem; more processors require more accesses to memory, but the performance of the processor–memory bus is not keeping pace. Techniques that reduce off-chip communication without degrading performance have the potential to solve this problem. *Cache compression* is one such technique; data in last-level on-chip caches, e.g., L2 caches, are compressed, resulting in larger usable caches. In the past, researchers have reported that cache compression can improve the performance of uniprocessors by up to 17% for memory-intensive commercial workloads [**56**] and up to 225% for memory-intensive scientific workloads [**57**]. Researchers have also found that cache compression and prefetching techniques can improve CMP throughput by 10–51% [**58**].

Although cache compression appear to be a promising technique to relieve the processor–memory gap, there are several complex issues involved when deploying this technique in microprocessor architecture. For example, the compression ratios of data from different applications, during different execution phases, may vary widely. Therefore, compressed cache lines may not be organized into uniform-size slots, making it challenging to efficiently distribute and locate data in the compressed cache. Unfortunately, most past work either assumes a fixed average

compression ratio for all cache lines, avoiding the complexity of variation in compression ratios, or assumes a segmentation based organization for compressed cache lines. Our evaluation indicates that this organization imposes high hardware and performance overhead.

Cache compression should be adaptive instead of static. Compression may increase effective cache size, thereby reducing the number of cache misses. However, it also increases cache hit latency, because each instruction accessing a compressed line must wait till the line is decompressed. For applications that are not performance-sensitive to cache size but are performance-sensitive to cache hit latency, compression does not bring any benefit, rather, it increases hit latency and therefore hurts performance. However, it is challenging to adapt compression to applications and execution phases, because the application sensitivity to cache size and latency is not easily obtainable. Moreover, not all cache lines are compressible. For lines that have bad compression ratios, compression increases hit latency without reducing cache misses.

Moreover, most previous cache compression techniques have commonly made assumptions about the implications of using existing compression algorithms for cache compression and the design of special-purpose cache compression hardware. However, they did not demonstrate whether the proposed compression and decompression hardware is appropriate for cache compression, considering the performance, area, and power consumption requirements. Some researchers, such as Alameldeen et al. [**56**], did a high level analysis of the hardware design implications of cache compression. However, detailed analysis is essential to estimate the performance impact of using cache compression.

In this chapter, we propose effective solutions to the above problems that present in cache compression. We propose a pair-matching scheme to store compressed cache lines, which is more efficient than previous segmentation-based approaches. In our proposed system, data in on-chip caches are adaptively compressed to benefit applications that require large cache without adding unnecessary overhead to applications that do not. In addition, data that are transferred between on-chip cache and off-chip main memory are compressed to reduce processor–memory communication latency and bandwidth requirements. We also propose a highly-efficient, lossless, hardware compression algorithm that has been designed for fast on-line data compression, and cache compression in particular. This work was done in collaboration with other researchers. In particular, Xi Chen was the leader on implementing, synthesizing, and evaluating the compression and decompression hardware.

Since it is not possible to determine whether compression at levels of the memory hierarchy closest to the processor is beneficial without understanding its costs, we reduced the proposed cache compression algorithm to a register transfer level hardware implementation, permitting performance, power consumption, and area estimation. To evaluate the adaptive cache compression and memory link compression techniques, we performed detailed full-system simulations using a wide range of benchmarks. Simulation results indicate that our techniques achieve an average performance improvement of 20.76% on 87.5% of the evaluated benchmarks. In the best case, the performance is improved by 107.14%. The performances for the rest of the benchmarks are not affected.

6.2. Related Work

A number of hardware cache compression [**59**, **60**, **57**, **61**, **56**, **62**, **63**, **64**] and memory compression [**7**, **8**] systems have been developed to increase effective cache size and memory size, thereby improving system performance.

Kjelso et al. [**9**] proposed an X-Match hardware compression algorithm for main memory compression. X-Match is a dictionary-based compression algorithm that has been implemented on an FPGA [**65**]. It matches 32-bit words using a content addressable memory that allows partial matching with dictionary entries and outputs variable-size encoded data that depend on the type of match. To improve coding efficiency, it also uses a move-to-front coding strategy and represents smaller indices with fewer bits. Although appropriate for compressing main memory, such hardware usually has a very large block size (1 KB for MXT and up to 32 KB for X-Match), which is inappropriate for compressing cache lines.

IBM's MXT (Memory Expansion Technology) [**7**] is a hardware memory compression technique that improves the performance of servers via increasing the usable size of off-chip main memory. Data are compressed in main memory and decompressed when moved from main memory to the off-chip shared L3 cache. Memory management hardware dynamically allocates storage in small sectors to accommodate storing variable-size compressed data block without the need for garbage collection. IBM reports compression ratios (compressed size divided by uncompressed size) ranging from 16% to 50%. Unfortunately, the performance, area, or power

consumption costs of such memory compression hardware contradict its use in cache compression. For example, if the MXT hardware were scaled to a 65 nm fabrication process and integrated within a 1 GHz processor, the decompression latency would be 16 processor cycles, about twice the normal L2 cache hit latency.

Benini et al. [**8**] proposed a data compression/decompression scheme to reduce memory traffic in general-purpose processor systems. They store uncompressed data in the cache, and compress/decompress on the fly when data is transferred to/from memory. They use a differential compression scheme based on the assumption that it is likely for data words in the same cache line to have some bits in common.

Lee et al. [**59**] proposed a compressed memory hierarchy model that selectively compresses L2 cache and memory blocks that can be reduced to half their original size. They used a variant of the X-Match algorithm that treats runs of zeros specially.

Hallnor and Reinhardt [**57**] proposed an indirect-index cache design to allocate variable amounts of storage to different cache lines based on their compressibility. Compressed cache lines are divided into fix-sized data segments, and indirection is used to translate cache line addresses into the physical indices in a data store where these segments are held.

Alameldeen and Wood [**56**] presented the first adaptive compression scheme for hardware caches. In their system, L1 cache is not compressed and L2 cache is adaptively compressed, depending on the LRU stack depth and current compression ratio. Their system uses decoupled variable-segment cache, where each cache line data array is broken into eight-byte segments,

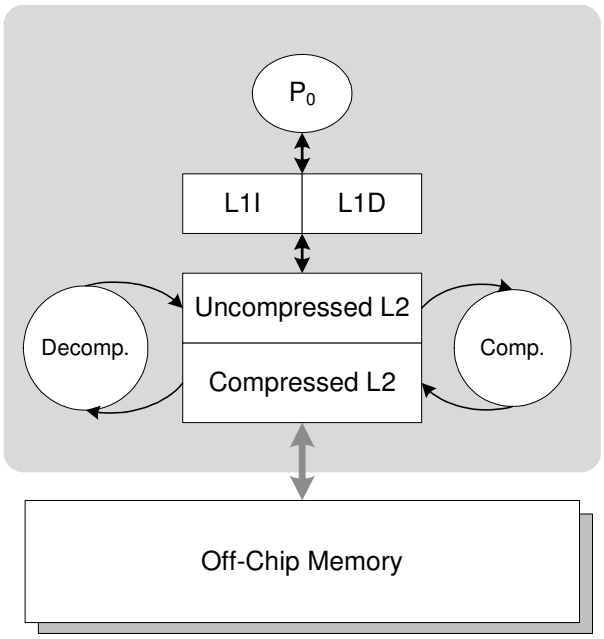

FIGURE 6.1. Cache and memory compression architecture overview.

with 32 segments statically allocated to each cache set. The frequent pattern compression hardware (FPC) [**66**] is used to compress and decompress L2 cache lines. Based on logical effort analysis [**67**], for a 64-byte cache line, compression can be completed in three cycles and decompression in five cycles, assuming 12 fan-out-four (FO4) gate delays per cycle. To the best of our knowledge, there is no register-transfer-level hardware implementation or FPGA implementation of FPC, and therefore its exact performance, power consumption, and area overheads are unknown. Although the area cost for FPC [**66**] is not discussed, our analysis shows that FPC would have an area overhead of at least 290 K gates, almost eight times the area of the approach proposed in this chapter, to achieve the claimed 5-cycle decompression latency.

6.3. Overview of the Cache and Memory Compression Architecture

In this section, we describe the architecture of a on-chip cache and off-chip memory compression. As illustrated in Figure 6.3, the L1 cache is split for instructions and data. The L2 cache is unified and is divided into an uncompressed region and a compressed region. The sizes of the uncompressed region and compression region adapt to the processor's needs dynamically. In extreme cases, the whole L2 cache can be compressed due to high cache capacity requirement, or uncompressed to minimize performance penalty on decompressing data. We assume a three-level cache hierarchy consisting of L1 cache, uncompressed L2 region, and compressed L2 region. The L1 cache communicates with the uncompressed region of the L2 cache, which in turn exchanges data with the compressed region through the compressor and decompressor, i.e., an uncompressed line can be compressed in the compressor and placed in the compressed region, and vice versa. Note that this new three-level cache hierarchy is not strictly inclusive. Data in the uncompressed L2 region are never in the compression L2 region. However, data in the L1 cache must also be in one of the two L2 regions.

In addition, cache lines are transferred in compressed format between compressed L2 region and off-chip main memory, if the data are compressible. Each cache line is stored in memory in uncompressed or C-Pack compressed format. To indicate whether a line is compressed or not, one more "compressed" bit is necessary for each cache line. There are two options: (1) using one bit encoded in the ECC, which would prevent memory from operating in the ECC mode, and (2) using a small RAM to keep the compressed bits for all cache lines. Note that the extra

space in the original cache line resulted from compression is not used by another compressed cache line. In other words, this architecture does not increase effective memory capacity, but reduces off-chip memory access latency and bus bandwidth requirement.

6.4. Pair Matching Compressed Line Organization

Compressed cache organization is a difficult task because different compressed cache lines may have different lengths. Some researchers have proposed line segmentation techniques [**57**, **56**] to handle this problem. The main idea is to divide compressed cache lines into fixed-size segments and use indirect indexing to locate all segments for a compressed line. However, the segmentation approach has significant overhead due to the complicated hardware necessary to address all segments. For a 4-way set-associative compressed cache with 8 segments per line, each set contains 8 compression information tags and 8 address tags because each set is constrained to hold up to eight compressed lines. The compression information tag indicates (1) whether the line is compressed and (2) the compressed size of the line. Data segments are stored contiguously in address tag order. In order to extract a compressed line from a set, eight segment offsets are computed in parallel with the address tag match. Therefore, deriving the segment offset for the last line in the set requires summing up all the previous 7 compressed sizes, which incurs a significant performance overhead. In addition, although the cache array may be split into two banks to reduce line extraction latency, addressing the whole compressed line may still takes 3 cycles in the worst case. During line insertion, the size of a newly inserted line may be larger than the LRU line plus the unused segments. In that case, prior work

proposed replacing two lines by replacing the LRU line and searching the LRU list to find the least-recently-used line that ensures enough space for the newly arrived line [**56**]. However, this will result in great performance and area overhead. Set compaction may be required after line insertion to maintain the contiguous storage invariant. This can be prohibitively-expensive in terms of latency and area cost because it may require reading and writing all the set's data segments. Cache compression techniques that assume segmented compressed lines are essentially proposing to implement kernel memory allocation and compaction in hardware [**26**].

We propose a *pair-matching* organization to store compressed cache lines. In a pair-matching based cache, the location of a newly compressed line depends on not only its own compression ratio but also the compression ratio of its "partner". More specifically, the compressed line locator first tries to locate the cache line (within the set) with sufficient unused space for the new compressed line without replacing any existing compressed lines. If no such line exists, one or two compressed lines are evicted to store the new line. A compressed line can be placed in the same line with a partner only if the sum of their compression ratios is less than 100%. Note that successful placement of a line does not require that it have a compression ratio smaller than 50%. It is only necessary that the line, combined with a "partner line" be as small as an uncompressed line. To reduce hardware complexity, the candidate partner lines are only selected from within the same set of the cache. Compared to segmentation techniques which allow arbitrary positions, pair-matching simplifies the hardware that manages the locations of the compressed lines. More specifically, line extraction in a pair-matching based cache only requires parallel

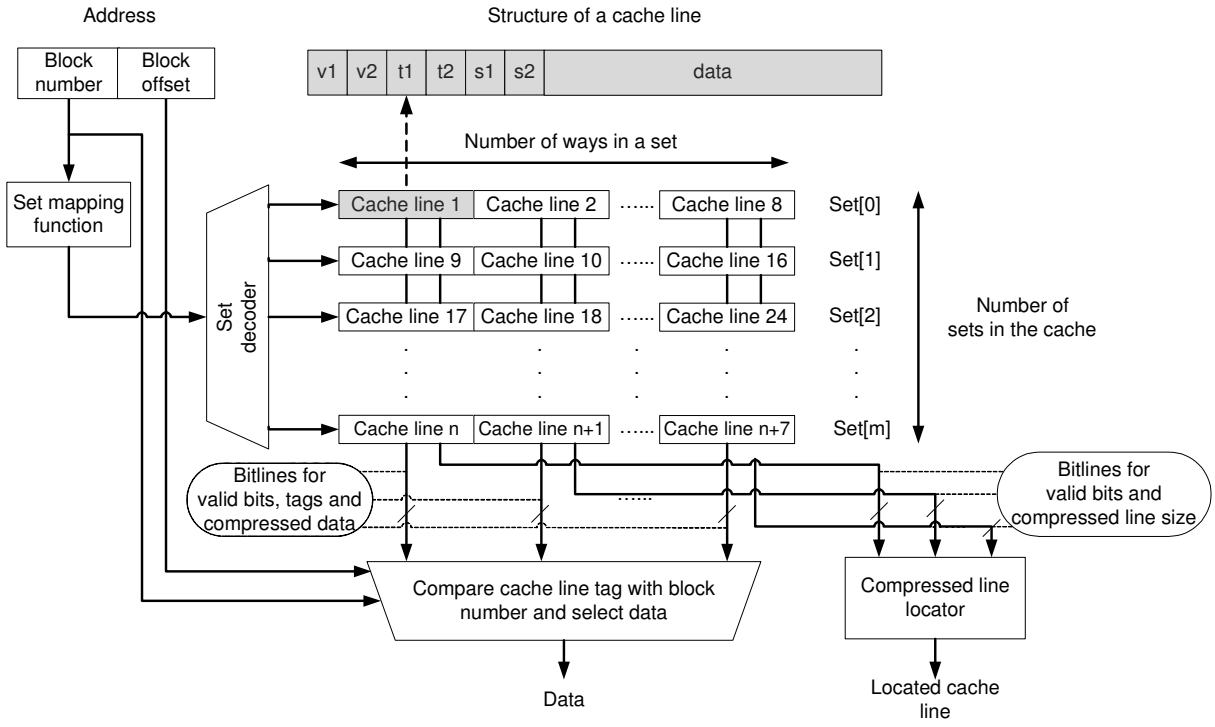

FIGURE 6.2. Structure of a pair-matching based cache.

address tag match and finishes in a single cycle. For line insertion, neither LRU list search nor set compaction is involved.

Figure 6.4 illustrates the structure of an 8-way set-associative pair-matching based cache. Since any line may store two compressed lines, each line has two *valid* bits and *tag* fields to indicate status and indexing. When compressed, two lines share a common *data* field. There are two additional *size* fields to indicate the compressed sizes of the two lines. Whether a line is compressed or not is indicated by its *size* field. A *size* of zero is used to indicate uncompressed lines. For compressed lines, *size* is set to the line size for an empty line, and the actual compressed size for a valid line. On 32-bit architectures, for a 64-byte line, the tag is no longer than

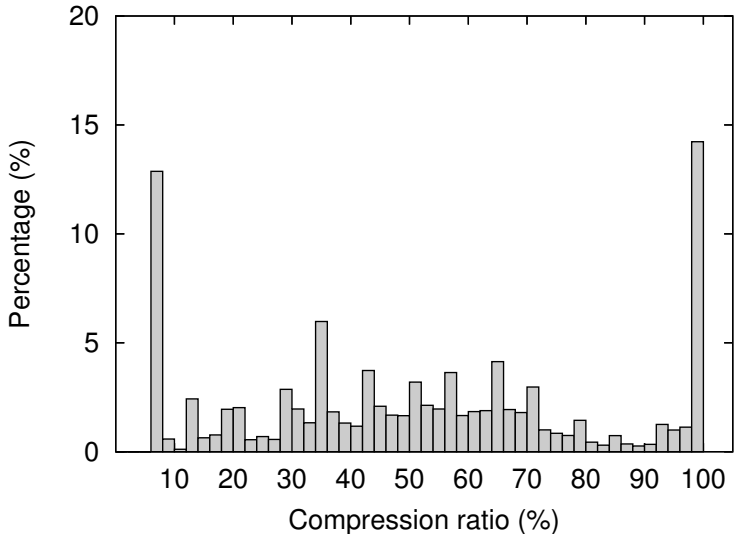

FIGURE 6.3. Distribution of compression ratios.

32 bits, hence the worst-case cache size overhead is less than 32 (tag) + 1 (valid) + 2 $\times$ 7 (size) bits, i.e., 6 bytes.

As shown in Figure 6.4, the compressed line locator uses the bitlines for valid bits and compressed line sizes to locate a newly compressed line. Note that only one compressed line locator is required for the entire compressed cache. This is because for a given address, only the cache lines in the set which the specific address is mapped to are activated, thanks to the set decoder. Each bitline is connected to a sense amplifier, which usually requires several gates [**68**], for signal amplification and delay reduction. The total area overhead is approximately 500 gates plus the area for the additional bitlines, compared to an uncompressed cache.

6.5. Effective System-Wide Compression Ratio

Based on the pair-matching concept, a newly-compressed line has an effective compression ratio of 100% when it takes up a whole cache line, and an effective compression ratio of 50% when it is placed with a partner in the same cache line. Note that when a compressed line is placed together with its partner without evicting any compressed lines, its partner's effective compression ratio decreases to 50%.

We define *effective system-wide compression ratio* as the average effective compression ratio of all cache lines in a compressed cache. It indicates how well a compression algorithm performs for pair-matching based cache compression. This effective compression ratio metric can also be adapted to a segmentation based approach. For example, for a cache line with 4 fixed-length segments, a compressed line has an effective compression ratio of 25% when it takes up one segment, 50% for two segments, 75% for three segments, and 100% for four segments (not compressible).

Varying raw compression ratio between 25% and 50% has little impact on the effective cache capacity of a four-part segmentation based technique. Figure 6.5 illustrates the distribution of raw compression ratios for different cache lines derived from real cache data. The x-axis shows different compression ratio intervals and y-axis indicates the percentage of all cache lines in each compression ratio interval. For real cache trace data, pair matching generally achieves a better effective system-wide compression ratio (58%) than line segmentation with four segments per line (62%) and the same compression ratio as line segmentation with eight segments,

which would impose substantial hardware overhead. We therefore use pair-matching to organize compressed cache lines in the compressed memory hierarchy presented in the following sections and use effective system-wide compression ratio as a metric for comparing different compression algorithms.

6.6. C-Pack Hardware Compression Algorithm

The C-Pack hardware compression algorithm was developed in collaboration with other researchers. In particular, Xi Chen was the leader on implementing, synthesizing, and evaluating the compression and decompression hardware. This section only gives an overview of the algorithm. For more details on C-Pack, interested readers may refer to our previous publication [**69**], which presents the hardware implementation of C-Pack, and compares its performance, compression ratio, and hardware overheads with other existing cache compression hardware.

6.6.1. Design Challenges

Cache compression presents several challenges. First, decompression and compression must be extremely fast: a significant increase in cache hit latency will overwhelm the advantages of reduced cache miss rate. This requires an efficient on-chip decompression hardware implementation. Second, the hardware should occupy little area compared to the corresponding decrease in the physical size of the cache, and should not substantially increase the total chip power consumption. Third, the algorithm should losslessly compress small blocks, e.g., 64-byte cache lines, while maintaining a good compression ratio. Conventional compression algorithm quality metrics, such as block compression ratio, are not appropriate for evaluating quality in

TABLE 6.1. Pattern Encoding For C-Pack

Code	Pattern	Output	Size (bits)	Frequency
00	zzzz	(00)	2	39.7%
01	xxxx	(01)BBBB	34	32.1%
10	mmmm	(10)bbbb	6	7.6%
1100	mmxx	(1100)bbbbBB	24	6.1%
1101	zzzx	(1100)B	12	7.3%
1110	mmmx	(1110)bbbbB	16	7.2%

this domain. Instead, the effective system-wide compression ratio should be used. Finally, cache compression should only impose small power consumption overhead. The above requirements prevent the use of high-overhead compression algorithms such as the PPM family [**70**] or Burrows-Wheeler transform [**71**]. A faster and lower-overhead technique is required.

6.6.2. C-Pack Compression Algorithm

C-Pack (for Cache Packer) is a lossless compression algorithm designed specifically for high-performance hardware-based on-chip cache compression. It achieves a good compression ratio when used to compress data commonly found in microprocessor low-level on-chip caches, e.g., L2 caches. Its design was strongly influenced by prior work on pattern-based partial dictionary match compression [**72**]. However, this prior work was designed for software-based main memory compression and did not consider a hardware implementation.

C-Pack achieves compression using (1) statically-decided, compact encodings for frequently-appearing data words and (2) a dynamically-updated dictionary allowing adaptation to other frequently-appearing words. The dictionary supports partial word matching as well as full word matching. The patterns and coding schemes used by C-Pack are summarized in Table 6.1,

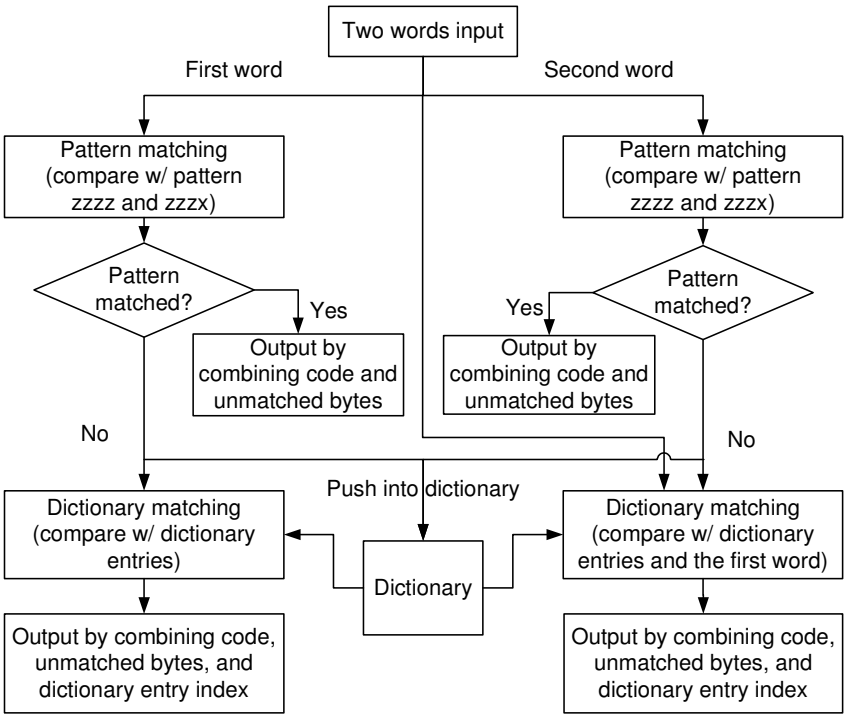

FIGURE 6.4. C-Pack compression.

which also reports the actual frequency of each pattern observed in real cache trace data file. The 'Pattern' column describes frequently-appearing patterns, where 'z' represents a zero byte, 'm' represents a byte matched against a dictionary entry, and 'x' represents an unmatched byte. In the 'Output' column, 'B' represents a byte and 'b' represents a bit.

The C-Pack compression and decompression algorithms are illustrated in Figure 6.6.2 and Figure 6.6.2. We use an input of two words per cycle as an example in the figures. However, the algorithm can be easily extended to cases with one or multiple words per cycle. During one iteration, each word is first compared with patterns "zzzz" and "zzzx". If there is a match, the compression output is produced by combining the corresponding code and unmatched bytes as indicated in Table 6.1. Otherwise, the compressor compares the word with all dictionary entries

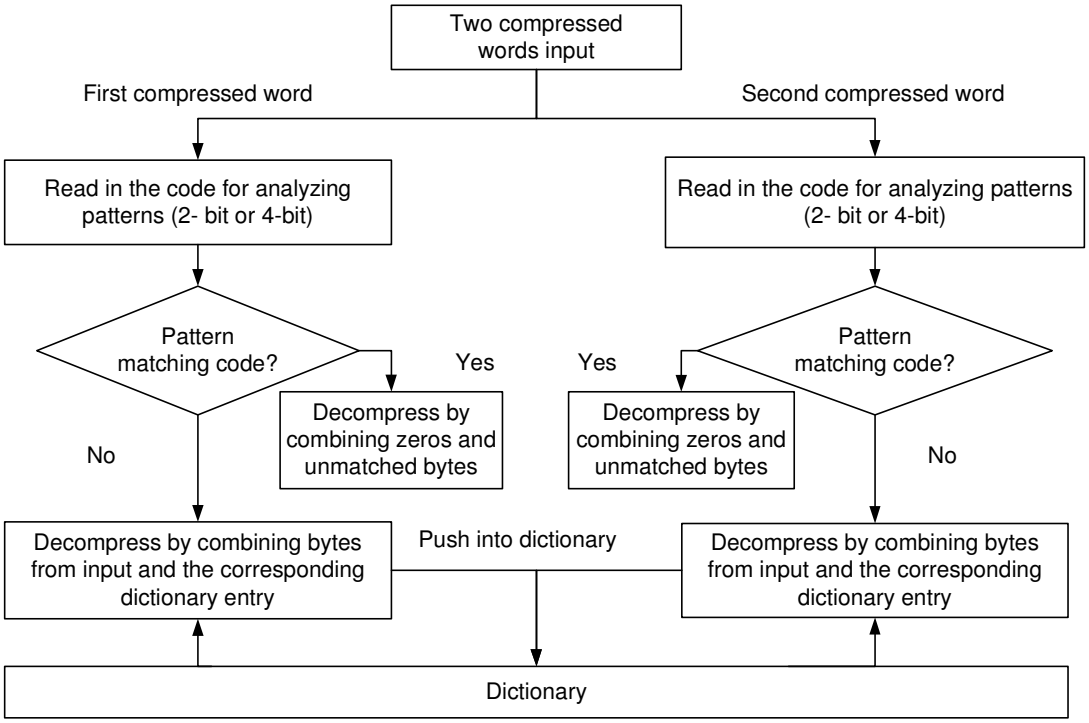

FIGURE 6.5. C-Pack decompression.

and finds the one with the most bytes matched. The compression result is then obtained by combining code, dictionary entry index, and unmatched bytes, if any. Words that fail pattern matching are pushed into the dictionary. Figure 6.6.2 shows the compression results for several different input words. In each output, the code and the dictionary index, if any, are enclosed in parentheses. Although we used a 4-word dictionary in Figure 6.6.2 for illustration, the dictionary size is set to 64 B in our implementation. Note that the dictionary is updated after each word insertion, which is not shown in Figure 6.6.2.

During decompression, the decompressor first reads compressed words and extracts the codes for analyzing the patterns of each word, which are then compared against the codes defined in Table 6.1. If the code indicates a pattern match, the original word is recovered by

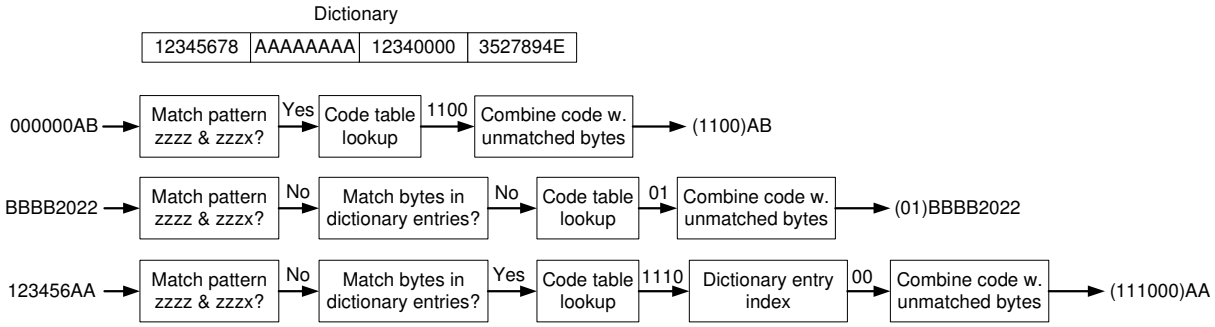

FIGURE 6.6. Compression examples for different input words.

combining zeroes and unmatched bytes, if any. Otherwise, the decompression output is given by combining bytes from the input word with bytes from dictionary entries, if the code indicates a dictionary match.

The C-Pack algorithm is designed specifically for hardware implementation. It takes advantage of simultaneous comparison of an input word with multiple potential patterns and dictionary entries. This allows rapid execution with good compression ratio in a hardware implementation, but may not be suitable for a software implementation. In general, software must process operations sequentially. For example, matching against multiple patterns can be prohibitively expensive for software implementations when the number of patterns or dictionary entries is large. C-Pack's inherently parallel design allows an efficient hardware implementation, in which pattern matching, dictionary matching, and processing multiple words are all done simultaneously. In addition, we chose various design parameters, such as dictionary replacement policy and coding scheme, to reduce hardware complexity, even if our choices slightly degrades the effective system-wide compression ratio.

In the proposed implementation of C-Pack, two words are processed in parallel per cycle. Achieving this, while still permitting an accurate dictionary match for the second word, is challenging. Let us consider compressing two similar words that have not been encountered by the compression algorithm recently, assuming the dictionary uses first-in first-out (FIFO) as its replacement policy. The appropriate dictionary content when processing the second word depends on whether the first word matched a static pattern. If so, the first word will not appear in the dictionary. Otherwise, it will be in the dictionary, and its presence can be used to encode the second word. Therefore, the second word should be compared with the first word and all but the first dictionary entry in parallel. This improves compression ratio as opposed to a more naive approach of not checking with the first word. Therefore, the proposed algorithm can compress two words in parallel without degrading compression ratio.

6.7. Adaptive On-Chip Cache Compression

As previously mentioned, cache compression should be adaptive because applications that can fit in the original cache can be hurt by the decompression overhead. This indicates that compression should only be used when the running application may benefit from a larger cache size. In this section, we study the performance-sensitivity to cache size of different workloads and propose a general metric to obtain such information on-line.

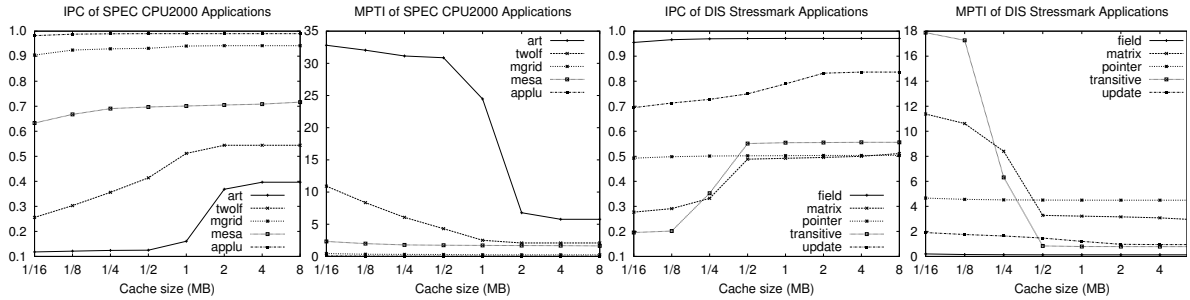

FIGURE 6.7. Performance and miss ratio of representative SPEC CPU2000 benchmarks and DIS Stressmarks.

6.7.1. Marginal Performance Gain

We use *marginal performance gain* as a measure of the usefulness of cache space for an application, or more specifically, the reduction in cycles per instruction (CPI) as a result of an incremental change to cache size. The benefit of an increase in the effective cache size depends largely upon the current running application and the current cache size. Figure 6.7 illustrates the impact of L2 cache size on performance as well as miss rate for ten representative applications from the SPEC CPU2000 benchmark suite and the Data-Intensive System (DIS) stressmark suite [**73**].

For a fixed number of instructions, the marginal performance gain at a given cache size represents the number of cycles saved executing the same number of instructions, if one more cache block were available. Thus, it indicates the benefit of increasing cache size from c to $c + 1$ for an application, or approximately the loss from decreasing cache size from c to $c - 1$ (considering that the marginal performance gain curve is smooth). Whenever the least recently used (LRU) line of processor i is accessed, if the number of cache lines of this processor is reduced by one, this access will be a miss. Thus, the number of cycles necessary to finish this

instruction will be increased by the miss penalty, MP_i. Therefore, the performance gain of process i can be estimated as:

$$g_i(c) = \frac{d}{dc}CPI_i(c) = \frac{CycleCount_i(c-1) - CycleCount_i(c)}{InstructionCount_i(c)} = \frac{LRURefCount_i(c) \cdot MP_i}{InstructionCount_i(c)} \tag{2}$$

As described in Section 6.3, MP_i is uniform across all processes, and is the off-chip memory access latency. We therefore define the marginal performance gain of process i as:

$$MPG_i(c) = \frac{LRURefCount_i(c)}{InstructionCount_i(c)} \tag{3}$$

Below, we describe the necessary hardware assistance to obtain runtime marginal performance gain. Over a time period T, an approximation of $MPG(c)$ can be obtained using two counters. The first counts the references to the LRU line and the second counts the number of instructions executed. For set-associative caches, LRU ordering is kept within each set. Therefore, we use set LRU to approximate global LRU, similar to Suh et al. [**74**]. Efficient hardware implementations of approximated LRU replacement for cache blocks are described in the literature [**75**]. When an LRU line in any set is accessed, the LRU counter is incremented. During each time period T, each processor calculates its marginal performance gain $MPG(c)$ by dividing the value of the two counters, and stores the result in a special register. The counters are then reset.

The marginal gain values of processors are also updated in the on-chip directory node. When the directory node receives a cache migration request from a processor, it selects the

processor with lowest marginal performance gain as the target processor and forwards the migration request to it. If the marginal performance gain of the requestor is lower than those of the remaining processors, the migration request is declined and the line is sent to off-chip memory, if modified. Once a processor receives a migration request from the directory node, it must place the compressed line in its own cache, even if this forces the eviction of one of its own cache lines.

We now describe how to decide update period T. A small T increases the accuracy of marginal performance gain estimation but imposes higher computational overhead. A large T reduces the computational overhead but may cause inaccurate estimation due to application phase changes being missed. Moreover, in a multitasking OS, T must be smaller than the context switching period, which is typically within the range of 100 ms to 200 ms. When developing their shared cache partitioning technique, Suh et al. [**74**] also encountered the problem of selecting an appropriate repartitioning period. They experimented with different repartitioning periods and found that both 5 million cycles and 10 million cycles permitted accurate identification of program phase changes with low overhead. We therefore used an update period of 10 million cycles. Note that at a CPU frequency of 1 GHz, 10 million cycles (10 ms) is significantly shorter than the context switch period. In order to control the recalculation of marginal performance gain, an OS timer may be used to interrupt the processor and switch to kernel mode every time period T. The additional overhead of the context switch is usually less than 10 µs for modern processors, which is negligible compared to the recalculation period.

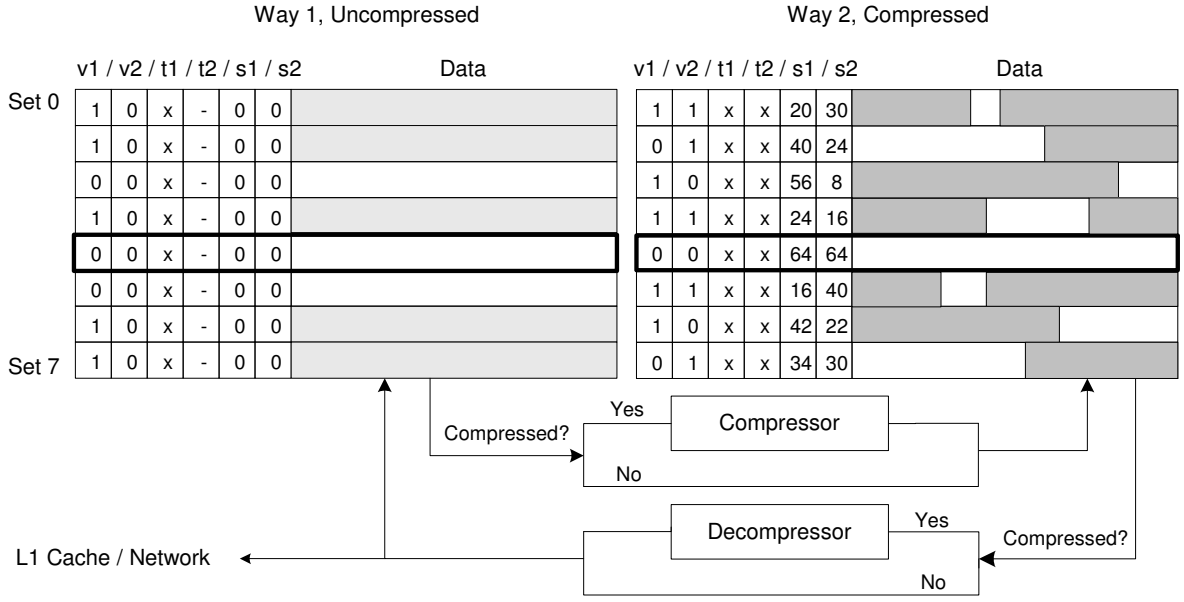

FIGURE 6.8. Compressed cache organization.

6.7.2. Adaptive Compression Policy

We use marginal performance gain to compress cache lines only when the running application may benefit from a larger cache size. The procedure of adaptive compression operates as follows. Two marginal performance gain thresholds are used to control adaption: MPG_{T1} and MPG_{T2}. In Figure 6.7.2, we give a simple example of a 2-way set-associative cache. A cache with higher associativity would be organized into two multi-way hierarchies.

1. When the marginal performance gain, mpg, of the running application is lower than the first threshold MPG_{T1}, neither cache way is compressed. However, a newly-loaded line is always placed in way-1 instead of way-2. Evicted lines from way-1 are placed in way-2 instead of off-chip memory. Evicted lines from way-2 are evicted off-chip. One can view way-2 as another level of cache hierarchy with the same access time as a regular L2 cache.

2. When mpg exceeds the first threshold MPG_{T1}, the compression of way-2 starts. Note that way-2 stores older data than way-1. Whenever a new line is sent to way-2, it is compressed, replacing a stale uncompressed line. The additional space left in the data block will later be used to store another compressed line. At this point, way-2 is split into two compressed ways. When mpg further exceeds the second threshold MPG_{T2}, way-1 is also compressed, following the same procedure.

3. When mpg decreases below MPG_{T2}, the decompression of way-1 starts. Whenever any compressed line is accessed, it is decompressed, evicting its partner compressed line, if present. When mpg falls below MPG_{T1}, decompression of way-2 begins, following the same procedure.

6.8. Off-Chip Memory Link Compression

To reduce off-chip memory access latency, and increase effective processor-memory bus bandwidth, we propose to transfer compressed cache lines between on-chip caches and off-chip memory controller. Therefore, both the on-chip memory interface and the off-chip memory controller must support variable-length compressed message formats. In addition, the message header contains a length field indicating size of the compressed cache line.

Although memory compression has been proposed to increase effective memory capacity, our objective is different: we aim to reduce off-chip memory access latency and processor-memory bus bandwidth requirement. Therefore, we propose a simple memory interface that cannot increase the effective memory capacity, but imposes little hardware overhead. Each

cache line is stored in memory in uncompressed or C-Pack compressed format, with one bit encoded in the ECC to indicate whether the line is compressed or not, which would prevent memory from operating in the ECC mode. Or, a small RAM can be used to keep the compressed bits for all cache lines.

6.9. Evaluation

This section presents the evaluation results of the proposed cache and memory compression techniques. We first present the hardware synthesis results of C-Pack hardware compression algorithm, and compare its raw and effective system-wide compression ratio with existing cache compression hardware. Then, we present the full-system simulation results of the compressed cache and memory hierarchy.

6.9.1. C-Pack Synthesis Results

In this section, we present the performance, power consumption, and area overheads of the compression/decompression hardware when synthesized for integration within a microprocessor. We synthesized our design using Synopsys Design Compiler with 180 nm, 90 nm, and 65 nm libraries. Table 6.2 presents the resulting performance, area, and power consumption at maximum internal frequency. "Loc" refers to the compressed line locator/arbitrator in a pair-matching compressed cache and "worst case delay" refers to the number of cycles required to compress, decompress, or locate a 64 B line in the worst case. As indicated in Table 6.2, the proposed hardware design achieves a throughput of 80 Gb/s (64 B $\times$ 1.25 GHz) for compression and 76.8 Gb/s (64 B $\times$ 1.20 GHz) for decompression in a 65 nm technology. Its area and

TABLE 6.2. Synopsys Design Compiler Synthesis Results

Parameters	180 nm			90 nm			65 nm		
	Comp.	Deco.	Loc.	Comp.	Deco.	Loc.	Comp.	Deco.	Loc.
Worst case delay (cycles)	13	8	2	13	8	2	13	8	2
Max. frequency (GHz)	0.38	0.31	0.60	1.09	0.91	1.79	1.25	1.20	2.00
Area (mm^2)	0.34	0.25	0.063	0.076	0.076	0.013	0.043	0.043	0.007
Power (mW)	111.78	75.18	110.03	73.88	51.50	15.96	32.63	24.14	5.20

power consumption overheads are low enough for practical use. The total power consumption of the compressor, decompressor, and compressed line arbitrator at 1 GHz is 48.82 mW (32.63 mW/1.25 GHz + 24.14 mW/1.20 GHz + 5.20 mW/2.00 GHz) in a 65 nm technology. This is only 7% of the total power consumption of a 512 KB cache with a 64 B block size at 1 GHz in 65 nm technology, derived using CACTI 5 [**76**].

6.9.2. Comparing C-Pack Compression Ratio with Literature

We compare C-Pack to several other hardware compression designs, namely X-Match, FPC, and MXT, that may be considered for cache compression. We exclude other compression algorithms because they either have not been implemented in hardware or are not suitable for cache compression. Although the proposed hardware implementation mainly targets online cache compression, it can also be used in other high-performance lossless data compression applications with few or no changes.

We tested the compression ratios of different algorithms on four cache data traces gathered from a full-system simulation of various workloads from the Mediabench [**37**] and SPEC CPU2000 benchmark suites. The block size and the dictionary size are both set to 64 B in all test cases. Since we are unable to determine or get access to the exact compression algorithm

TABLE 6.3. Compression Ratio Comparison

Benchmark	Raw compression ratio (%)				System-wide compression ratio (%)			
	MXT	FPC	X-Match	C-Pack	MXT	FPC	X-Match	C-Pack
mpeg2	70.88	63.39	49.50	52.10	75.55	64.28	57.97	58.47
mesa	49.50	69.81	42.80	51.97	60.50	66.18	53.59	55.80
art	57.69	59.27	46.60	51.74	64.84	66.67	60.63	61.40
twolf	84.09	80.73	70.20	77.40	85.90	75.60	62.37	69.92
average	65.54	68.30	52.28	58.30	71.70	68.18	58.64	61.40

used in MXT, we used the LZSS Lempel-Ziv compression algorithm to approximate its compression ratio [**77**]. The raw compression ratios and effective system-wide compression ratios in a pair-matching scheme are summarized in Table 6.3. Each row shows the raw compression ratios and effective system-wide compression ratios using different compression algorithms for an application. As indicated in Table 6.3, raw compression ratio varies from algorithm to algorithm, with X-Match being the best and MXT being the worst on average. The poor raw compression ratios of MXT are mainly due to its limited dictionary size. The same trend is seen for effective system-wide compression ratios: X-Match has the lowest (best) and MXT has the highest (worst) effective system-wide compression ratio. Since the raw compression ratios of X-Match and C-Pack are close to 50%, they achieve better effective system-wide compression ratios than MXT and FPC. On average, the system-wide compression ratio of C-Pack is 4.7% worse than that of X-Match, 9.9% better than that of FPC, and 14.4% better than that of MXT.

6.9.3. Simulation Results

In this section, we present full-system simulation results of the proposed adaptive cache and memory compression techniques.

1
TABLE 6.4. Parameters in Simulations

Parameter	Value
Processor	In-order, 1 GHz
L1 I/D cache	32 KB, 2-way, 64 B line size, 2 cycles
L2 cache	8-way, 64 B line, 10 cycles private, 30 cycles shared
Off-chip memory	80 cycles, 10 bytes per cycle
On-chip network	Point-to-point, 5 cycles/hop, 100 bytes/cycle

For adaptive cache compression, there are only performance benefits when the following two requirements are both met: (1) the current cache size is a performance sensitive point, and (2) the application has good compression ratio. For all other cases, cache compression cannot improve the performance of the application. In Section 6.9.3.2 and Section 6.9.3.3, we evaluate the applicability of adaptive cache compression on the benchmarks by evaluating their compression ratio and performance-sensitivity to cache size. In Section 6.9.3.4, we present the evaluation results of the adaptive cache compression and main memory link compression techniques.

6.9.3.1. *Simulation Environment and Workloads*

We used the Simics [**78**] full-system simulator and the Ruby module from the GEMS infrastructure [**79**] to simulate the detailed memory system and interconnect network. Our simulator is based on the Simics/Aurora SPARC Linux, which models the SPARC V9 architecture in sufficient detail to run a Linux 2.6.13 SMP kernel. We used the in-order processor model provided by Simics mainly to reduce simulation time. For all our simulations, the L1 cache is split and

the L2 cache is unified. Inclusion is maintained between L1 and L2 caches. The architectural parameters used in the simulations are listed in Table 6.4.

To evaluate the cache and memory compression techniques, we used five commonly-accepted benchmark suites, including single-threaded SPEC CPU2000 benchmarks and Data-Intensive Systems (DIS) stressmarks [**73**] as well as the multi-threaded SPEC OMP v3.2, Nasa Parallel Benchmarks [**80**] (NPB) v3.2, and ALPBench benchmarks [**81**].

For single-threaded SPECOMP and DIS workloads, we fast forward to the "early single" SimPoint [**82**], warm up for 100 million instructions, and then measure performance for the next 100 million instructions. For multithreaded workloads, we set the number of threads to four. For SPEC OMP benchmarks, we use the medium versions and the reference inputs. The central part of these benchmarks is a major loop whose body contains code executed by multiple threads in parallel. We first fast forward to the major loop of the benchmarks, then warm up the cache with 100 million instructions, and finally measure performance in the following 100 million instructions. For NPB benchmarks, we use the problem class S, warm up the cache with 100 million instructions, and then run to completion and measure the performance. For ALPBench multimedia benchmarks, we fast forward to the main media processing function, warm up the cache with 100 million instructions, and measure performance in the following 100 million instructions.

6.9.3.2. *Workload Compression Ratio*

To understand the compressibility of the benchmarks, we dumped the contents of L2 cache during simulation, and compressed them using C-Pack. Figure 6.9 illustrates the average

raw compression ratio and effective system-wide compression ratio of all seventeen SPEC CPU2000, six DIS, six SPECOMP, ten NPB, and five ALPBench benchmarks that we were able to compile and run on our simulation target.

We calculated the percentage of applications with an effective system-wide compression ratio below 60% for all benchmark suites. The percentages are 41% for SPEC CPU2000, 50% for DIS, 0% for SPECOMP, 10% for NPB, 0% for ALPBench, and 25% for all benchmarks. The compressibility of the evaluated benchmarks are not very good. One possible reason is that many single-threaded and multi-threaded scientific computing applications require a large number of floating-point operations. Compressing floating point data is known to be a difficult problem. For incompressible applications, cache compression cannot help improve performance. However, a threshold can be defined to prevent the unnecessary overhead of compressing incompressible data. In our simulations, cache lines that with compression ratios above 80% is not stored in compressed format.

6.9.3.3. *Workload Performance-Sensitivity to Cache Size*

Since cache compression may only improve the performance of applications that are performance-sensitive to cache size, we evaluated the performance-sensitivity to cache size of all available benchmarks at six cache size configurations. The results are presented in Table 6.5, Table 6.6, and Table 6.7.

We observe that a significant reduction in miss ratio does not always translate to a significant improvement in performance. Moreover, for most applications, there are only a few cache sizes from which an increase results in significant performance improvement. This is due to the

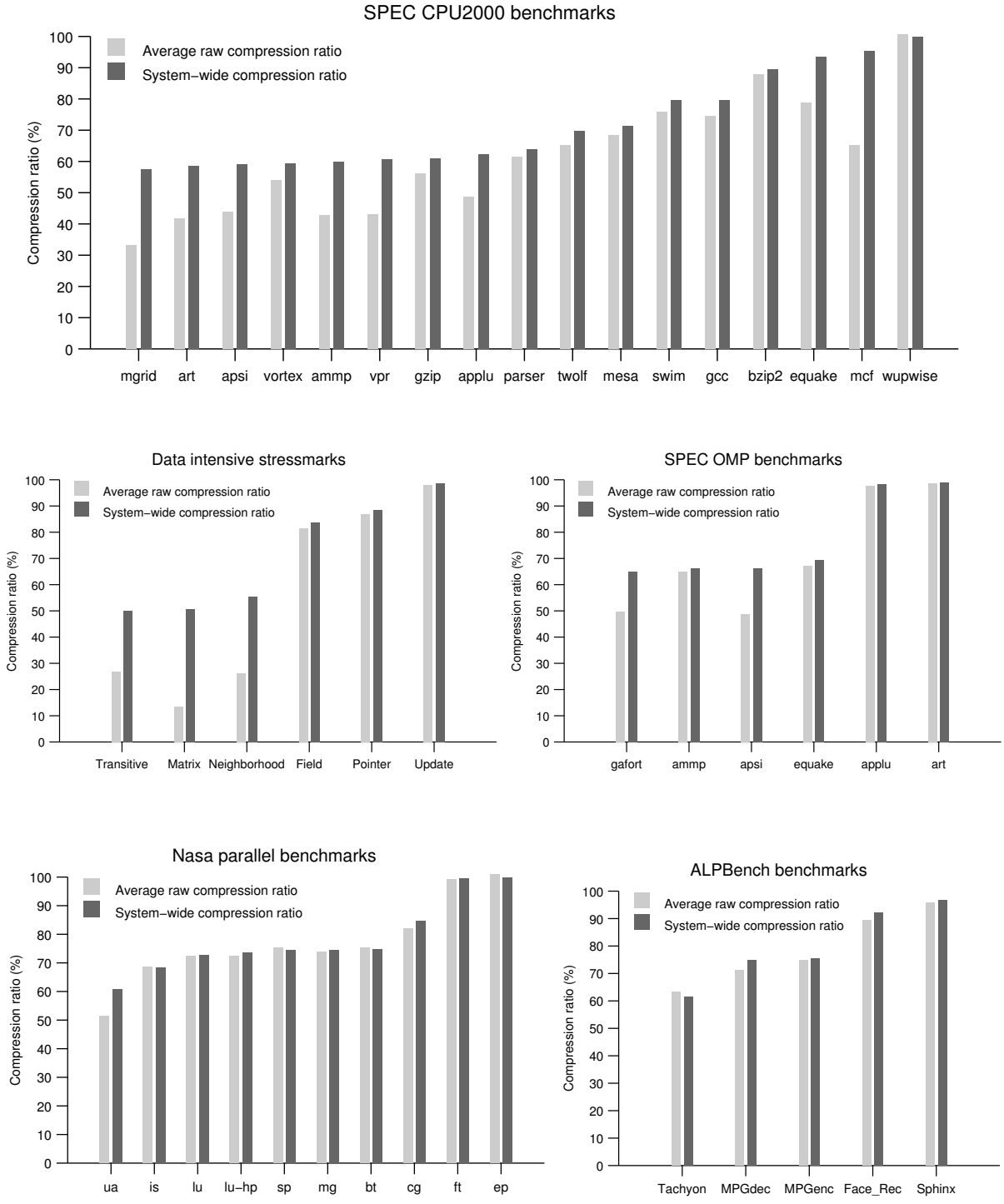

FIGURE 6.9. Raw and system-wide effective compression ratios of benchmarks.

working data set constraint of the applications. For cache compression to improve performance, the current cache size has to be at a sensitive point of the performance-cache curve, i.e., doubling the current cache size would result in a significant higher performance. Of all 44 benchmarks, 19 have the same performance[1] at all evaluated cache sizes, 10 have one size sensitive point, and 15 have multiple size sensitive points. The maximum performance improvement when cache size is doubled, across all applications and all cache sizes, is 131.3%; the average performance improvement is 6.5%.

These results further demonstrate the importance of adaptation of cache compression. When the currently-running application is not performance-sensitive at the current cache size, compression should not be performed, thus avoiding unnecessary overheads. We will show in the next section that the adaptive cache compression scheme controlled by marginal performance gain only imposes small overhead (on average 2.39%) on applications that are not performance-sensitive to cache sizes.

6.9.3.4. *Performance of Cache Compression and Memory Compression*

In this section, we present the evaluation results of the adaptive cache compression and main memory link compression techniques. To show the performance improvements of cache and main memory compression, we evaluated eight benchmarks that satisfy both requirements. The IPC and miss ratio values of these benchmarks are compared under four different techniques: the original system (none), adaptive cache compression (cc), off-chip memory link

[1]The standard deviation of IPCs at different cache sizes is within 1% of the average IPC.

TABLE 6.5. SPEC2000 Benchmark Sensitivity to Cache Size

	art	twolf	mcf	applu	bzip2	gcc	gzip	mesa	mgrid	parser	swim	vortex	vpr	apsi	equake	wupwise	ammp
Instructions per cycle (IPC)																	
256 KB	0.13	0.36	0.11	0.99	0.71	0.25	0.81	0.86	1.00	0.72	0.90	0.99	1.00	0.83	0.54	0.84	0.99
512 KB	0.13	0.43	0.12	1.00	0.77	0.35	0.84	0.86	1.00	0.78	0.90	0.99	1.00	0.83	0.55	0.85	0.99
1 MB	0.16	0.54	0.12	1.00	0.86	0.36	0.85	0.86	1.00	0.79	0.90	0.99	1.00	0.83	0.55	0.85	0.99
2 MB	0.37	0.58	0.15	1.00	0.87	0.37	0.85	0.87	1.00	0.80	0.90	0.99	1.00	0.83	0.56	0.85	0.99
4 MB	0.41	0.58	0.24	1.00	0.87	0.37	0.85	0.87	1.00	0.80	0.91	0.99	1.00	0.84	0.57	0.86	0.99
8 MB	0.41	0.58	0.30	1.00	0.87	0.37	0.85	0.88	1.00	0.80	0.91	0.99	1.00	0.84	0.58	0.87	0.99
L2 misses per thousand instructions (MPTI)																	
256 KB	30.78	6.06	34.27	0.04	1.72	7.70	1.29	0.57	0.00	1.31	0.45	0.04	0.02	1.08	3.73	0.80	0.03
512 KB	30.53	4.16	32.88	0.02	1.08	1.32	1.13	0.54	0.00	0.72	0.43	0.04	0.01	1.07	3.67	0.76	0.03
1 MB	24.31	2.26	30.89	0.02	0.37	0.93	1.10	0.54	0.00	0.62	0.42	0.04	0.01	1.07	3.64	0.75	0.03
2 MB	6.72	1.85	25.95	0.02	0.30	0.79	1.08	0.52	0.00	0.58	0.41	0.04	0.01	1.07	3.58	0.74	0.03
4 MB	5.60	1.85	16.23	0.02	0.30	0.78	1.08	0.50	0.00	0.58	0.41	0.04	0.01	1.06	3.48	0.67	0.03
8 MB	5.60	1.85	11.84	0.02	0.30	0.78	1.08	0.46	0.00	0.58	0.41	0.04	0.01	1.06	3.32	0.61	0.03

TABLE 6.6. DIS (a) and SPEC OMP (b) Benchmarks Sensitivity to Cache Size

	Matrix	Update	Transitive	Pointer	Neighborhood	Field
Instructions per cycle (IPC)						
256 KB	0.42	1.84	0.54	1.00	1.53	1.95
512 KB	0.88	1.84	1.08	1.00	1.54	1.95
1 MB	0.88	1.85	1.10	1.00	1.55	1.95
2 MB	0.88	1.85	1.11	1.01	1.55	1.95
4 MB	0.88	1.85	1.11	1.01	1.55	1.95
8 MB	0.88	1.85	1.12	1.01	1.55	1.95
L2 misses per thousand instructions (MPTI)						
256 KB	15.63	0.36	10.84	4.55	1.30	0.12
512 KB	2.50	0.34	0.97	4.53	1.25	0.11
1 MB	2.45	0.33	0.80	4.51	1.23	0.11
2 MB	2.43	0.33	0.76	4.49	1.22	0.11
4 MB	2.41	0.33	0.74	4.48	1.22	0.11
8 MB	2.36	0.33	0.73	4.48	1.21	0.11

	ammp	art	apsi	equake	gafort	applu
Instructions per cycle (IPC)						
256 KB	0.20	0.11	0.81	0.47	0.90	0.08
512 KB	0.26	0.12	0.82	0.50	0.91	0.08
1 MB	0.39	0.12	0.82	0.53	0.91	0.08
2 MB	0.46	0.12	0.84	0.55	0.91	0.08
4 MB	0.47	0.12	0.85	0.56	0.91	0.08
8 MB	0.47	0.13	0.85	0.57	0.91	0.08
L2 misses per thousand instructions (MPTI)						
256 KB	17.92	33.84	0.62	5.51	0.44	53.89
512 KB	12.34	32.82	0.58	4.99	0.43	52.69
1 MB	8.05	32.36	0.54	4.64	0.42	52.04
2 MB	6.50	32.14	0.40	4.38	0.42	51.67
4 MB	6.40	32.03	0.33	4.20	0.41	51.45
8 MB	6.37	30.11	0.32	4.08	0.41	51.41

TABLE 6.7. NPB (a) and ALPBench (b) Benchmarks Sensitivity to Cache Size

	cg	ua	ep	bt	ft	is	lu	lu-hp	sp	mg
Instructions per cycle (IPC)										
256 KB	0.21	0.52	0.70	0.63	0.26	1.00	0.99	0.43	0.36	0.99
512 KB	0.22	0.57	0.70	0.77	0.26	0.99	0.99	0.47	0.50	0.99
1 MB	0.39	0.58	0.72	0.78	0.26	0.99	0.99	0.48	0.51	1.00
2 MB	0.54	0.58	0.74	0.78	0.26	1.00	1.00	0.48	0.51	1.00
4 MB	0.54	0.58	0.86	0.78	0.28	1.00	1.00	0.48	0.51	1.00
8 MB	0.54	0.58	0.90	0.78	0.32	1.00	1.00	0.48	0.51	1.00
L2 misses per thousand instructions (MPTI)										
256 KB	16.52	4.62	1.87	2.53	17.01	0.02	0.04	6.98	8.48	0.03
512 KB	15.47	3.91	1.82	1.38	16.85	0.03	0.04	6.15	5.66	0.04
1 MB	5.05	3.79	1.63	1.26	16.74	0.02	0.03	5.98	5.46	0.02
2 MB	1.46	3.76	1.53	1.28	16.63	0.02	0.03	6.03	5.43	0.02
4 MB	1.45	3.75	1.02	1.28	15.87	0.02	0.02	6.03	5.43	0.02
8 MB	1.45	3.75	0.88	1.28	14.45	0.02	0.02	6.03	5.43	0.02

	Tachyon	Sphinx	MPGenc	MPGdec	Face_Rec
Instructions per cycle (IPC)					
256 KB	0.82	0.23	0.88	0.90	0.14
512 KB	0.83	0.24	0.90	0.90	0.23
1 MB	0.84	0.25	0.92	0.91	0.23
2 MB	0.84	0.25	0.92	0.91	0.23
4 MB	0.84	0.30	0.92	0.91	0.23
8 MB	0.84	0.50	0.92	0.91	0.23
L2 misses per thousand instructions (MPTI)					
256 KB	0.44	14.25	0.45	0.14	26.89
512 KB	0.33	12.93	0.31	0.12	13.27
1 MB	0.29	12.42	0.21	0.10	13.12
2 MB	0.27	12.18	0.20	0.09	12.93
4 MB	0.27	8.90	0.19	0.09	12.86
8 MB	0.27	2.25	0.19	0.09	12.84

TABLE 6.8. Instruction Per Cycle

	none	cc	mc	both	none	cc	mc	both	none	cc	mc	both	none	cc	mc	both
	SPEC2000 art				SPEC2000 twolf				DIS matrix				DIS transitive			
256 KB	0.25	0.24	0.36	0.36	0.72	0.70	0.79	0.78	0.42	0.76	0.72	0.87	0.54	0.96	0.78	1.01
512 KB	0.25	0.28	0.36	0.40	0.86	0.83	0.91	0.88	0.88	0.88	1.01	1.01	1.08	1.08	1.15	1.15
1 MB	0.32	0.59	0.44	0.63	1.08	1.08	1.10	1.10	0.88	0.88	1.01	1.01	1.10	1.10	1.16	1.16
2 MB	0.74	0.73	0.78	0.78	1.15	1.15	1.16	1.15	0.88	0.88	1.01	1.01	1.11	1.10	1.16	1.16
	SPEC OMP ammp				SPEC OMP equake				NPB cg				NPB sp			
256 KB	0.39	0.42	0.45	0.50	0.93	0.92	1.00	1.00	0.42	0.42	0.44	0.44	0.72	0.73	0.77	0.75
512 KB	0.52	0.61	0.56	0.65	1.00	1.00	1.06	1.06	0.44	0.42	0.46	0.46	1.00	0.99	1.01	1.01
1 MB	0.77	0.77	0.80	0.80	1.05	1.05	1.11	1.11	0.78	0.86	0.79	0.86	1.02	1.02	1.02	1.02
2 MB	0.91	0.91	0.92	0.92	1.09	1.09	1.15	1.15	1.07	1.07	1.07	1.07	1.02	1.02	1.02	1.02

TABLE 6.9. L2 Misses Per Thousand Instructions

	none	cc	mc	both	none	cc	mc	both	none	cc	mc	both	none	cc	mc	both
	SPEC2000 art				SPEC2000 twolf				DIS matrix				DIS transitive			
256 KB	30.78	30.49	31.14	30.82	6.06	5.15	6.05	5.14	15.63	2.52	15.05	2.52	10.84	0.82	10.86	0.82
512 KB	30.53	26.84	30.77	26.95	4.16	3.29	4.18	3.33	2.50	2.49	2.46	2.50	0.97	0.97	0.97	0.97
1 MB	24.31	7.78	24.40	7.79	2.26	2.25	2.26	2.26	2.45	2.46	2.40	2.43	0.80	0.79	0.80	0.80
2 MB	6.72	6.70	6.70	6.70	1.85	1.85	1.85	1.85	2.43	2.43	2.39	2.41	0.76	0.76	0.76	0.76
	SPEC OMP ammp				SPEC OMP equake				NPB cg				NPB sp			
256 KB	17.92	14.73	17.92	14.57	5.51	5.49	5.53	5.52	16.52	16.51	16.52	16.52	8.48	7.26	8.26	7.27
512 KB	12.34	9.19	13.32	9.61	4.99	4.92	5.02	5.01	15.47	15.43	15.47	15.47	5.66	5.63	5.59	5.64
1 MB	8.05	8.04	8.12	8.07	4.64	4.66	4.66	4.66	5.05	2.24	5.10	2.24	5.46	5.46	5.45	5.46
2 MB	6.50	6.38	6.49	6.40	4.38	4.38	4.38	4.38	1.46	1.46	1.46	1.46	5.43	5.47	5.45	5.47

compression (mc), and combining both adaptive cache compression and memory link compression (both). The performance (IPC) and miss ratio (L2 misses per thousand instructions) of the eight evaluated benchmarks are presented in Table 6.8 and Table 6.9, respectively.

When comparing the performance of adaptive cache compression and memory link compression, we make the following observations:

- Out of the total 32 cases (8 benchmarks, 4 cache sizes each), adaptive cache compression improves performance in 8 cases. The improvement ranges from 1.39% to

84.38%, and is on average 36.47%. Adaptive cache compression also slightly degrades performance in 8 cases. The degradation ranges from 0.9% to 4.55%, and is on average 2.39%. In all other cases, performance in not affected.

- Out of the total 32 cases, memory link compression improves performance in 29 cases. The improvement ranges from 0.87% to 71.43%, and is on average 13.69%. In all other cases, performance in not affected.
- Out of the total 32 cases, using both adaptive cache compression and memory link compression improves performance in 28 cases. The improvement ranges from 1.00% to 107.14%, and is on average 20.76%. In all other cases, performance in not affected. For the 8 cases when performance is improved by using only adaptive cache compression, on average an additional 15.87% of improvement is achieved by combining memory link compression.

These results indicate that cache compression tends to have higher performance improvement than memory link compression, but only improves performance for 25% of the benchmarks. In contrast, although memory link compression does not provide as much performance improvement as cache compression, good results are shown on 87.5% of the benchmarks. When combined together, the two techniques achieve higher performance improvement than used alone, and also affect a wide range of applications. Our benchmarks are representative of scientific computing applications and multimedia applications, both single-threaded and multi-threaded. We believe there might be some other type of applications that might also benefit

from cache compression. For example, In the past, researchers have reported that cache compression can improve the performance memory-intensive commercial workloads [**56**] by up to 17%. However, we do not have access to such commercial benchmarks.

6.10. Conclusion

In this Chapter, we presented a unified, adaptive on-chip cache and off-chip memory compression system. We presented a new, efficient pair-matching scheme to organize compressed cache lines. We demonstrated that this scheme is as effective as conventional segmentation based approaches, but imposes much less hardware overhead. Unlike most previous work on cache compression that made assumptions about a high-performance cache compression hardware, we designed a lossless compression algorithm for fast cache compression, and synthesized the corresponding hardware. We also evaluated the applicability of cache compression for a wide range of benchmark applications, and described two critical requirements that must be met for cache compression to improve performance. Unlike most previous work on cache compression, which picked one or a few cache sizes and several special applications to show the benefits of cache compression, we used full-system simulation to evaluate the compressibility and performance-sensitivity to cache of 45 applications from five well-known single-threaded and multi-threaded benchmark suites. For eight benchmarks that meet both requirements for cache compression, the average performance improvement is 36.47%. Off-chip memory link compression improves performance of the same set of benchmarks by 13.69% on average. We

also show that when used with off-chip memory link compression, further performance improvement is possible. Although the average improvement is 20.76%, such improvement resulted from a wider range of applications.

Chapter 7

Chip-Multiprocessor Cooperative Compression and Migration

CMPs are now in common use. Increasing core counts implies increasing demands for instruction and data. However, processor-memory bus throughput has not kept pace. As a result, cache misses are becoming increasingly expensive. Yet the performance benefits of using more cores limits the cache per core, and cache per process, increasing the importance of efficiently using cache.

To alleviate the increasing scarcity of on-chip cache, we propose a cooperative and adaptive data compression and migration technique and validate it with detailed hardware synthesis and full-system simulation. This is the first work that completes optimized microarchitectural design and synthesis of the hardware to support adaptive cache compression and migration. Full-system simulation on multiprogrammed and multithreaded workloads indicate that, for cache-sensitive applications, the maximum CMP throughput improvement with the proposed technique ranges from 4.7%–160% (on average 34.3%), relative to a conventional private L2

cache architecture. No performance penalty is imposed for cache-insensitive applications. Furthermore, we demonstrate that our cooperative techniques may influence the optimal core cache area ratio for maximum CMP throughput, and has the potential to reduce on-chip cache requirement, and thereby increase the number of cores.

7.1. Introduction and Motivation

Increases in VLSI integration density and the increasing importance of power consumption are leading to the use of chip-level multiprocessor (CMP) architectures. The move to CMPs substantially increases capacity pressure on the on-chip memory hierarchy. The 2006 ITRS Roadmap [**3**] predicts that transistor speed will continue to grow faster than DRAM speed and pin bandwidth. This means that cache miss costs will continue to increase. However, increases in on-chip cache size may block the addition of processor cores, thereby resulting in suboptimal CMP throughput. Our full-system simulation results indicate that under area constraints, the allocation of core-cache area that achieves maximum CMP throughput use less cache per core than is common for existing processors. However, such allocation increases capacity misses, and thereby reduces the performance of individual cores. As a result, the marginal improvement in performance as a result of increasing usable cache size increases. In other words, *the move to CMPs increases the importance of carefully using available cache area.*

In this chapter, we address the problem of optimizing the use of on-chip cache to reduce off-chip memory access, and thereby improve CMP throughput. We develop and evaluate our techniques for private on-chip L2 cache hierarchy, because in contrast to a shared L2 cache,

the performance and design styles of private L2 caches remain consistent and predictable when the number of processor cores increases. The techniques proposed in this chapter address the weakness of private caches by increasing their capacity without significantly increasing their access latency.

When data are evicted from a private L2 cache, they are generally discarded (if not modified) or transferred to the next level of the memory hierarchy, i.e., off-chip memory. Writing to off-chip memory can cause contention with other accesses to the processor-memory bus and result in significant latency when later reading back the same data for reuse. In CMPs, lower-overhead alternatives should be considered, such as compressing data and storing them locally, migrating data to the private L2 caches of other processor cores, or a combination of these techniques. One can view compression and migration as introducing a new, virtual layer in the memory hierarchy. This new layer permits an increase in usable L2 cache without increasing physical L2 cache size, at a cost of higher access latency.

We propose a method of alleviating the increasing scarcity of on-chip memory via cooperative and adaptive cache compression and migration. Both of these techniques introduce a number of design problems, and their cooperative use compounds these problems. We propose a unifying adaptive policy that considers the time-varying merits of the workloads. Defining this compression–migration policy and analyzing its hardware implications are the main foci of our work.

7.2. Contributions

This work is the first to cooperatively use cache compression and migration for CMPs. We propose an adaptive control policy integrating the two techniques and permitting run-time adaptation to workload. This control policy is based on a new optimization metric: processor marginal performance gain. Compared to static techniques, our adaptive techniques allow on average 42.3% greater performance improvement for cache-sensitive applications and prevent performance degradation for cache-insensitive applications. We heavily modified the *MSI_MOSI_CMP_directory* cache coherence protocol provided by the GEMS infrastructure [**79**] to support cache compression and migration.

This work is the fist to present optimized microarchitectural design and synthesis results for the hardware required for control, compression/decompression, and migration. We propose a new scheme to organize compressed cache lines, namely pair-matching, that is more efficient than commonly accepted segmentation based schemes. We also present the first hardware implementation of the PBPM [**72**] compression/decompression algorithm. Compared to prevailing cache compression/decompression algorithms, our implementation provides the best overall performance, in terms of compression ratio, maximum frequency, decompression latency, power consumption, and area cost.

The proposed techniques are evaluated using full-system (application and OS) simulation of numerous multiprogrammed SPEC CPU2000 benchmarks and Data-Intensive Systems (DIS) stressmarks [**73**] and multithreaded SPEC OMP and Nasa Parallel Benchmarks. We found that

they can improve the throughput of CMPs by up to 160% for cache-sensitive applications. Consequently, given a fixed core count, our techniques can help reduce chip area by allowing cache size reduction. Alternatively, given a fixed area, our techniques allow more processor cores to be used while maintaining good performance for each core.

The rest of the chapter is organized as follows. Section 7.4 describes the proposed cooperative cache compression and migration techniques. Section 7.5 describes our simulation environment and workloads, and presents and analyzes the simulation results. Section 7.3 discusses related work. We conclude in Section 7.6.

7.3. Related Work

When exploring the design space for future CMPs, Huh, Burger, and Keckler [**83**] observed that off-chip bandwidth will likely limit the number of cores per die because transistor count is increasing much faster than the number of signaling pins. Li et al. [**84**] studied the tradeoff between number of cores and cache size under area constraints. Their results showed the challenges of accommodating both CPU-bound and memory-bound workloads in the same design. The proposed techniques provide the most benefit when used with such applications.

A number of cache compression [**56**, **57**] and memory compression [**8**, **7**] systems have been developed for improving single-processor memory system performance. However, few researchers have studied reducing off-chip communication cost using compression for CMPs. We are aware of only two recent articles on this topic. Ozturk et al. [**85**] proposed to compress data blocks for which reuse intervals are large using compile time optimization based on an

integer-linear programming formulation. This approach was evaluated using access patterns extracted from embedded applications. However, it has not been implemented or evaluated within a multiprocessor simulator: the hardware complexity and performance impact are not yet well understood. Alameldeen and Wood investigated L2 cache and link compression in CMPs [**58**]. Their CMP design assumed private L1 caches for each core and a shared L2 cache for all cores. However, we believe compression is more important for private L2 caches because they face more severe capacity issue than shared caches. Moreover, decompression adds overhead to the shared cache access latency, which is already much higher than private caches. Therefore, their technique is more useful for larger commercial benchmarks with huge cache requirements.

Suh et al. [**74**] proposed a dynamic technique to partition shared on-chip cache among processes. Their technique collects the cache miss characteristics of processes at run-time and uses this information to partition cache among cores. They defined their control metric to be the derivative of the cache miss curve. However, this definition has shortcomings and may produce suboptimal performance. Because the same reduction in miss rate may have different performance implications for different applications. For example, as shown in Figure 6.7, for application *art*, doubling the L2 cache size at 512 KB would result in six fewer misses per thousand instructions, and 0.05 absolute IPC improvement. Meanwhile, for application *twolf*, doubling the cache size at 512 KB would result in two fewer misses per thousand instructions, but a 0.1 absolute IPC improvement. Defining marginal performance gain as the reduction in cache misses implies that the benefit of increasing cache size for *art* is larger than *twolf*, which

is not correct. If one's goal is to improve performance rather than reduce miss, our definition of marginal performance gain should be used instead.

Quresi and Patt [**86**] improved upon Suh's dynamic cache partitioning scheme by separating the utility monitoring unit from the shared cache. However, they used reduction in misses to make cache partitioning decisions, the same metric as in Suh's work. Chang and Sohi [**87**] proposed cooperative caching to manage the distributed on-chip caches for CMPs. Their techniques include cache-to-cache transfers of clean data, replication-aware data replacement, and global replacement of inactive data. Zhang and Asanovic proposed victim replication [**88**] for private L2 caches, which attempts to keep copies of local primary cache victims within the local L2 cache. They later proposed victim migration [**89**] that improves on victim replication. Beckmann et al. proposed the adaptive selective replication (ASR) scheme [**90**], a more advanced replication scheme for private caches. Our migration technique is orthogonal to these replication techniques, because they make decisions on whether to duplicate a cache line fetched from a remote cache and we make decisions on what we should do with a useful cache line upon eviction. In fact, our adaptive compression and migration technique can be used together with any replication techniques, and is likely to improve their performance by replicating and transmitting compressed data.

7.4. Cooperative Cache Compression and Migration

This section describes the proposed cooperative cache compression and migration techniques for optimizing on-chip cache utilization. We first describe the problem definition for

1
TABLE 7.1. Comparison of Eviction, Compression, and Migration

Scheme	$\Delta cycles$
Eviction	$P \cdot Miss\_Penalty_{L2}$
Compression	$(P + P') \cdot Decompress\_Penalty$
Migration	$P \cdot Remote\_Latency_{L2} + P'' \cdot Miss\_Penalty_{L2}$

optimizing on-chip cache utilization, then give an overview of the proposed architecture. Finally, we present the details of our technique to control adaptive compression and migration.

7.4.1. Optimizing On-Chip Cache Utilization

Our objective is to improve the overall throughput of CMPs by optimizing the utilization of on-chip cache resources. To that end, we consider two methods: (1) compressing data stored in its local cache or (2) making use of the caches of other cores.

Designing the architectural extensions required to support either compression or migration is challenging, and using the two techniques cooperatively complicates the problem. It is important to select the right lines to compress/migrate, determine the right moment to compress/migrate, and decide the locations of cache lines after compression/migration. In addition, both compression and migration involve overhead and should only be used when the CMP throughput can be improved. Furthermore, migration increases on-chip network traffic and may result in contention if used improperly: only "useful" data should be migrated. Therefore, the overall technique must be able to determine the time-varying cache requirement of the workload during execution and must adaptively control compression and migration.

To further understand the costs associated with compression and migration, Table 7.1 compares the first-order effect on total CPU cycle count when a L2 cache line is (1) evicted off-chip, (2) compressed locally, and (3) migrated to a remote L2 cache. For compression, in order to have the same effect as eviction, two lines must be compressed and placed in one cache line. We call that compressed line the *compression victim line*. For migration, in order to accept the migrated block, a local cache block must be evicted, which we call the *migration victim line*. P, P', and P'' represent the probability of the cache line, the compression victim line, and the migration victim line being accessed again, respectively. To simplify illustration, we assume $P' \approx P''$. Therefore, as long as $Decompress\_Penalty < Remote\_Latency_{L2}$, the penalty of compression is always smaller than migration. In modern CMP cache hierarchies, the $Remote\_Latency_{L2}$ is usually greater than 15 cycles, while the $Decompress\_Penalty$ of our proposed hardware is only 8 cycles. Therefore we conclude that it is always more beneficial to compress than to migrate.

7.4.2. Overview of Proposed Solution

We propose a unified solution that uses compression and migration to cooperatively and adaptively manage on-chip cache resource in CMPs. In our technique, when the running process can benefit from a larger cache size, the least recently used (LRU) lines in local L2 cache are compressed. Depending on the sensitivity of the application to cache size, the number of cache lines being compressed may continue to grow until all lines are compressed. After all the lines in the local cache are compressed, if the running process would benefit from a further increased cache size more than the processes on other cores, then the least recently used compressed lines

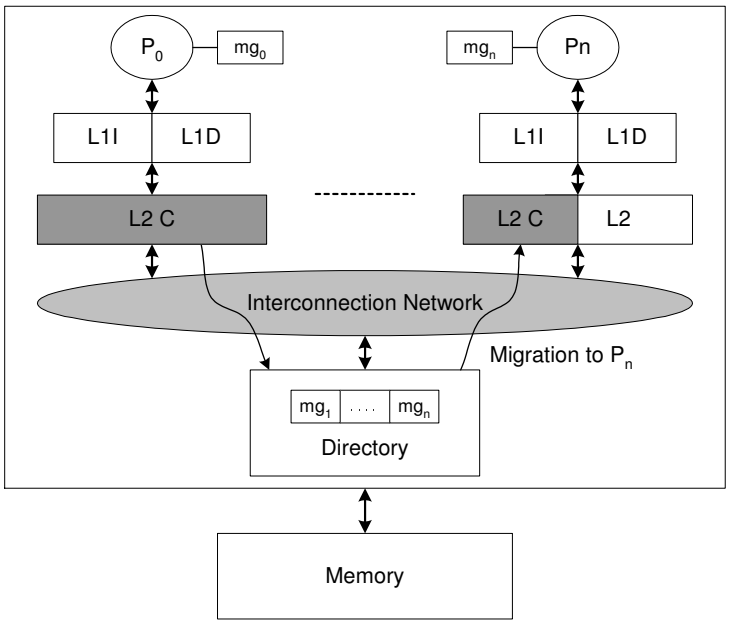

FIGURE 7.1. Technique overview.

are migrated to remote on-chip caches. This allows the aggregate cache resources to adapt to the dynamic demands of different applications. Migrated lines are transferred in compressed format and stay compressed once they arrive at their targets. Based on the analysis in Section 7.4.1, local compression is used as a first defense to cache shortage and migration is used as a last resort before evicting data off chip. We evaluate the marginal performance gain of an application to decide its sensitivity to cache size, and organize compressed lines using our pair-matching scheme.

Figure 7.4.2 gives an overview of the proposed techniques. Each processor has private L1 and L2 caches. For processor P_0, the whole L2 cache is compressed; for processor P_n, half of the L2 cache is compressed (L2C in the figure), which results in a four-level cache hierarchy: L1, uncompressed L2, compressed L2, and remote L2. Note that uncompressed

L2 and compressed L2 share the same resources but have different access latencies. After being accessed, data are placed in L1 cache and are demoted through the levels of the memory hierarchy until dropped or sent to off-chip memory.

7.4.3. Adaptive Compression and Migration

We use marginal performance gain (please refer to Section 6.7.1) to compress and migrate lines only when the running application may benefit from a larger cache size. The procedure of adaptive compression and migration operates as follows. Three marginal performance gain thresholds are used to control adaption: MPG_{T1}, MPG_{T2}, and MPG_{T3}. In Figure 6.7.2, we give a simple example of 2-way associative cache. A cache with higher associativity would be organized into two multi-way hierarchies.

1. When the marginal performance gain, mpg, of the running application is lower than the first threshold MPG_{T1}, neither cache way is compressed. However, a newly-loaded line is always placed in way-1 instead of way-2. Evicted lines from way-1 are placed in way-2 instead of off-chip memory. Evicted lines from way-2 are evicted off-chip. One can view way-2 as another level of cache hierarchy with the same access time as a regular L2 cache.

2. When mpg exceeds the first threshold MPG_{T1}, the compression of way-2 starts. Note that way-2 stores older data than way-1. Whenever a new line is sent to way-2, it is compressed, replacing a stale uncompressed line. The additional space left in the data block will later be used to store another compressed line. At this point, way-2 is split into two compressed ways.

When mpg further exceeds the second threshold MPG_{T2}, way-1 is also compressed, following the same procedure.

3. When mpg decreases below MPG_{T2}, the decompression of way-1 starts. Whenever any compressed line is accessed, it is decompressed, evicting its partner compressed line, if presents. When mpg falls below MPG_{T1}, decompression of way-2 begins, following the same procedure.

4. When a compressed line in way-2 is to be evicted, if the current mpg exceeds MPG_{T3}, depending on the cache coherence protocol, data may be sent to the directory node. The directory node decides whether to drop the line, migrate it to a remote on-chip cache, or send it to off-chip memory.

In practice, MPG_{T1}, MPG_{T2}, and MPG_{T3} can be decided via experiments on typical applications. Fixed empirical values can then be used for all applications. Our experimental results validate this approach[1]. We now discuss several important issues for compression and migration.

• Whenever a line is compressed, the technique first tries to find the best-fit compressed partner that has space available in the data block. If that fails, the LRU line in the compressed ways is evicted to accommodate this line. If necessary, the partner line of the LRU line is also evicted. Note that no processor stalls or additional buffers are necessary to handle compression requests. Compression does not block the issue of later instructions because data awaiting compression

[1]The OS might potentially dynamically adjust threshold values, but fixed, empirically-determined values were sufficient for good results.

are placed in the input buffer of the hardware compressor. Furthermore, since compression is only triggered by L2 cache misses, another compression request will not be generated before the first missed line is sent to the processor, either from off-chip memory or from a remote L2 cache. In both cases the latency is longer than compression plus arbitrating the location of a compressed line.

• Whenever a compressed line is accessed, it is decompressed and sent to the L1 cache or the on-chip network and then forwarded to the requesting processor. The line is also sent back to the uncompressed ways of the same set in the L2 cache. If there is no available space, the LRU line in the uncompressed ways is sent to the compressed ways. Note that decompression is on the performance critical path and therefore its latency must be minimized.

• The LRU compressed lines in local L2 cache may be migrated to remote caches to remain on-chip. Cache migration can improve the performance of the local processor, but may also hurt the performance of the remote processor. Therefore, migration requests must be sent to the right target, i.e., a processor where the running application has smaller marginal performance gain at its current cache size. Our goal is to minimize the total CPI for simultaneous processes:

$$\sum_{i=1}^{N} CPI_i(c_i) = \sum_{i=1}^{N} \frac{CycleCount_i(c_i)}{InstructionCount_i(c_i)} \tag{4}$$

In this equation, $\sum_{i=1}^{N} c_i = C$, where C is the total number of cache lines in all on-chip L2 caches. To minimize the total CPI, the net benefit of incrementing the number of cache lines for one processor and the loss of decrementing the number of cache lines for another processor

1

TABLE 7.2. Parameters in Simulations

Parameter	Value
Processor	In-order, 1 GHz
L1 I/D cache	32 KB, 2-way, 64 B line size, 2 cycles
L2 cache	8-way, 64 B line, 10 cycles private, 30 cycles shared
Off-chip memory	80 cycles, 10 bytes per cycle
On-chip network	Point-to-point, 5 cycles/hop, 100 bytes/cycle

must be positive. Therefore, migration should only be allowed from processors with higher marginal performance gains to processors with lower marginal performance gains.

7.5. Full-System Simulation Results

This section describes full-system simulation results of the proposed cooperative compression and migration techniques. We first describe our simulation environment and workloads, and then present and analyze the results.

7.5.1. Simulation Environment

We used the Simics [**78**] full-system simulator and the Ruby module from the GEMS infrastructure [**79**] to simulate the detailed memory system and interconnect network. We heavily modified the *MSI_MOSI_CMP_directory* cache coherence protocol provided by GEMS to support cache compression and migration. Our simulator is based on the Simics/Aurora SPARC Linux, which models the SPARC V9 architecture in sufficient detail to run a Linux 2.6.13 SMP kernel. We used the in-order processor model provided by Simics mainly to reduce simulation time. For all our simulations, each processor has private L1 instruction and data caches, and

r

TABLE 7.3. Benchmarks

Multiprogrammed SPEC Benchmarks			Multiprogrammed DIS Benchmarks		
Mix 1	art, twolf	T1 and T1	Mix 5	matrix, field	T1 and T2
Mix 2	applu, mesa	T2 and T2	Mix 6	transitive, update	T1 and T1
Mix 3	art, mgrid	T1 and T2	Mix 7	matrix, transitive	T1 and T1
Mix 4	twolf, applu	T1 and T2	Mix 8	field, pointer	T2 and T2

a private L2 instruction/data cache. Inclusion is maintained between L1 and L2 caches. The architectural parameters used in the simulations are listed in Table 7.2.

We used CACTI 5.0 beta [**76**] to estimate L2 cache area. We held line size (64 Byte) and associativity (8-way) constant. All cache area estimates are based on 65 nm process. For CPU area estimation, we assume a fixed-area processor model [**84**], in which each processor core and its associated L1 cache is have the same area as 1 MB of L2 cache, i.e., approximately 20 mm^2.

7.5.2. Workloads

To evaluate the cache compression and migration techniques, we used eight multiprogrammed workloads and four multithreaded workloads. Our multiprogrammed workloads are combinations of ten heterogeneous SPEC CPU2000 benchmarks and Data-Intensive Systems (DIS) stressmarks [**73**]. Our multithreaded workloads include two benchmarks (*ammp* and *art*) from the SPEC OMP v3.2 and two benchmarks (*CG* and *EP*) from the Nasa Parallel Benchmarks [**80**] (NPB) v3.2. Our benchmark configurations are listed in Table 7.3. Below we describe the set up of the multiprogrammed and multithreaded benchmarks, respectively.

For multiprogrammed workloads, we studied via simulation the memory characteristics of the SPEC CPU2000 benchmarks and the DIS stressmarks and divided them into two categories:

performance sensitive to L2 cache size (*T1*) and performance-insensitive to L2 cache size (*T2*). We then used benchmark mixes that represent the following three types of application combinations: (1) mix of two cache-sensitive applications, (2) mix of one cache-sensitive application and one cache-insensitive application, and (3) mix of two cache-insensitive applications. For each mixed workload of two applications, we ran the simulation twice, each time starting at the "early single" SimPoint [**82**] of a different application, and stopping simulation when this application has executed 100 million instructions. We then averaged the results of the two runs to approximate the performance of this benchmark mix.

For multithreaded workloads, we set the number of threads to the number of processors. For the two multithreaded SPEC OMP benchmarks, *ammp* and *art*, we use the medium versions and the reference inputs. The central part of these two benchmarks is a major loop whose body contains code executed by multiple threads in parallel. We first fast forward to the major loop of the benchmarks, then warm up the cache with 100 million instructions, and finally measure performance in the following 100 million instructions. For the two multithreaded NPB benchmarks, *CG* and *EP*, we use the problem class S, warm up the cache with 100 million instructions, and then run to completion and measure the performance.

7.5.3. Performance Evaluation on Multiprogrammed Workloads

For multiprogrammed workloads, we performed two sets of experiments to evaluate the impact of cooperative and adaptive cache compression and migration on the overall system throughput and the performance of each application. First, we fixed the number of cores to two and varied the L2 cache size. We show that our techniques reduce cache requirements and

thereby reduce total chip area. Second, we identify the impact of our techniques on optimal core-cache area ratio given a fixed on-chip area.

For each set of experiments, we present the results of multiprogrammed benchmarks listed in Table 7.3 under four system settings: (1) basic private L2 cache, (2) compressed private L2 cache, (3) private L2 cache with migration under the control of marginal performance gain (using MPG_{T1} to control when migration should start), and (4) private L2 cache with adaptive and cooperative compression and migration. We use a fixed cache line decompression latency of eight cycles as indicated by our hardware synthesis results.

In order to decide the appropriate threshold values MPG_{T1}, MPG_{T2}, and MPG_{T3}, we measured the marginal performance gain values of all evaluated applications at the cache size range of 64 KB to 8 MB. Three threshold values spanning the range of variation in performance for cache-sensitive applications were empirically determined (MPG_{T1} = 0.0000375, MPG_{T2} = 0.0000625, and MPG_{T3} = 0.0001). A single set of thresholds were used: it was not necessary to tune them to particular application.

7.5.3.1. *Evaluation at Fixed Core Count*

In this section, we show the evaluation of the cooperative compression and migration techniques in a two-core CMP with varied private L2 cache sizes. The overall throughput and individual performance of evaluated multiprogrammed benchmarks are shown in Figure 7.2 and Figure 7.3. Table 7.4 summarizes the relative throughput improvements of the evaluated techniques: adaptive migration (mig.), static compression (comp.), and cooperative and adaptive compression and migration (coop.).

TABLE 7.4. Throughput Improvements for Evaluated Techniques as Functions of L2 Cache Size

	mig.	comp.	coop.	mig.	comp.	coop.	mig.	comp.	coop.	mig.	comp.	coop.
L2 size	art, mgrid			applu, mesa			art, twolf			applu, twolf		
64 KB	0%	0.5%	0.7%	0.6%	0.5%	0.6%	6.5%	8.8%	9.9%	2.0%	3.9%	3.9%
128 KB	0%	-0.5%	0%	0.5%	0%	0.5%	4.6%	6.3%	6.8%	1.8%	2.7%	2.8%
256 KB	0%	-0.4%	0%	0%	-1.8%	0%	3.4%	5.5%	5.7%	1.8%	2.8%	2.9%
512 KB	0%	2.0%	8.4%	0%	-2.1%	0%	3.1%	10.7%	12.7%	2.6%	4.7%	4.7%
1 MB	6.9%	11.3%	12.6%	0%	-2.0%	0%	1.1%	12.4%	17.0%	0%	-1.9%	0%
2 MB	0.4%	-1.7%	0.5%	0%	-2.1%	0%	0%	-10.7%	0%	0%	-4.1%	0%
L2 size	matrix, field			matrix, transitive			transitive, update			field, pointer		
64 KB	0.6%	1.0%	0.8%	2.1%	4.8%	2.7%	0.6%	1.1%	0.6%	0%	-4.2%	0%
128 KB	0%	1.0%	1.1%	2.3%	14.6%	23.2%	0.2%	9.8%	15.2%	0%	-4.6%	0%
256 KB	3.9%	3.3%	4.7%	18.8%	34.9%	37.5%	14.8%	16.9%	17.5%	0%	-4.7%	0%
512 KB	0%	-1.6%	0%	0%	-8.3%	0%	0%	-3.2%	1.8%	0%	-4.8%	0%
1 MB	0%	-1.8%	0%	0%	-9.3%	0%	1.7%	3.0%	2.1%	0%	-4.8%	0%
2 MB	0%	-1.7%	0%	0%	-9.2%	0%	0%	-5.8%	0%	0%	-4.9%	0%

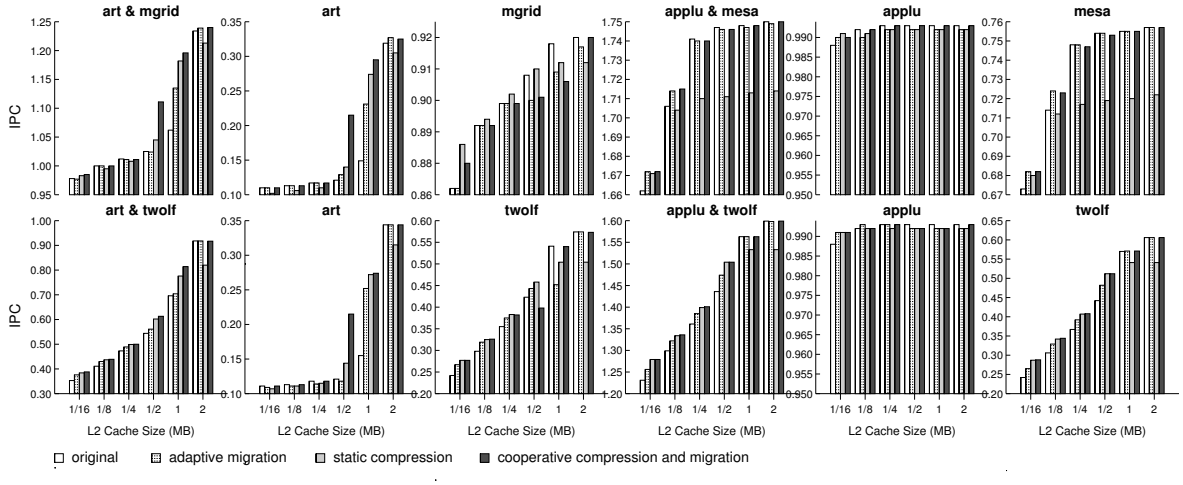

FIGURE 7.2. Performance of multiprogrammed SPEC benchmarks as a function of cache size.

As shown in Figure 7.2 and Figure 7.3, the cooperative cache compression and migration techniques provide the maximum throughput improvement over all system settings whenever improvement is possible by increasing L2 cache size. When such improvement is not possible,

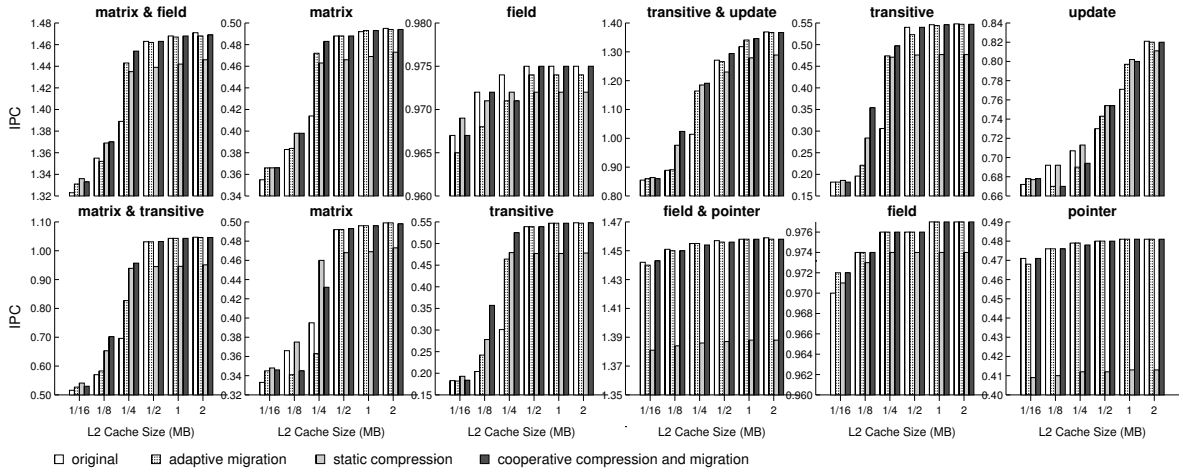

FIGURE 7.3. Performance of multiprogrammed DIS benchmarks as a function of cache size.

the cooperative technique maintains the lowest performance degradation due to its adaptive nature. We find that for the three *T1 + T2* mixes (i.e., *art* and *mgrid*, *applu* and *twolf*, as well as *matrix* and *field*) the proposed techniques result in significant throughput improvements. The maximum improvements for these applications over different L2 cache sizes range from 4.7% to 12.6%. For the three *T1 + T1* mixes (i.e., *art* and *twolf*, *matrix* and *transitive*, as well as *transitive* and *update*) the maximum throughput improvements range from 17% to 37.5%. The other two *T2 + T2* mixes (i.e., *applu* and *mesa*, as well as *field* and pointer) are not affected by our techniques because their performance cannot be improved with increased cache size. We make the following observations on the evaluated techniques.:

1. The cooperative technique combine the advantages of both static compression and adaptive migration: in all cases, it enables the maximum performance improvement in sensitive cache size ranges, and results in minimum performance penalty in insensitive cache size ranges.

Therefore, if compression and/or migration are used, they should be adaptively guided using a metric such as the proposed marginal performance gain.

2. In their sensitive cache size ranges, *T1* applications usually benefit more from static compression than adaptive migration. This is because locally compressed caches have lower access latency than remote caches. However, in cache size ranges for which performance is not strongly dependent on cache size, static compression usually results in a performance penalty, due to the increased cache hit latency. In contrast, adaptive migration seldom imposes any performance penalty because lines are not migrated in this case.

3. The cooperative technique has the greatest potential to reduce on-chip cache requirement. For example, for the *matrix* and *field* mix, with the cooperative techniques, the throughput at the cache size of 256 KB is 99.4% of the throughput at the cache size of 512 KB; the individual performance of *matrix* and *field* are also almost identical. This implies that the actual cache area requirement can be reduced to half, and the total chip area requirement can be reduced by 17%. For the mix of *art* and *mgrid*, to achieve similar performance of a 2 MB private cache (4 MB in total for two cores), only half the cache area is required with the proposed technique, thereby reducing the total chip area by 33%.

7.5.3.2. *Evaluation at Various Core-Cache Area Ratio*

This section evaluates the impact of the proposed techniques on CMPs with different core-cache area ratios given a fixed chip area constraint. The total chip area is approximately the sum of the cache area and the CPU area multiplied by number of CPUs. We assume the L1

1

TABLE 7.5. Evaluated Core-Cache Area Ratios

Core/L2 area ratio	Number of cores	Core area (mm^2)	L2 size (MB)	Cache area (mm^2)	Total area (mm^2)
0.28	2	40	8	143.1	183.1
0.72	4	80	6	109.9	189.9
1.70	6	120	4	70.6	190.6
4.37	8	160	2	36.6	196.6
∞	10	200	0	0	200

TABLE 7.6. Throughput Improvements for Evaluated Techniques for Different Core–Cache Area Usages

	mig.	comp.	coop.	mig.	comp.	coop.	mig.	comp.	coop.	mig.	comp.	coop.
Cores	art, mgrid			applu, mesa			art, twolf			applu, twolf		
2	0%	-5.2%	0%	0%	-2.0%	0%	0%	-13.9%	0%	0%	-4.1%	0%
4	5.6%	5.6%	9.9%	0%	-1.2%	0%	4.9%	4.1%	7.9%	0%	-3.0%	0%
6	0%	7.7%	8.1%	0%	-2.1%	0%	3.5%	18.0%	18.0%	1.4%	1.4%	1.6%
8	0%	-0.5%	0%	0%	-1.1%	0%	3.2%	5.8%	7.6%	1.0%	1.8%	2.5%
Cores	matrix, field			matrix, transitive			transitive, update			field, pointer		
2	0%	-1.6%	0%	0%	-9.8%	0%	0%	-6.5%	0%	0%	-4.9%	0%
4	0%	-1.7%	0%	0%	-9.9%	0.1%	0.7%	-4.3%	1.1%	0%	-4.7%	0.1%
6	0%	-1.5%	0.1%	0%	-4.6%	0%	1.0%	-1.2%	2.8%	0%	-4.8%	0%
8	3.7%	3.2%	4.3%	19.8%	34.0%	36.0%	11.0%	16.8%	17.6%	0%	-4.4%	0%

caches are integrated into the CPU cores, and use the CACTI 5.0 Beta model [**76**] to estimate the area of L2 cache given its size (with a 65 nm technology). Finally, we set the total chip area constraint to 200 mm^2: the approximate area of a 10 MB L2 cache.

To demonstrate the effect of migration, we simulate configurations at two-processor increments to balance the mixes of application types. Each core executes one thread. For example, in a six-core system, for benchmark mix 1, three copies of *art* and three copies of *mgrid* are executed, and each copy is assigned to a single processor core. We evaluate the core-cache

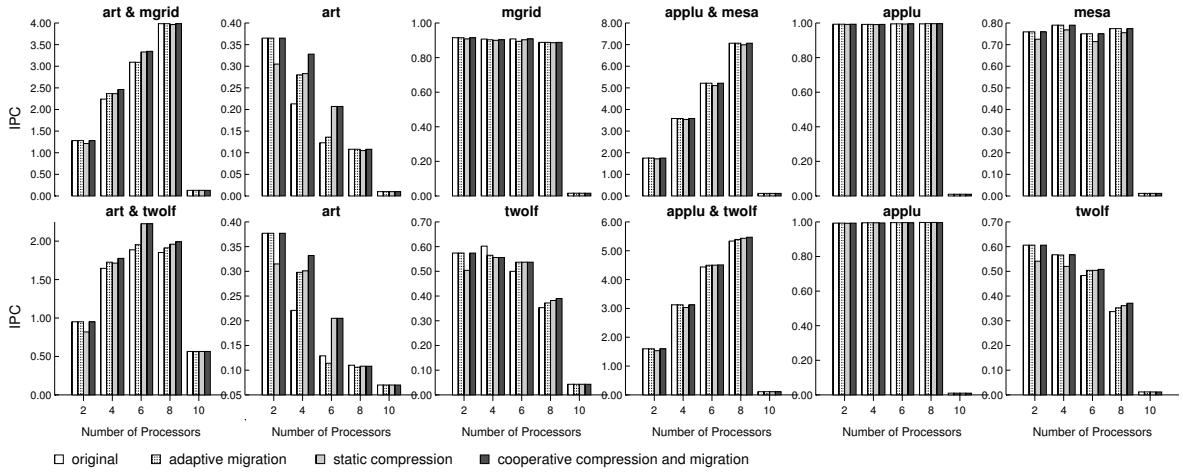

FIGURE 7.4. Performance of multiprogrammed SPEC benchmarks for different cache–core area tradeoffs.

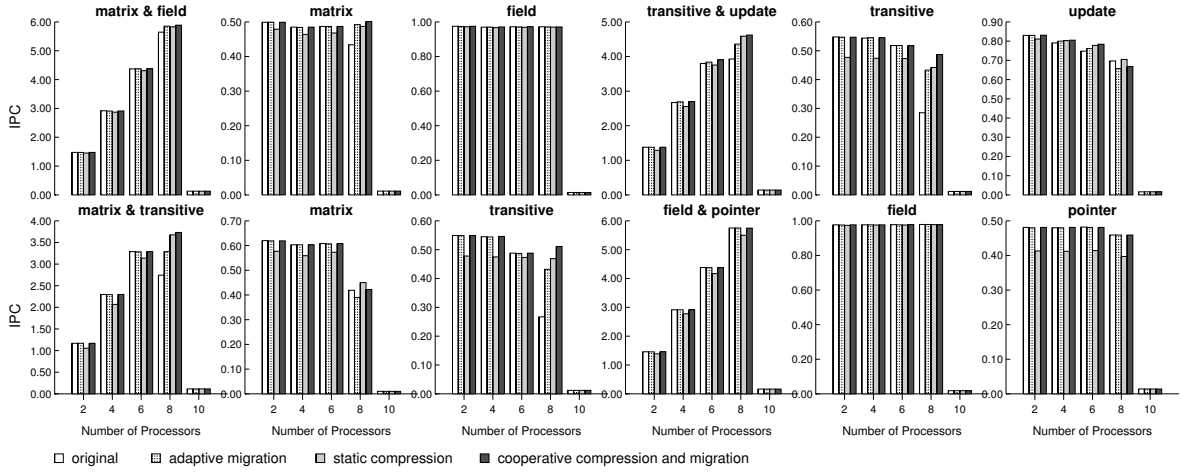

FIGURE 7.5. Performance of multiprogrammed DIS benchmarks for different cache-core area tradeoffs.

area ratios listed in Table 7.5, approximating the area constraint of 200 mm^2 as closely as possible. Figure 7.4 and Figure 7.5 report the IPC of each individual core and the total throughput. Table 7.6 summarizes the relative throughput improvements of the three evaluated techniques. Note that the 10-core configuration has no L2 cache and therefore consistently suffered large

performance penalties. We plot the results of 10-core CMP in the figures to illustrate an extreme case: with no on-chip L2 cache, none of the evaluated techniques can help improve throughput.

The behavior of the evaluated technique for different benchmark mixes is similar to that shown in Section 7.5.3.1. Recall that our goal is to optimize the overall throughput, i.e., total IPC over all cores. We make the following observations on the evaluated techniques.

1. The area-constrained, performance-optimal solution uses less cache per core than is common for existing processors. In current dual-core and quad-core processors, a significant portion of the chip area is devoted to the cache. The Intel Core 2 Duo processor has a die size of 143 mm^2, about 50% of which appears to be used by the 4 MB L2 cache. The AMD Athlon 64 FX-62 processor also has 50% of its die area used as on-chip cache. However, our experimental results indicate that adding processor cores often achieves better throughput than increasing L2 cache size. We feel that the current design trend of CMPs has been influenced by the lack of online techniques to adjust cache use to running applications. For example, for the mix of *art* and *mgrid*, when the core count is increased from four to six, although the throughput is improved by 38%, the performance of *art* reduces by 42.3%. The performance of *mgrid* is not affected by this change. However, with the proposed cooperative and adaptive techniques, the individual performance of *art* at the core count of six is 97% of the original performance without our techniques at the core count of four. This indicates that, for this benchmark mix, with the proposed adaptive techniques, the 6-core design not only outperforms the 4-core design, providing 36% higher throughput, but also guarantees same good performance for individual cores.

2. We found that the cooperative techniques may influence optimal core-cache area ratio for maximum throughput. For the mix of *art* and *twolf*, without the cooperative techniques, the overall performances of 6-core CMP and 8-core CMP are approximately the same. However, with the 18% performance improvement brought by cooperative compression and migration, it is clear that the optimal core count is six instead of eight. On the contrary, for the mix of *matrix* and *transitive*, without the cooperative techniques, the 6-core CMP has the maximum throughput. However, with cooperative cache compression and migration, a 36% improvement makes the throughput of 8-core 13.5% higher than that of the 6-core design, thereby becoming the new optimal core-cache ratio. In summary, the optimal core-cache ratio depends on the application mix for which the CMP is optimized and is influenced by the use of adaptive cache compression and data migration.

7.5.4. Performance Evaluation on Multithreaded Workloads

In this section, we present the evaluation results of the cooperative cache compression and migration technique on four multithreaded workloads. We compare the overall system throughput of our technique with that of a private L2 cache and a shared L2 cache on a four-core CMP. We set the number of OpenMP threads to the number of processors. We evaluated four L2 cache sizes; in each case, the aggregate cache size is the same for each comparison, i.e., 1 MB private cache vs. 4 MB shared cache. The results are presented in Table 7.7.

The OpenMP implementations of the SPEC OMP and NPB benchmarks achieve parallelization by searching for loops with fully independent iterations and then annotating these loops with `OMP PARALLEL DO` directives. Therefore, the threads usually perform identical tasks and

l

TABLE 7.7. Evaluation on Multithreaded Benchmarks as a Function of Aggregate L2 Cache Size

Total L2	Pivate L2	Shared L2	Coop.	+% private	+% shared
ammp					
1MB	1.37	2.98	1.37	0%	-54%
2MB	1.39	3.00	3.61	160%	21%
4MB	4.23	3.00	4.23	0%	41%
8MB	4.25	3.01	4.25	0%	41%
art					
1MB	3.01	3.13	3.01	0%	-4%
2MB	3.08	3.23	3.14	2%	-3%
4MB	3.26	3.34	3.26	0%	-2%
8MB	3.30	3.35	3.31	0%	-1%
CG					
1MB	2.85	2.55	3.96	39%	55%
2MB	4.36	3.40	4.34	0%	28%
4MB	4.46	5.38	5.42	22%	1%
8MB	5.52	5.39	5.53	0%	3%
EP					
1MB	5.41	5.37	5.41	0%	1%
2MB	5.46	5.40	5.89	8%	9%
4MB	6.18	6.31	6.88	11%	9%
8MB	7.06	6.74	7.05	0%	5%

seldom communicate or share data. As a result, the marginal performance gains of the threads are very similar and we observed little migration for these benchmarks. The benefits of our technique mainly comes from adaptive cache compression. Table 7.7 illustrates that our technique outperforms private cache in all cases and outperforms shared cache in most cases. In specific, for benchmark *ammp*, the maximum throughput improvement over private cache is 160%, and the maximum throughput improvement over shared cache is 41%. The only case

where our technique has a lower performance than a shared cache is when the cache size per core is 256 KB. This is because at that cache size, the marginal performance gain of *ammp* is not large enough to trigger compression. (Doubling cache size does not improve throughput, but quadrupling it does.) The only case when shared L2 cache performs the best is for benchmark *art*. The cooperative technique does not result in obvious throughput improvement for *art* because its marginal performance gain is small over all evaluated cache sizes.

Our evaluation of cache-sensitive multithreaded benchmarks indicates that if the threads of the application execute almost identical tasks, the difference between their marginal performance gain is too small to trigger migration. Therefore, only compression may help improve the throughput. However, note that our migration technique is orthogonal to replication techniques. Therefore, other replication techniques may also be deployed if the applications exhibit good data sharing characteristics.

7.5.5. Sensitivity to Decompression Latency

To determine the sensitivity of cooperative cache compression and migration techniques to decompression latency, we performed the following experiments. We used the experimental setup in Section 7.5.3.1 with a fixed core count of two and evaluated the benchmark mixes that benefit from cooperative techniques at their largest performance improvement cache size. We varied the decompression latency from 2 to 12 cycles. Figure 7.5.5 illustrates the improvements to overall throughput under different decompression penalties. Note that it illustrates the importance of decompression speed, not variation in PBPM decompression speed. We found that

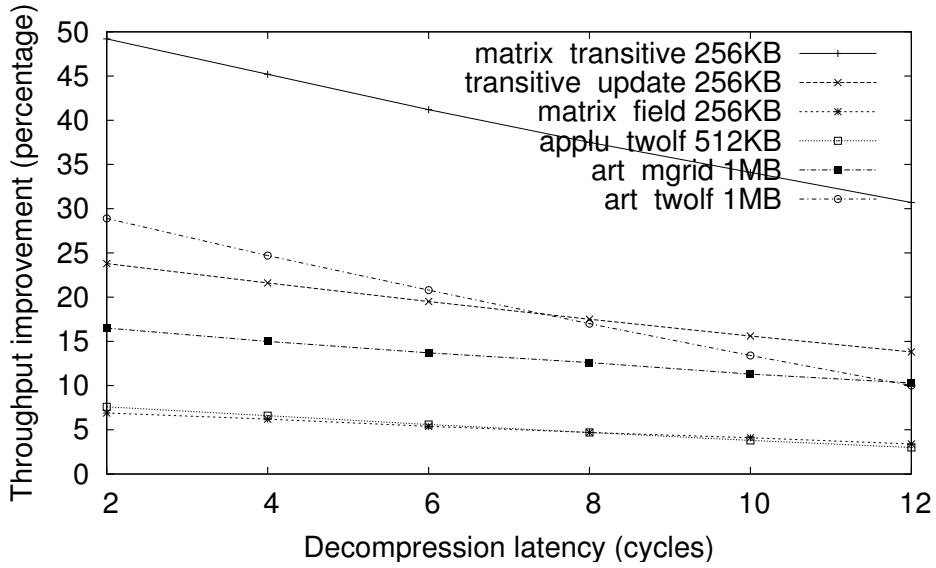

FIGURE 7.6. Sensitivity to decompression latency.

the performance improvements for all benchmark mixes are strongly affected by increased decompression latency. In fact, the percentage improvement is directly linear in decompression latency. The results clearly demonstrate the importance of efficient compression/decompression hardware design.

7.6. Conclusions

The move to CMPs increases the importance of carefully using available cache area. We proposed a cooperative L2 cache compression and data migration technique to permit improvement in CMP throughput without increasing area, or to reduce area without degrading throughput. We evaluated these techniques using full-system simulation running various multiprogrammed and multithreaded workloads. We show that for cache-sensitive applications, the maximum CMP throughput improvement with the proposed technique range from 4.7%–160%

(on average 34.3%), relative to a conventional private L2 cache architecture. No performance penalty is imposed for cache-insensitive applications. The results indicate that proactively distributing cache resources among processors based on their relative performance impacts permits the best overall performance on a wide range of applications. Using a cooperative combination of compression and migration guided by marginal performance gain permitted better performance than either migration or compression, alone. The feasibility, area, and performance overheads of the required control, compression, and decompression hardware were evaluated via detailed design and synthesis. The performance impact was explicitly modelled and the area overhead was found to be negligible compared to processor and cache area. We therefore conclude that cooperative compression and migration appears to be a viable technique for improving area and/or performance for workloads containing cache-limited applications.

Chapter 8

Contributions and Conclusions

We have presented a comprehensive set of new virtual and physical memory hierarchy design techniques that have the potential to dramatically improve the functionality and performance of modern embedded systems, e.g., portable consumer electronics and wireless sensor network nodes.

We have designed and evaluated a software-only memory expansion method with approximately the same impact as a cost-free DRAM process shrink. Our high-performance operating system controlled on-line memory compression technique is capable of increasing available application memory by 2.5 $\times$ without changes to applications or hardware, and with negligible performance and power consumption penalties. To the best of our knowledge, this is the first software-based memory compression technique that has been used in commercial embedded systems. In June 2007, sales of cellphones running CRAMES started in Europe and Japan. The first such cellphone was the NEC FOMA 904i.

We have extended the above software-based memory compression technique to MMU-less systems, which are usually the least expensive, lowest-power, and most memory constrained embedded systems, such as sensor network nodes and portable consumer electronics.

We are also the first to cooperatively design and evaulate a combined cache compression and migration technique for CMPs. We propose an adaptive control policy integrating the two techniques and permitting run-time adaptation to workload. We proposed a new optimization metric, processor marginal performance gain, as the adaptive control metric. We also proposed a new scheme to organize compressed cache lines that is more efficient than commonly-accepted segmentation-based schemes.

We have developed several high-performance compression algorithms that can be used at different levels of memory hierarchy. We presented PBPM, a fast compression algorithm for software-based on-line memory compression. For this application, we were unable to find any existing algorithm with similar speed that matched the compression ratio of PBPM. We presented a Delta compression algorithm, a fast compression algorithm designed for sensor network data. We are also the first to fully design, optimize, and report performance and power consumption of a cache compression hardware when implemented using a design flow appropriate for on-chip integration with a microprocessor.

We initially set out with the goal of developing novel memory hierarchies for embedded systems. In the process of working toward this goal, we have explored, and solved, a number of specific problems within this research area. We attempted to select problems that are

likely to become more important in the next few years and tested our ideas by complete implementation in software, comparison against past work, and evaluation on a wide range of commonly-accepted benchmarks. It is our hope that others can benefit from our work. If you found this work interesting, and would like to discuss it, please contact me.

Lei Yang

l-yang@northwestern.edu

Appendix A

Human and Application Driven Frequency Scaling for Processor Power Efficiency

Support for dynamic voltage and frequency scaling is now common in processors used in battery-powered portable systems. Conventional dynamic CPU frequency scaling techniques generally use high CPU utilization as an indication of the requirement to increase CPU frequency. These techniques ignore an important fact: the ultimate goal of any computer system is to satisfy its users. In reality, different users have different performance requirements for different applications. In this chapter, we demonstrate that for many interactive applications, the perceivable performance is highly-dependent upon the user, and is not linearly related to the CPU utilization level. This creates a great opportunity for reducing power consumption.

We propose HAPPE (Human and Application driven frequency scaling for Processor Power Efficiency), a dynamic CPU frequency scaling technique that adapts frequency to the performance requirement of the current user for the current application of focus. This technique only

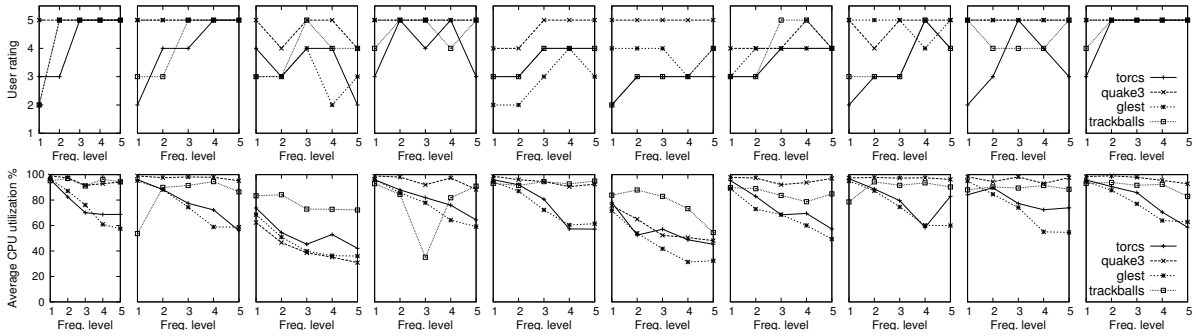

FIGURE A.1. Satisfaction rating and CPU utilization of 10 users at 5 different frequency levels. Level 1 is the lowest frequency and level 5 is the highest frequency.

requires a short training period the first time a user runs an application. After the training period, the user is not required to provide continued feedback but is permitted to provide additional feedback to adjust the control policy. We have evaluated HAPPE on a Linux-based laptop, and have conducted user studies with 24 users and four interactive applications. Comparing to the default Linux CPU frequency controller, HAPPE provides the same satisfaction level to users but reduces measured system-wide power consumption by 25% on average.

A.1. Introduction

Power consumption is a critical design criteria for battery-powered portable systems, such as mobile phones, personal digital assistants, MP3 players, and laptops. Power consumption has also become increasingly-important for line-powered desktop systems and data centers because of its impact on power dissipation and chip temperature, which affects the performance, reliability, and lifetime.

Processors generally dominate the power consumption of high-performance embedded systems [**91**]. Dynamic frequency and voltage scaling (DVFS) is one of the most commonly-used power reduction techniques for processors. DVFS changes the frequency and voltage of processor at runtime to trade off power consumption and performance. Most existing DVFS techniques, such as those used in the Linux and Windows operating systems [**92**, **93**], determine the appropriate processor frequency based on the current CPU utilization. These approaches view CPU utilization as a measure of required performance. Therefore, to guarantee high performance, they only allow a decrease in frequency when CPU utilization is below a certain threshold, e.g., 80%.

These traditional, commonly-used approaches ignore an important fact: the ultimate goal of any computer system is to satisfy its users, not to deliver an absolute level of performance. Throughout this chapter, we define machine performance as the number of instructions executed per second. Although the CPU utilization level is a good indication of machine performance, the actual perceivable performance is highly-dependent upon individual users and applications, and is not linearly related to the CPU utilization level. For some applications, some users are satisfied with the system performance when the processor is at the lowest frequency, while other users may not be satisfied even when operating at the the highest frequency. A user may be insensitive to the performance at different frequency levels for one application, but may be very sensitive to the performance difference for another application. Traditional DVFS policies that consider only CPU utilization or other user-independent hardware performance counters

are usually too pessimistic about user performance requirement, and are generally bound to use a high performance level that can satisfy all users, resulting in wasted power.

There are generally two methods to get user feedback when making DVFS decisions: explicitly obtain user input, e.g, by monitoring continuous key-press events [**94**], or implicitly estimate user satisfaction by measuring metrics such as the rate of change of video output [**95**]. The first method is completely user-driven and does not make use of any machine performance metric, and therefore involves no learning and requires continuous user inputs, which may annoy users and prevent its practical use. The second method, although using the user perceivable performance rather than the absolute machine performance, cannot distinguish among different requirements of different users, and is therefore forced to treat all users the same.

In this chapter, we propose HAPPE (Human and Application driven frequency scaling for Processor Power Efficiency), a new CPU DVFS technique that adapts voltage and frequency to the performance requirement of the current user for the current application of focus. HAPPE associates individual users and applications at different CPU utilization levels with the lowest frequency that satisfies the user. Unlike previous work that requires continuous explicit user feedback, HAPPE only requires a short training period the first time a user runs an application. Then, for each user and application, HAPPE saves the required CPU frequency at different utilization level, and automatically loads this information upon later invocation of the application. After the training period, the user is not required to provide additional feedback, but may occasionally send new inputs to change the control policy, if desired.

We implemented HAPPE as a user-space CPU frequency governor for Linux and compared it to the Linux default *ondemand* [**92**] frequency governor. To find out whether HAPPE can save more power but still satisfy the user, we conducted a study on 24 users with four applications, which represent typical interactive applications on portable embedded systems and laptops, and are all highly CPU intensive. We averaged the measured system power consumptions and user satisfaction ratings for all users and applications. Compared to the ondemand governor, HAPPE reduces the overall system power consumption by 25% without degrading user satisfaction. This reduction is significant, considering that the highest possible system-wide power reduction by scaling frequency from the highest level to the lowest level is 43.5%, when the CPU is above 80% busy (the average CPU utilization of our testing applications).

The rest of this chapter is organized as follows. Section A.2 introduces related work. Section A.3 describes our study on user-perceived performance, and its relationship with CPU utilization level. Section A.4 presents the user and application driven frequency control algorithm used in HAPPE. Section A.5 presents our experimental setup for the user studies and power measurements, and shows the comparison results for HAPPE and the Linux ondemand frequency governor. Finally, Section A.6 concludes the chapter.

A.2. Related Work

Traditional DVFS policies use CPU utilization as the metric to determine when to change frequency [**92**, **93**]. Sasaki et al. [**96**] proposed to use other hardware performance information available to the operating system to make frequency decisions. Vertigo [**97**] adjusts CPU

frequency to different workload characteristics. Choi et al. [**98**] proposed to change CPU frequency based on workload decomposition, which tends to provide power improvements only for memory-bound applications. Wu et al. [**99**] presented a design framework for a run-time DVFS optimizer in a general dynamic compilation system. Xu et al. [**100**] proposed a DVFS scheme that captures the variability of workloads by the probability distribution of the computational requirement of each task in the system. To the best of our knowledge, none of these techniques considered user satisfaction when making frequency decisions.

Lorch and Smith [**101**] found that different types of user interface events, e.g., mouse movements, mouse clicks, and keystrokes, trigger tasks with significantly different CPU requirements, suggesting that DVFS algorithms should use different speeds for them. Mallik et al. [**94**] proposed a DVFS scheme that takes direct user feedback by monitoring keyboard events and adjusts frequency accordingly. However, this technique uses a simple control policy that requires ongoing feedback from the user to maintain adequate performance. The PICSEL [**95**] framework uses measurements of variations in the rate of change of video output to estimate user-perceived performance, and adapts CPU frequency accordingly. However, it ignores the variation among different users, and uses the same threshold for all users. Shye et al. [**102**] proposed to correlate hardware performance counter (HPC) readings with user satisfaction, and used an off-line neural network model to predict user satisfaction based on HPC and set CPU frequency accordingly. Their implementation requires an off-line training stage for each user and each application, which runs the application at different frequency level and asks the user

for verbal satisfaction rating. The user cannot do productive work during the training phase. In addition, if the operating conditions or user preferences change, the training has to be redone.

A.3. User-Perceived Performance

CPU utilization is generally viewed as a predictor for required performance. Therefore, in traditional CPU DVFS policies, it is used as the metric to determine CPU frequency. To guarantee high machine performance, these policies only decrease frequency when CPU utilization is below a certain threshold, e.g., 80%. However, machine performance is not identical to user-perceived performance. Would different users have the same performance requirement for the same application? Would one user have the same performance requirement for different applications? To find out, we conducted a user study with 10 users[1] on a Lenovo Thinkpad T61 laptop, which has a Intel Core 2 Duo processor, 2 GB memory, and runs OpenSuse 10.3 and Linux 2.6.22 kernel. Linux supports five frequencies for this processor: 0.8 GHz, 1.2 GHz, 1.6 GHz, 2.2 GHz, and 2.3 GHz[2], and scales voltage automatically with frequency.

We used four Linux interactive games as our testing applications: Torcs, a 3D car racing game; Quake3, a 3D shooting game; Glest, a 3D real-time strategy game; and Trackballs, a 3D ball maze game. We chose to use these games as our testing applications because (1) they represent typical interactive applications on portable systems, (2) they are usually CPU intensive

[1]The scale of this user study was kept small because each evaluation took about one hour. We conducted a larger scale user study with 20 users to evaluate HAPPE, presented in Section A.5.

[2]The Linux *acpi-cpufreq* driver appears to think that the highest frequency is 2,201 MHz. However, we found the actual frequency is 2.3 GHz using a timing analysis program that operates within cache.

and power consuming, and (3) more users may volunteer to participate in the user study if they are asked to play games.

In the user study, each user played these games at all five frequency levels. The user studies were double-blind and randomized, i.e., the order of frequencies was randomized to eliminate any possible "first-time" execution impact. Each play lasted for at least 2 minutes, and users were permitted to play as long as they desired to evaluate the responsiveness/performance of the system. Then, after each play, users were prompted to enter their "*satisfaction level with the system responsiveness/performance on a scale of 1 to 5, where 5 is the most satisfied and 1 is the least satisfied.*" While users were playing the game, a program ran in the background to sample utilization level for both CPUs on the laptop. Because our test platform has two CPUs, in all user studies the testing application being evaluated is scheduled to run on CPU0 using the Linux *taskset* utility.

Figure A.1 illustrates the satisfaction ratings of 10 users at five frequency levels, where level 1 represents the lowest frequency (0.8 GHz) and level 5 represents the highest (2.3 GHz), and the average utilization of CPU0 (the CPU that is actually running the task), obtained by sampling the CPU utilization every second during the user study for each user. As shown in Figure A.1, all four games are CPU intensive applications. In general, CPU utilization decreases linearly as frequency increases. However, there is a non-linear relationship between user satisfaction and frequency. In fact, different users behave very differently. There appear to be four main categories of users:

- User's satisfaction level continues to increase with frequency.

- User becomes satisfied at a certain frequency level, and is not more satisfied when frequency further increases.
- User is satisfied at all frequency levels, but may have slight variation in ratings.
- User is not satisfied at any frequency level, but may have slight variation in ratings.

Note that user satisfaction–frequency functions also depend on applications. For one application, the user's reaction may belong to one category, but for another application, the same user's reaction may belong to a different category. Our results provide strong evidence that conventional DVFS policies that only use CPU utilization as the control metric, ignoring variation among users and applications, are likely to either annoy users or waste power.

A.4. The HAPPE Approach

The user study results shown in Section A.3 demonstrates that user-perceived performance is highly-dependent on individual user and application, and is not linearly related to CPU frequency. For user-interactive systems, the optimal processor frequency for the current application at the current CPU utilization level is the lowest frequency necessary to satisfy the current user. To approximate the optimal frequency, HAPPE learns user preferences and obtains user feedback by monitoring key-press events on two special keys: the performance key and the power key, which can be mapped to any two keys that are not frequently used. Users may press the performance key when the responsiveness/performance of the system does not meet their requirements. They may also press the power key when they are satisfied with performance and want to save power.

Algorithm 3 HAPPE frequency controller and w_map function

for each sample period P_i **do**
  $f_{next} = f_{cur}$
  if performance key pressed **then**
    $f_{next} = f_{cur} + 1$
    $w\_map(user, app, f_{next}, \overline{util})$
  else if power key pressed **then**
    $f_{next} = f_{cur} - 1$
    $w\_map(user, app, f_{next}, \overline{util})$
  else
    $f_{next} = r\_map(user, app, \overline{util})$
  end if
  if $f_{next} \neq f_{cur}$ **then**
    $set\_freq(f_{next})$
  end if
end for

For each user, the HAPPE frequency controller creates a *user application frequency profile* (UAFP) for every interactive application this user executes[3]. We indicate the highest CPU frequency with f_{max}, the current frequency with f_{cur}, and the current CPU utilization with $util$. The *normalized CPU utilization* is $\overline{util} = util \cdot f/f_{max}$. UAFP divides the normalized CPU utilization into ten discrete levels, e.g., 0 – 10%, 10% – 20%, and maps a user-satisfactory frequency to each level. The frequencies at all normalized utilization levels are initialized to the lowest frequency. Then, every sample period (P) seconds, HAPPE refreshes the frequency for the next sample period (f_{next}), and updates the corresponding UAFP if necessary.

HAPPE determines f_{next} by checking user feedback in the last sample period and looking up the corresponding UAFP entry based on: normalized CPU utilization ($\overline{util}$), current user ($user$), and current application of focus (app). Please see Algorithm 3 for details. If neither

[3]UAFP must distinguish between applications as well as users. Consider the example of user 1 in Figure A.1. If the frequency profile for Quake3 is used for Torcs or Trackballs, the user will be very unsatisfied.

Algorithm 4 HAPPE frequency controller and w_map function

Input: $user$, app, f_{next}, $\overline{util}$
  $map[user][app][\overline{util}] = f_{next}$
  for all $u > \overline{util}$ **do**
    if $map[user][app][u] < f_{next}$ **then**
      $map[user][app][u] = f_{next}$
    end if
  end for
  for all $u < \overline{util}$ **do**
    if $map[user][app][u] > f_{next}$ **then**
      $map[user][app][u] = f_{next}$
    end if
  end for

performance key nor power key was pressed in the last sample period, HAPPE uses the UAFP to determine the frequency that previously satisfied the user at the current normalized utilization level, and adjusts the CPU voltage and frequency appropriately. If HAPPE detects that the performance key was pressed in the last sample period, it increases CPU frequency by one level. Otherwise, if HAPPE detects that the power key was pressed in the last sample period, it decreases CPU frequency by one level. Then, HAPPE updates the corresponding UAFP entry using the method presented in Algorithm 3. Based on the data presented in Section A.3, we assume that for the same user and application, lower CPU utilization levels require equal or lower CPU frequencies. Therefore, HAPPE not only updates the current normalized utilization level, but also checks to make sure that all utilization levels that are higher than the current level have at least the same frequency, and that all utilization levels that are lower than the current level have at most the same frequency.

Unlike previous work that requires continuous explicit user feedback [**94**], HAPPE only requires a short training period the first time a user runs an application. During the training period, the user need only use a few keystrokes to find the satisfactory frequency level. Then, for each user and application, HAPPE learns the user requirement of CPU frequency at different utilization level using the UAFP, and automatically loads the profile for later invocation of the application. After the training period, the user is not required to provide additional input, but is permitted to do so if desired.

We implemented HAPPE as a user-space CPU frequency governor for Linux. In our implementation, the sample period P is set to one second. The governor program uses *Pthreads* to create two threads: T_1 and T_2. The first thread (T_1) polls CPU utilization every second, checks user input signal from T_2, and then scales frequency and updates UAFP when necessary. The second thread (T_2) monitors keyboard events, and sends a signal to T_1 when the performance key or power key is pressed. There are two additional details that proved important when implementing HAPPE.

- Some users may send a burst of key presses when they are unsatisfied with performance, without waiting to observe improvement. This would result in the highest frequency, which may not be necessary to improve satisfaction. To prevent this, we treat all series of key presses with intervals smaller than one second as a single key press.

- For multiprocessor systems, HAPPE manages each CPU individually if their frequencies can be changed independently. If the processors must share the same frequency due to hardware limitation, HAPPE manages the frequency of the group based on the highest utilization.

A.5. Evaluation Results

To evaluate HAPPE, we conducted a study on 24 users running the four Linux interactive games presented in Section A.3. We hope to find answers to the following questions:

- Can HAPPE reduce power consumption more than the Linux ondemand governor by making use of the variation among users and applications? How close are its improvements to the best possible via any DVFS policy?
- Can HAPPE provide similar level of user satisfaction to the Linux ondemand governor?
- How does providing users feedback on the power consumption implications of their decisions influence power consumption and user satisfaction?

A.5.1. Experimental Setup

A.5.1.1. *Power Measurements*

All our experiments were performed on the Lenovo Thinkpad T61 laptop described in Section A.3. Note that in Linux, the frequencies of the two cores on this laptop must scale together. We connected the T61's DC power supply in series with a 100 mΩ Ohmite Lo-Mite 15FR025 molded silicone wire element resistor. Then we measured the voltage across the resistor to obtain the current of the laptop, using a National Instruments 6034E data acquisition board attached to the PCI bus of a host workstation running Linux. This allows us to measure the power consumption of the entire system (including other power consuming components such as memory, graphic card, LCD display, and hard disk). During all experiments, the back-light of the LCD display is set to maximum brightness.

A.5.1.2. *Feedback for Power Key*

When the performance key is pressed, the user receives almost immediate feedback via change in performance. However, when the power key is pressed, the user does not normally receive feedback for a long period of time, e.g., until the battery is exhausted. In real world scenarios, users have incentive to save power when their laptop or other portable device is running on battery, but the benefits come at a longer time scale than that allowed in our user study (2 minutes for each evaluation). Researchers have found that making battery life information visible to end users has a strong impact on power saving for mobile devices [**103**]. In our user studies, to allow feedback on both performance and power consumption, we designed a graphical battery life indicator, which is displayed in the bottom-left corner of the screen. We drive the indicator with simulated battery that is designed to last for two hours when the processor is at the lowest frequency. The indicator displays the remaining operating time based on current battery energy and power consumption, obtained from an on-line power model based on the measured power consumption in Table A.2. *Note that in our evaluation, we tested HAPPE both with and without using the battery life indicator.*

A.5.1.3. *Setup for User Study*

The 24 users are graduate and undergraduate university students; they span a wide range of professional backgrounds, races, ages, and computer and gaming experience levels. The distribution of the users are presented in Figure A.2. Each user evaluation lasted about 40 minutes. Prior to each evaluation, the user was asked to fill out a questionnaire to rate his or

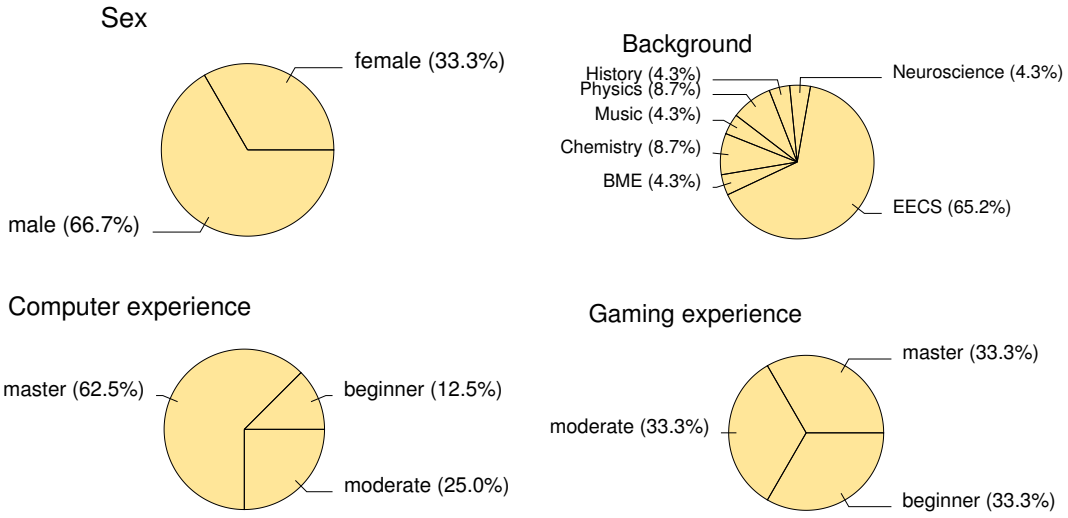

FIGURE A.2. Distribution of users.

her level of experience with computers and computer games using one of the following levels: beginner, moderate, and master. Then the user read a handout with the instructions for the experiment[4]. The users were also shown how the games are played, and were permitted to practice until they are able to play on their own.

In the user study, the users play the four games four times, denoted as *ondemand*, *T-HAPPE*, *O-HAPPE*, and *B-HAPPE*. Each time, the game is played for two minutes and then exits automatically. Afterwards, the users are prompted to enter their "*satisfaction level with the computer performance/responsiveness on a scale of 1 to 5, where 5 is the most satisfied and 1 is the least satisfied*". During the first run, users play the game normally. CPU frequency is controlled by the Linux ondemand frequency controller, with the *sample_rate* set to 80,000 and the *up_threshold* set to 80%. During next three runs, HAPPE is used to control CPU frequency.

[4]We plan to later make the questionnaire and instructions available in a technical report.

Users may press a green-colored key to require higher performance or better responsiveness, or press a yellow-colored key to save power when they are satisfied with the current performance. Note that users are not required to press either key. The differences between the three runs follow.

- *T-HAPPE* is the training phase. During this phase, the frequency at all utilization levels is initialized to the lowest level and adjusted when the two special keys are pressed. Our intention with the evaluation of *T-HAPPE* is to find out whether the training phase annoys the user.
- During *O-HAPPE*, HAPPE loads the UAFP created during *T-HAPPE* and controls frequency accordingly. The user may still press the two keys to adjust performance. However, the frequency of user interaction is likely to be lower, and better approximate HAPPE in normal use.
- During *B-HAPPE*, the user is provided with a battery indicator to get feedback on energy use. It is otherwise equivalent to *O-HAPPE*.

A.5.1.4. *Example of Dynamic Behavior*

To illustrate dynamic system behavior and user interaction with HAPPE, Figure A.3 shows the time series data of a randomly-selected user (user A) playing the Torcs car racing game under the control of HAPPE during the training phase. Figure A.3 shows the utilization of both CPUs, the frequency level sampled every second, and key press events. When user A started playing the game, frequency was set to the lowest level: 0.8 GHz. After 16 seconds, user A pressed the performance key. HAPPE increased the frequency to 1.2 GHz, and recorded the frequency requirement of user A at this utilization level in this user's UAFP. Then, user A pressed

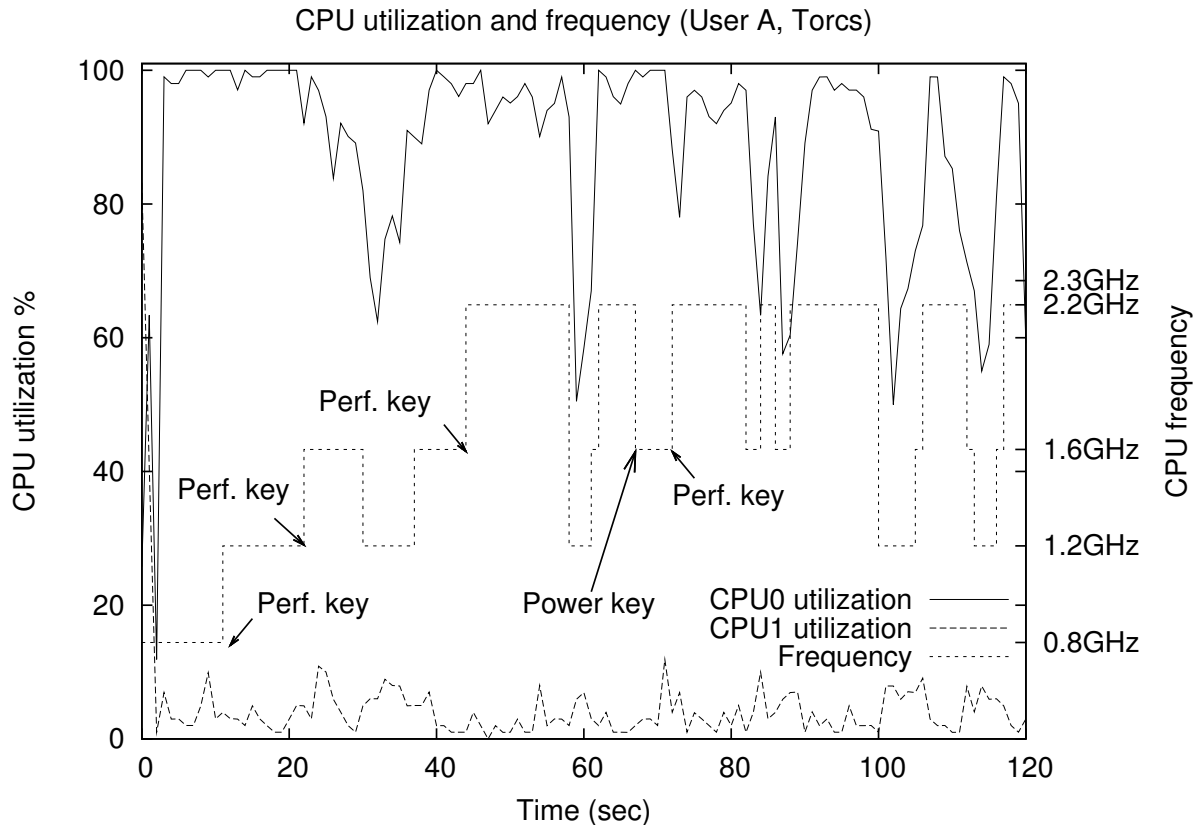

FIGURE A.3. Example of HAPPE training phase.

the performance key again at 22 seconds and 43 seconds, further increasing the frequency to 2.2 GHz. At 67 seconds, user A pressed the power key and soon realized that the resulting degradation in performance was not tolerable. The user therefore pressed the performance key after 7 seconds. Afterwards, frequency stayed at 2.2 GHz for high CPU utilization levels, and 1.2 GHz for low CPU utilization levels.

A.5.2. Comparing HAPPE with Linux Ondemand

Table A.1 presents the aggregated power consumption and satisfaction ratings across all users and applications, comparing HAPPE with the Linux ondemand frequency controller.

TABLE A.1. Average Power Consumption and User Satisfaction Rating

	Torcs					Quake3				
Run	Pwr.	Stdev	Sat.	Stdev	Imp.	Pwr.	Stdev	Sat.	Stdev	Imp.
Ondemand	39.46	1.18	4.54	0.59	0.00%	38.89	1.54	4.88	0.34	0.00%
T-HAPPE	30.38	2.32	4.42	0.58	23.02%	28.03	0.46	4.88	0.34	27.93%
O-HAPPE	32.72	2.92	4.67	0.64	17.09%	28.38	1.27	4.92	0.28	27.03%
B-HAPPE	31.00	2.31	4.33	0.70	21.45%	27.23	0.88	4.92	0.28	29.97%
	Glest					Trackballs				
Run	Pwr.	Stdev	Sat.	Stdev	Imp.	Pwr.	Stdev	Sat.	Stdev	Imp.
Ondemand	37.70	0.77	4.54	0.59	0.00%	42.30	0.51	4.46	0.72	0.00%
T-HAPPE	27.58	1.72	4.50	0.59	26.85%	26.99	2.66	4.38	0.58	36.18%
O-HAPPE	28.45	2.72	4.58	0.58	24.54%	30.07	6.05	4.58	0.50	28.91%
B-HAPPE	28.43	2.52	4.67	0.56	24.60%	29.52	5.69	4.58	0.50	30.21%

For each user, each application, and each technique (*ondemand*, *T-HAPPE*, *O-HAPPE*, and *B-HAPPE*), we obtain the average power consumption during the two minutes the user plays the game. Then, for each application and each technique, the following aggregated results are presented in Table A.1 (from left to right): average power consumption of all users (in Watts), standard deviation of average power consumption (in Watts), average satisfaction rating of all users, standard deviation of average satisfaction rating, and the improvement in power consumption compared to the Linux ondemand controller. We make the following observations based on the results presented in Table A.1.

• Compared to the Linux ondemand governor, across all four applications, the average reduction in power consumption is 28.50% during the HAPPE training phase, 24.39% during the HAPPE operating phase without the battery indicator, and 26.56% during the HAPPE operating phase with the battery indicator. These results are expected, because (1) during the training

phase, the frequency starts low and only increases gradually when user requires higher performance, and (2) with the battery indicator, which provides a feedback on the long-term battery life benefits when the power key is pressed, the user has more incentive to save power.

• Across all four applications, the average user satisfaction is 4.61 with the ondemand governor, 4.55 during the HAPPE training phase, 4.69 during the HAPPE operating phase without the battery indicator, and 4.63 during the HAPPE operating phase with the battery indicator. These results indicate that users are in general slightly less satisfied during the training phase, and more satisfied during the operating phase. In addition, the battery indicator can motivate users to save power by pressing the power key, but very slightly reduces their satisfaction. Nonetheless, compared to the default ondemand governor, during the HAPPE operating phase, users satisfaction actually increases slightly. This slight improvement could be just noise, or might be due to the fact that users are happier when they feel they have control over the computer, i.e., when they press the performance button, they see an instant improvement in the performance/responsiveness[5].

A.5.3. Compare HAPPE with the Best Possible DVFS

We consider the 24.39% – 28.50% power reduction relative to the default ondemand controller achieved by HAPPE significant. To provide the readers a context, in Table A.2, we present the measured results of average system power consumption and CPU temperature of the laptop at all five frequencies and ten CPU utilization levels by running the CPU *playload* [**104**]

[5]We considered the possibility of hardware thermal emergency throttling. However, we ruled out that explanation based on the dynamic frequency, temperature, and power consumption obtained during the user studies.

TABLE A.2. Average Power Consumption of T61 at Different CPU Utilization Level and Frequency

	Power Conumption (Watts)										
	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
0.8 GHz	24.40	24.80	25.22	25.03	25.37	25.81	25.81	26.41	26.60	26.69	26.74
1.2 GHz	25.73	26.02	26.34	25.95	26.92	27.40	27.80	27.92	27.94	28.15	28.55
1.6 GHz	26.14	26.73	27.30	27.93	28.55	29.51	29.86	30.00	30.50	31.19	32.27
2.2 GHz	29.35	30.01	30.81	31.91	32.77	33.79	34.87	36.00	37.25	38.52	40.18
2.3 GHz	30.72	32.01	33.07	34.75	35.55	36.78	39.06	40.52	42.24	43.62	45.04
	Temperature (C)										
	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
0.8 GHz	43.15	44.45	42.70	42.00	42.00	42.00	42.00	42.00	42.00	42.18	42.80
1.2 GHz	42.05	42.00	42.00	42.48	42.83	43.00	43.43	43.83	44.02	44.78	45.00
1.6 GHz	44.33	44.00	44.65	44.82	45.52	46.05	46.83	47.67	48.13	48.90	49.78
2.2 GHz	47.92	48.45	48.82	50.25	51.37	52.67	53.73	55.02	56.58	58.12	59.70
2.3 GHz	54.68	53.00	53.12	54.40	55.42	57.02	58.52	60.28	61.92	64.05	65.35

tool. To approximate the scenarios in the user study, we played the different load on CPU0 for one minute at each frequency level. At each load, we also ran a 3D GPL screen saver that stresses the graphic processing unit (GPU) but not the CPU. This is because the testing applications all require 3D graphic acceleration and are all GPU intensive. We sampled the temperature of CPU0 every second from the Linux ACPI interface, and measured the power consumption of the whole system using the data acquisition card. As shown in the table, when CPU utilization is above 80%, the best possible reduction in system-wide power consumption by decreasing CPU frequency from the highest level to the lowest level is 40.63%.

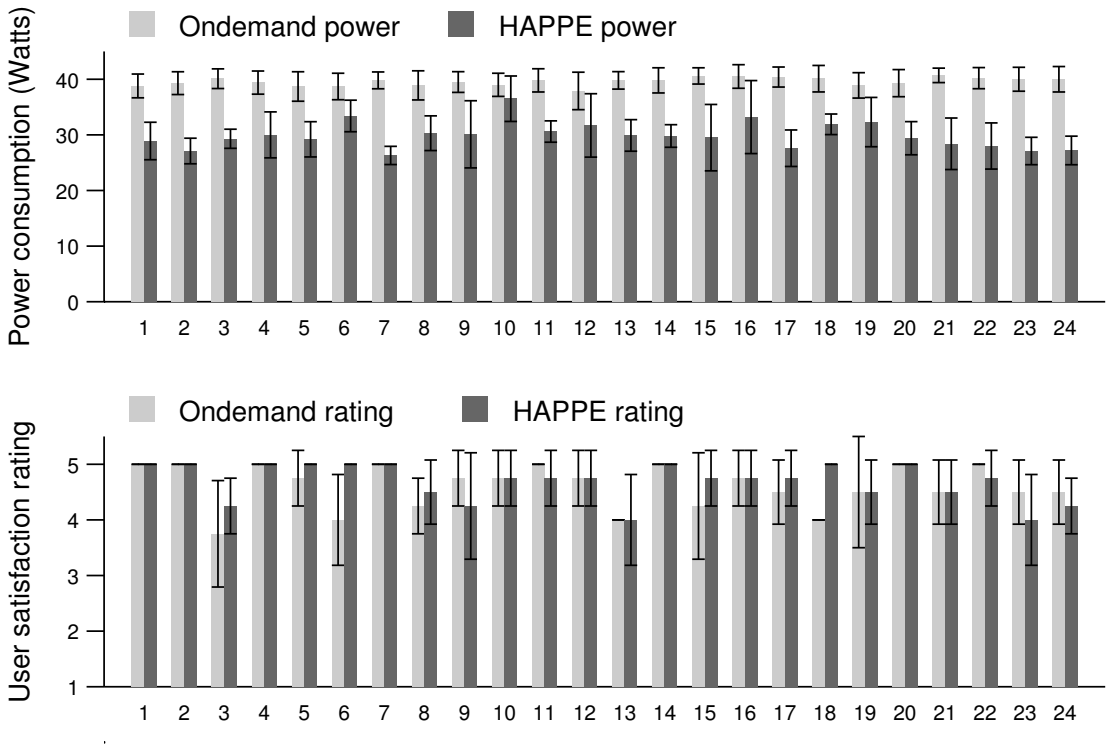

FIGURE A.4. Variation among user power consumption and satisfaction.

A.5.4. Variation Among Users

Figure A.4 shows the variation among users by presenting the power consumptions and satisfaction ratings of all 24 users, comparing *O-HAPPE* to *ondemand*. For each user, we present the average power consumption and average satisfaction ratings for the four testing applications, and show the standard deviations using error bar. As shown in the figures, there is significant variation among users. Some users are very sensitive to performance changes at different frequencies, and require a high frequency to be satisfied, resulting in high power

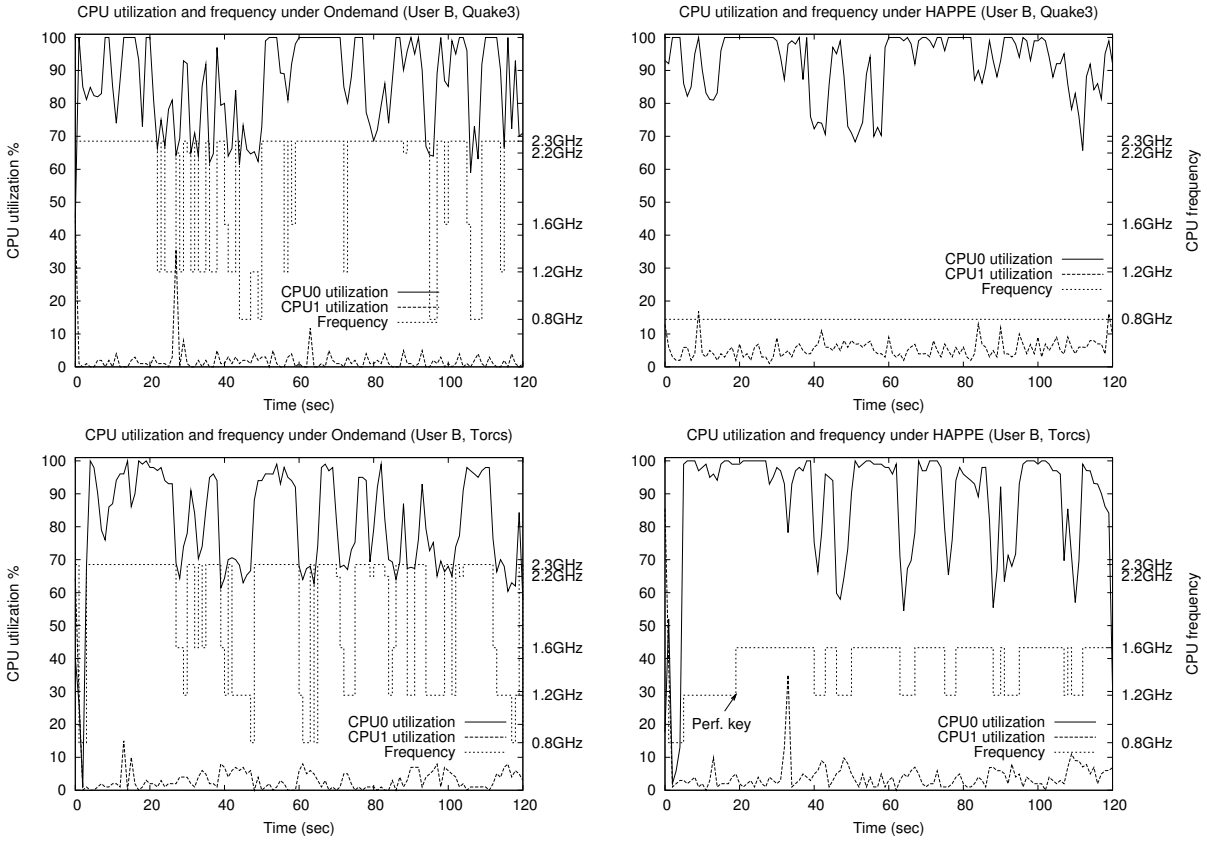

FIGURE A.5. Variation among different applications for same user.

consumption (e.g., user 10); others are less sensitive to the performance difference, and are satisfied at lower frequencies, resulting in lower power consumption (e.g., user 7).

A.5.5. Variation Among Applications for Same User

In this section, we present the variation among the performance requirements for different applications from the same user. Figure A.5 illustrates a randomly-selected user (user B) playing Torcs, the car racing game, and Quake3, the shooting game, under the control of *on-demand* and *O-HAPPE*. Under the control of HAPPE, when playing Torcs, user B pressed the

performance key once at 20 seconds, and became satisfied with the resulting 1.6 GHz frequency. Therefore, HAPPE set the CPU frequency to 1.6 GHz at high utilization levels and 1.2 GHz at low utilization levels. On the contrary, when playing Quake3, the same user was satisfied at the lowest frequency, and did not press the performance key at all. Therefore, HAPPE set the CPU frequency to 0.8 GHz at all times. In contrast, the ondemand governor did not distinguish between the two applications, and decided frequency using only CPU utilization. Because both applications are CPU-intensive, the frequency stayed at the highest level most of the time, resulting in wasted power without increasing user satisfaction.

A.5.6. Variation Among Groups

Based on the user distribution presented in Figure A.2, we did analysis on user groups and calculated the average reduction in power consumption over all testing applications and all runs under control of HAPPE. We found that on average, the power reduction is (1) 25.40% for EECS majors and 28.26% for non-EECS majors, and (2) 23.16% for master level game players, 25.91% for moderate level game players, and 30.29% for beginner level game players. These results indicate that people with more experiences with computers and computer games tend to be more sensitive to the performance differences at different frequency levels, thereby requiring higher frequency to be satisfied. On the contrary, people with less experience are likely to be less sensitive to the performance differences at lower frequency levels, resulting in higher reduction in power consumption. Note that HAPPE automatically adapts to each type of user.

TABLE A.3. Average Number of Key Presses Per Minute

	Training		Operating			Training		Operating	
App.	Perf.	Power	Perf.	Power	App.	Perf.	Power	Perf.	Power
Torcs	1.13	0.40	0.46	0.27	Trackballs	0.69	0.14	0.14	0.16
Glest	0.36	0.23	0.19	0.34	Quake3	0.13	0.38	0.11	0.38

A.5.7. Analysis on User Feedback

To create the UAFP, HAPPE requires user inputs by means of pressing the performance key and the power key during the training phase. After the training phase, user input is not required but is still permitted. Therefore, if the user's performance requirements at different CPU utilization levels for this application never change, the user never needs to send HAPPE inputs again. Table A.3 presents the number of key presses during the training phase and the operating phase, averaged over all users. As shown in the table, during the training phase, there are usually more performance key presses and fewer power key presses than during the operating phase. On average, less than two performance key presses per minute are necessary for users to adapt to a desired frequency, during the first few minutes of the training phase.

A.6. Conclusions

For CPU-intensive interactive applications, traditional DVFS policies that use CPU utilization as the only metric to decide frequency usually result in unnecessarily-high frequency and therefore wasted power. In this chapter, we have presented HAPPE, a dynamic CPU DVFS controller that adapts CPU voltage and frequency to the performance requirements of individual users and applications. HAPPE uses a learning algorithm that creates a profile for each

user and application. For each CPU utilization level, HAPPE learns the frequency necessary to satisfy the user. The learning algorithm trains the profile by accepting user key-press inputs during the first few minutes the first time the user runs the application. After the training phase, HAPPE does not require continued user input. We evaluated HAPPE with a study on 24 users and four testing applications. Compared to the Linux default ondemand frequency governor, HAPPE reduces system-wide power consumption by 25% on average, without degrading user satisfaction.

References

[1] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, CA, third edition, 2003.

[2] Nihar R. Mahapatra and Balakrishna Venkatrao. The processor–memory bottleneck: problems and solutions. *ACM Crossroads*, 5, 1999.

[3] International Technology Roadmap for Semiconductors, 2006. http://public.itrs.net.

[4] Masashi Yokotsuka. Memory motivates cell-phone growth. *Wireless Systems Design*, April 2004.

[5] HP online shopping. http://www.shopping.hp.com.

[6] Metrowerks Embedix. http://www.metrowerks.com/MW/Develop/Embedded.

[7] B. Tremaine, P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, M.E. Wazlowski, and P. M. Bland. IBM memory expansion technology. *IBM J. of Research and Development*, 45(2):271–285, March 2001.

[8] Luca Benini, Davide Bruni, Alberto Macii, and Enrico Macii. Hardware-assisted data compression for energy minimization in systems with embedded processors. In *Proc. Design, Automation & Test in Europe Conf.*, March 2002.

[9] M. Kjelso, M. Gooch, and S. Jones. Design and performance of a main memory hardware data compressor. In *Proc. Euromicro Conf.*, pages 423–430, September 1996.

[10] Haris Lekatsas, Jörg Henkel, and Wayne Wolf. Code compression for low power embedded system design. In *Proc. Design Automation Conf.*, pages 294–299, June 2000.

[11] X. H. Xu, C. T. Clarke, and S. R. Jones. High performance code compression architecture for the embedded ARM/Thumb processor. In *Proc. Conf. Computing Frontiers*, pages 451–456, April 2004.

[12] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression*. Prentice Hall, New Jersey, 1990.

[13] Chandrama Shaw, Debashis Chatterji, Pradipta Maji Subhayan Sen, B. N. Roy, and P. Pal Chauduri. A pipeline architecture for encompression (encryption + compression) technology. In *Proc. International Conf. on VLSI Design*, January 2003.

[14] *An introduction to Thumb*, March 1995.

[15] Fred Douglis. The compression cache: Using on-line compression to extend physical memory. In *Proc. USENIX Conf.*, pages 519–529, January 1993.

[16] Mark Russinovich and Bryce Cogswell. RAM compression analysis, February 1996. http://ftp.uni-mannheim.de/info/OReilly/windows/win95.update/model.html.

[17] Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis. The case for compressed caching in virtual memory systems. In *Proc. USENIX Conf.*, pages 101–116, April 1999.

[18] Compressed caching in Linux virtual memory. http://linuxcompressed.sourceforge.net.

[19] M. Kjelso, M. Gooch, and S. Jones. Performance evaluation of computer architectures with main memory data compression. In *J. Systems Architecture*, volume 45, pages 571–590, 1999.

[20] Luigi Rizzo. A very fast algorithm for RAM compression. *Operating Systems Review*, 31(2):36–45, April 1997.

[21] T. Cortes, Y. Becerra, and R. Cervera. Swap compression: Resurrecting old ideas. *Software-Practice and Experience J.*, (30):567–587, June 2000.

[22] Sumit Roy, Raj Kumar, and Milos Prvulovic. Improving system performance with compressed memory. In *Proc. Parallel & Distributed Processing Symp.*, April 2001.

[23] Irina Chihaia Tuduce and Thomas Gross. Adaptive main memory compression. In *Proc. USENIX Conf.*, pages 237–250, April 2005.

[24] Cramfs: Cram a filesystem onto a small ROM. http://sourceforge.net/projects/cramfs.

[25] Cloop: Compressed loopback device. http://www.knoppix.net/docs/index.php/cloop.

[26] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly & Associates, Inc., second edition, December 2002.

[27] CBD compressed block device. http://lwn.net/Articles/168725/.

[28] RAM doubler. http://www.lowtek.com/maxram/rd.html.

[29] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, B. Mathiske, and M. Wolczko. Heap compression for memory-constrained Java environments. In *Proc. OOPSLA Conf.*, pages 282–301, October 2003.

[30] David Woodhouse. JFFS: The journalling flash file system. In *Ottawa Linux Symp.* Red-Hat Inc., 2001.

[31] LZO real-time data compression library. http://www.oberhumer.com/opensource/lzo.

[32] M. K. McKusick and M. J. Karels. Design of a general-purpose memory allocator for the 4.3BSD UNIX kernel. In *Proc. USENIX Conf.*, pages 295–303, June 1988.

[33] J. L. Peterson and T. A. Norman. Buddy systems. *Communications of the ACM*, 20(6):421–431, June 1977.

[34] T. P. Lee and R. E. Barkley. A watermark-based lazy buddy system for kernel memory allocation. In *Proc. USENIX Conf.*, pages 1–13, June 1989.

[35] R. E. Barkley and T. P. Lee. A lazy buddy system bounded by two coalescing delays per class. In *Proc. ACM Symp. Operating Systems Principles*, pages 167–176, December 1989.

[36] *Intel XScale Microarchitecture for the PXA250 and PXA210 Applications Processors User's Manual*, February 2002. http://www.intel.com/.

[37] Chunho Lee, Miodrag Potkonjak, and William H. Mangione Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. http://cares.icsl.ucla.edu/MediaBench.

[38] Trolltech Qtopia Overview. http://www.trolltech.com/products/qtopia.

[39] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[40] Lan Bai, Lei Yang, and Robert P. Dick. Automated compile-time and run-time techniques to increase usable memory in MMU-less embedded systems. In *Proc. Int. Conf. Compilers, Architecture & Synthesis for Embedded Systems*, pages 125–135, October 2006.

[41] Chris Karlof and David Wagner. Secure routing in wireless sensor networks: Attacks and countermeasures. *Elsevier's AdHoc Networks J.*, 1(2–3):293–315, September 2003.

[42] Prasanth Ganesan, Ramnath Venugopalan, Pushkin Peddabachagari, Alexander Dean, Frank Mueller, and Mihail Sichitiu. Analyzing and modeling encryption overhead for sensor network nodes. In *Proc. Int. Conf. on Wireless Sensor Networks and Applications*, pages 151–159, September 2003.

[43] Johannes Gehrke and Samuel Madden. Query processing in sensor networks. *Pervasive Computing*, 3(1):46–55, January 2004.

[44] D. Li, K. Wong, Y. Hu, and A. Sayeed. Detection, classification, and tracking of targets. *Signal Processing Magazine*, 19(2):17–29, March 2002.

[45] David Gay, Phillip Levis, and David Culler. Software design patterns for TinyOS. In *Proc. Languages, Compilers, and Tools for Embedded Systems*, pages 40–49, June 2005.

[46] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, and R. Han. MANTIS: System support for MultimodAl NeTworks of In-situ Sensors. In *Proc. Int. Wkshp. Wireless Sensor Networks and Applications*, pages 50–59, September 2003.

[47] Siddharth Choudhuri and Tony Givargis. Software virtual memory management for MMU-less embedded systems. Technical report, Center for Embedded Computer Systems, University of California, Irvine, November 2005.

[48] G. J. Pottie and W. J. Kaiser. Wireless integrated network sensors. *Commun. ACM*, 43(5):51–58, May 2000.

[49] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TAG: a Tiny AGgregation service for ad-hoc sensor networks. In *Proc. Symp. Operating Systems Design and Implementation*, pages 131–146, December 2002.

[50] Carlos Guestrin, Peter Bodi, Romain Thibau, Mark Paski, and Samuel Madde. Distributed regression: an efficient framework for modeling sensor network data. In *Proc. Int. Symp. Information Processing in Sensor Networks*, pages 1–10, April 2004.

[51] Joseph M. Hellerstein and Wei Wang. Optimization of in-network data reduction. In *Proc. of Int. Wkshp. on Data Management for Sensor Networks*, pages 40–47, August 2004.

[52] Suman Nath, Phillip B. Gibbons, Srinivasan Seshan, and Zachary R. Anderson. Synopsis diffusion for robust aggregation in sensor networks. In *Int. Conf. Embedded Networked Sensor Systems*, pages 250–262, November 2004.

[53] Cristiano Pereira, Sumit Gupta, Koushik Niyogi, Iosif Lazaridis, Sharad Mehrotra, and Rajesh Gupta. Energy efficient communication for reliability and quality aware sensor networks. Technical report, University of California at Irvine, April 2003.

[54] S. Sandeep Pradhan, Julius Kusuma, and Kannan Ramchandran. Distributed compression in a dense microsensor network. *IEEE Signal Processing Magazine*, 19(2):51–60, March 2002.

[55] Joseph Polastre, Robert Szewczyk, Alan Mainwaring, David Culler, and John Anderson. Analysis of wireless sensor networks for habitat monitoring. *Wireless sensor networks*, pages 399–423, 2004.

[56] Alaa R. Alameldeen and David A. Wood. Adaptive cache compression for high-performance processors. In *Proc. Int. Symp. Computer Architecture*, June 2004.

[57] Erik G. Hallnor and Steven K. Reinhardt. A compressed memory hierarchy using an indirect index cache. In *Proc. 3rd workshop on Memory performance issues*, 2004.

[58] Alaa R. Alameldeen and David A. Wood. Interactions between compression and prefetching in chip multiprocessors. In *Proc. Int. Symp. High-Performance Computer Architecture*, February 2007.

[59] Jang-Soo Lee, Won-Kee Hong, , and Shin-Dug Kim. Design and evaluation of a selective compressed memory system. In *Proc. Int. Conf. Computer Design*, October 1999.

[60] Nam Sung Kim, Todd Austin, and Trevor Mudge. Low-energy data cache using sign compression and cache line bisection. In *Proc. Wkshp. on Memory Performance Issues*, May 2002.

[61] Keun Soo Yim, Jihong Kim, and Kern Koh. Performance analysis of on-chip cache and main memory compression systems for high-end parallel computers. In *Proc. Int. Conf. Parallel and Distributed Processing Techniques and Applications*, pages 469–475, June 2004.

[62] Nihar R. Mahapatra, Jiangjiang Liu, Krishnan Sundaresan, Srinivas Dangeti, and Balakrishna V. Venkatrao. A limit study on the potential of compression for improving memory system performance, power consumption, and cost. *J. Instruction-Level Parallelism*, July 2005.

[63] Magnus Ekman and Per Stenstrom. A robust main-memory compression scheme. *SIGARCH Comput. Archit. News*, pages 74–85, May 2005.

[64] Prateek Pujara and Aneesh Aggarwal. Restrictive compression techniques to increase level 1 cache capacity. In *Proc. Int. Conf. Computer Design*, October 2005.

[65] J. L. Núñez and S. Jones. Gbit/s lossless data compression hardware. *IEEE Trans. VLSI Systems*, 11(3):499–510, June 2003.

[66] Alaa Alameldeen and David A. Wood. Frequent pattern compression: A significance-based compression scheme for l2 caches. Technical report, Dept. of Computer Sciences, University of Wisconsin-Madison, April 2004.

[67] Ivan Sutherland, Robert F. Sproull, and David Harris. *Logical Effort: Designing Fast CMOS Circuits*. Morgan Kaufmann, first edition, 1999.

[68] Jan M. Rabaey. *Digital Integrated Circuits*. Prentice-Hall, NJ, 1998.

[69] Xi Chen, Lei Yang, Haris Lekatsas, Robert P. Dick, and Li Shang. Design and implementation of a high-performance microprocessor cache compression algorithm. In *Proc. Data Compression Conf.*, March 2008.

[70] Alistair Moffat. Implementing the PPM data compression scheme. In *IEEE Trans. on Communications*, November 1990.

[71] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[72] Lei Yang, Haris Lekatsas, and Robert P. Dick. High-Performance Operating System Controlled Memory Compression. In *Proc. Design Automation Conf.*, pages 701–704, July 2006.

[73] Haitao Du. Analysis of memory behavior of DIS stressmark suite and optimization. Technical report, University of California, Irvine, December 2000. http://www.ics.uci.edu/~amrm/hdu/DIS_Stressmark/DIS_stressmark.html.

[74] G. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *J. of Supercomputing*, 28(1):7–26, 2004.

[75] Hassan Ghasemzadeh, Sepideh Sepideh Mazrouee, and Mohammad Reza Kakoee. Modified pseudo LRU replacement algorithm. In *Proc. Int. Symp. Engineering of Computer Based Systems*, March 2006.

[76] CACTI: An integrated cache access time, cycle time, area, leakage, and dynamic power model. http://quid.hpl.hp.com:9082/cacti/.

[77] P. Franaszek, J. Robinson, and J. Thomas. Parallel compression with cooperative dictionary construction. In *Proc. Data Compression Conf.*, April 1996.

[78] Simics. http://www.virtutech.com.

[79] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. GEMS: Multifacet's general execution-driven multiprocessor simulator. In *Proc. Int. Symp. Computer Architecture*, June 2005.

[80] H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical report, 1999. NAS Technical Report: NAS-99-011.

[81] Man-Lap Li, Ruchira Sasanka, Sarita V. Adve, Yen-Kuang Chen, and Eric Debes. The alpbench benchmark suite for complex multimedia applications. In *Proc. Int. Symp. Workload Characterization*, October 2005.

[82] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *Proc. Int. Conf. Parallel Architectures and Compilation Techniques*, September 2003.

[83] Jaehyuk Huh, Doug Burger, and Stephen W. Keckler. Exploring the design space of future CMPs. In *Proc. Int. Conf. Parallel Architectures and Compilation Techniques*, September 2001.

[84] Yingmin Li, Benjamin Leez, David Brooks, Zhigang Huyy, and Kevin Skadron. CMP design space exploration subject to physical constraints. In *Proc. Int. Symp. High-Performance Computer Architecture*, pages 17–28, February 2006.

[85] Ozcan Ozturk, Mahmut Kandemir, and Mary Jane Irwin. Increasing on-chip memory space utilization for embedded chip multiprocessors through data compression. In *Proc. Int. Conf. Hardware/Software Codesign and System Synthesis*, pages 87–92, September 2005.

[86] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proc. Int. Symp. Microarchitecture*, December 2006.

[87] Jichuan Chang and Gurindar S. Sohi. Cooperative caching for chip multiprocessors. In *Proc. Int. Symp. Computer Architecture*, pages 264–276, June 2006.

[88] Michael Zhang and Krste Asanovic. Victim replication: Maximizing capacity while hiding wire delay in tiled CMPs. In *Proc. Int. Symp. Computer Architecture*, June 2005.

[89] Michael Zhang and Krste Asanovic. Victim migration: Dynamically adapting between private and shared CMP caches. Technical report, Massachusetts Institute of Technology, October 2006.

[90] Bradford M. Beckmann, Michael R. Marty, and David A. Wood. ASR: Adaptive selective replication for cmp caches. In *Proc. Int. Symp. Microarchitecture*, December 2006.

[91] Aqeel Mahesri and Vibhore Vardhan. Power consumption breakdown on a modern laptop. In *Proc. Workshop on Power Aware Computing Systems, Int. Symp. Microarchitecture*, December 2004.

[92] Venkatesh Pallipadi and Alexey Starikovskiy. The ondemand governor: Past, present, and future. In *Proc. the Linux Symposium*, volume 2, July 2006.

[93] Windows native processor performance control. Technical report, Microsoft Corporation, 2002.

[94] Arindam Mallik, Bin Lin, Peter Dinda, Gokhan Memik, and Robert P. Dick. Process and User Driven Dynamic Voltage and Frequency Scaling. Technical report, Northwestern University, August 2006.

[95] Arindam Mallik, Jack Cosgrove, Robert P. Dick, Gokhan Memik, and Peter Dinda. PICSEL: Measuring user-perceived performance to control dynamic frequency scaling. In *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, March 2008.

[96] Hiroshi Sasaki, Yoshimichi Ikeda, Masaaki Kondo, and Hiroshi Nakamura. An intra-task DVFS technique based on statistical analysis of hardware events. In *Proc. Int. Conf. Computing frontiers*, May 2007.

[97] Krisztián Flautner and Trevor Mudge. Vertigo: automatic performance-setting for linux. In *Proc. Int. Symp. Operating Systems Design and Implementation*, December 2002.

[98] Kihwan Choi, Ramakrishna Soma, and Massoud Pedram. Dynamic voltage and frequency scaling based on workload decomposition. In *Proc. Int. Symp. Low Power Electronics & Design*, August 2004.

[99] Qiang Wu, Margaret Martonosi, Douglas W. Clark, V. J. Reddi, Dan Connors, Youfeng Wu, Jin Lee, and David Brooks. A dynamic compilation framework for controlling microprocessor energy and performance. In *Proc. Int. Symp. Microarchitecture*, November 2005.

[100] Ruibin Xu, Daniel Mossé, and Rami Melhem. Minimizing expected energy consumption in real-time systems through dynamic voltage scaling. *ACM Trans. on Computer Systems*, December 2007.

[101] Jacob R. Lorch and Alan J Smith. Using user interface event information in dynamic voltage scaling. Technical report, University of California at Berkeley, August 2002.

[102] Alex Shye, Berkin Ozisikyilmaz, Arindam Mallik, Gokhan Memik, Peter A. Dinda, Robert P. Dick, and Alok N. Choudhary. Learning and leveraging the relationship between architecture-level measurements and individual user satisfaction. In *Proc. Int. Symp. Computer Architecture*, June 2008.

[103] Ahmad Rahmati, Angela Qian, and Lin Zhong. Understanding human-battery interaction on mobile phones. In *Proc. Int. Conf. Human Computer Interaction with Mobile Devices and Services*, September 2007.

[104] P. Dinda and D. O'Hallaron. Realistic CPU workloads through host load trace playback. In *Proc. Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, May 2000.