High-Performance Operating System Controlled Online Memory Compression

LEI YANG and ROBERT P. DICK
Northwestern University
HARIS LEKATSAS and SRIMAT CHAKRADHAR
NEC Laboratories America

Online memory compression is a technology that increases the amount of memory available to applications by dynamically compressing and decompressing their working datasets on demand. It has proven extremely useful in embedded systems with tight physical RAM constraints. The technology can be used to increase functionality, reduce size, and reduce cost, without modifying applications or hardware. This article presents a new software-based online memory compression algorithm for embedded systems. In comparison with the best algorithms used in online memory compression, our new algorithm has a competitive compression ratio but is twice as fast. In addition, we describe several practical problems encountered in developing an online memory compression infrastructure and present solutions. We present a method of adaptively managing the uncompressed and compressed memory regions during application execution. This memory management scheme adapts to the predicted memory requirements of applications. It permits efficient compression for a wide range of applications. We have evaluated our techniques on a portable embedded device and have found that the memory available to applications can be increased by 2.5× with negligible performance and power consumption penalties, and with no changes to hardware or applications. Our techniques allow existing applications to execute with less physical memory. They also allow applications with larger working datasets to execute on unchanged embedded system hardware, thereby increasing functionality.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management—*Virtual memory*; C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Real-time and embedded systems*

General Terms: Design, Management, Performance

Additional Key Words and Phrases: Embedded system, memory, compression

L. Yang is currently affiliated with Google Inc. amd R. P. Dick is currently affiliated with the University of Michigan.
This work was supported in part by NEC Labs America and in part by the National Science Foundation under awards CNS-0721978 and CNS-0347941.
Authors' addresses: L. Yang; email: leiyang@google.com; R. P. Dick; email: dickrp@eecs.umich.edu; H. Lekatsas and S. Chakradhar, NEC Laboratories America, Princeton, NJ 08540; email: {lekatsas, chak}@nec-labs.com.

DOI 10.1145/1721695.1721696 http://doi.acm.org/10.1145/1721695.1721696

ACM Reference Format:
Yang, L., Dick, R. P., Lekatsas, H., and Chakradhar, S. 2010. High-Performance operating system controlled online memory compression. ACM Trans. Embedd. Comput. Syst. 9, 4, Article 30 (March 2010), 28 pages. DOI = 10.1145/1721695.1721696 http://doi.acm.org/10.1145/1721695.1721696

1. INTRODUCTION

The design of modern embedded systems is a challenging task due to several conflicting goals: size, cost, and power consumption must be minimized while performance and functionality must be maximized. Functionality depends on the flexibility and features of applications; increasing flexibility and features often results in increased memory requirements. Software running on some embedded devices, such as cellular phones and Personal Digital Assistants (PDAs), is becoming increasingly complicated, requiring faster processors and more memory to execute. Meanwhile, at any particular time, RAM price increases superlinearly with size. For manufacturers of cellular phones and other cost-constrained embedded systems, a $5 difference in RAM or ROM can have a large impact on volume profits. Moreover, adding physical RAM to embedded systems generally increases their cost, size, and power consumption. The current trend is to pack more functionality in less space, while meeting performance and power consumption constraints. In this article, we describe how more functionality can be achieved using the same hardware by making better use of physical memory via memory compression.

1.1 Memory Compression Motivation and Introduction

The desire to minimize embedded system memory requirements has led to the design of various memory compression techniques. These techniques may help embedded systems maintain the same software functionality while reducing cost, or increase software functionality while keeping cost constant. Memory compression is a complex problem that differs from file compression. Memory compression techniques can be divided into two main categories: hardware-based and software-based.

Numerous hardware-based memory compression techniques have been proposed for reducing the amount of ROM or RAM used in embedded systems. Some of these techniques compress application programs that are stored in ROM. In such an architecture, a hardware decompression unit sits between processor and ROM. Application code is compressed offline and stored in ROM, while decompression is performed online during application execution. Other techniques insert a hardware compression–decompression unit between cache and RAM; data are stored in compressed format in RAM and remain uncompressed in cache. These two approaches share the same prerequisite: special-purpose compression-decompression hardware must be designed and integrated into the target system.

Software-based compression techniques have been proposed to improve the performance of general-purpose computing systems with hard disks. In such techniques, a software-based compressed cache is inserted into the virtual

memory hierarchy. This cache uses part of physical RAM to store data in compressed format. When system memory is low, before sending selected pages to hard disks, the system first attempts to compress these pages and store them in the software cache. Since the overhead of compression and decompression is several orders of magnitude smaller than that of disk access, the overall performance of the system can be improved significantly for benchmarks that have large memory requirements.

Unlike hardware-based approaches, software-based techniques do not require the design of dedicated hardware. For software-based techniques, design, test, and deployment of a new application-specific compression algorithm is relatively easy because it requires no hardware changes. Therefore, software-based memory compression generally simplifies the design process in comparison with hardware-based techniques, reducing time-to-market and design costs. Consequently, they can be more easily applied to existing embedded systems and processors.

Despite the advantages of software-based memory compression techniques, few have been used in commercial embedded systems. The primary reason is that the performance and power consumption penalties of software-based online compression are generally considered too high [Benini et al. 2004]. There are other practical issues that must be thoroughly addressed before software-based memory compression can be deployed in real embedded systems. For example, in hardware-assisted RAM compression architectures, data in RAM are compressed at all times: they start compressed and remain compressed. This simplifies hardware design, but is not necessarily optimal. Compression generally imposes performance and energy consumption penalties. Therefore, it should not be used when applications do not require additional memory. In software-based approaches, data in RAM can start uncompressed and be compressed only when necessary. This introduces the new problem of managing migration between compressed and uncompressed portions of memory. More specifically, it is necessary to design methods of selection and scheduling of pages to compress and decompress and dynamic allocation and management of compressed memory.

1.2 Background and Overview

In previous publications [Yang et al. 2005, 2006], we described our software-based memory compression infrastructure: CRAMES. CRAMES was originally motivated by a specific engineering problem faced by our industrial collaborators at NEC during the design of a new cellphone. 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 system function as if RAM had been added without actually changing the hardware. The second approach was chosen, resulting in the CRAMES infrastructure. 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.

Via online memory compression, CRAMES is capable of increasing the 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 (OS'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.

We performed extensive experiments using typical noninteractive and user interactive applications on a Sharp Zaurus PDA with its original RAM size. 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 applications that were able to run without compression.

To demonstrate that CRAMES can be used to reduce the resource requirements of applications with negligible performance and power consumption penalties, we artificially reduced the system RAM of a Zaurus 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, that is, 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. We feared that this performance penalty might prevent CRAMES from being used in some embedded systems.

In this article, we describe two techniques to further improve the performance of online software-based memory compression. More specifically, this article makes the following contributions.

(1) We present PBPM, a fast compression algorithm that efficiently compresses data pages in RAM. PBPM exploits patterns that frequently occur within words in memory and takes advantage of the similarities among words by keeping a small dictionary. When compressing in-RAM data, PBPM is twice as fast as the best compression algorithms of the Lempel-Ziv family [Ziv and Lempel 1977] and offers a competitive compression ratio. For online software memory compression, we were unable to find any existing algorithm with similar speed that matched the compression ratio of PBPM. We also describe how PBPM supports some embedded processors that do not support high-performance unaligned memory access.
(2) We present an 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 preallocation method is able to further increase available memory to applications by up to 13%, comparing to the same system without preallocation.

(3) We demonstrate that it is possible to increase available application memory by 2.5× without changes to applications or hardware, and with negligible performance and power consumption penalties. To the best of our knowledge, this is the first work explaining a software-only memory expansion method with approximately the same impact as a cost-free DRAM process shrink.

1.3 Article Organization

The rest of this article is organized as follows. Section 2 summarizes related work. Section 3 gives a brief overview of the online OS-controlled compression infrastructure that motivates and provides a realistic testing environment for the techniques proposed in this article. Section 4 describes the proposed software-based memory compression algorithm. Section 5 presents our method of adaptively managing the uncompressed and compressed regions of memory. Section 6 describes the experimental setup, workloads, and experimental results in detail. Finally, Section 7 concludes the article.

2. RELATED WORK AND CONTRIBUTIONS

There are a number of techniques that introduce compression into the memory hierarchy. Hardware-based code compression and main memory compression have been proposed to reduce ROM and RAM size in embedded systems. Software-based RAM compression techniques have been proposed to improve overall system performance by reducing the number of disk accesses for general-purpose computing systems. Next, we categorize these related techniques and summarize the primary contributions of this article.

2.1 Hardware-Based Memory Compression

Code compression techniques [Lekatsas et al. 2000; Xu et al. 2004; Shaw et al. 2003] store instructions in compressed format in ROM and decompress them during execution. Some techniques require the compressed code to be entirely decompressed before execution. Other techniques allow the compressed code to be decompressed instruction by instruction during execution. Compression is usually done offline and can be slow, while decompression is done during execution, usually by special hardware, and must be very fast. It is worth noting that some work proposes software methods to reduce code size with runtime decompression. For example, Lefurgy et al. [2000] presented a method of decompressing programs using software that relied on using a software-managed instruction cache under control of the decompressor.

Main memory compression techniques insert a hardware compression and decompression unit between cache and RAM. Data are stored uncompressed in cache, and are compressed on-the-fly when transferred to memory. This technique is usually deployed to improve the performance of general-purpose systems, where memory bandwidth constitutes a serious bottleneck to program execution speed. It also has the potential to reduce embedded system RAM requirements and power consumption. However, it requires changes to the

hardware and thus cannot be easily incorporated into existing processors and embedded systems.

IBM MXT [Tremaine et al. 2001] is a hardware-based main memory compression technique. 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[1] between 16.7% and 50%, as measured in real-world system applications. This compression ratio is consistent with that achieved by CRAMES.

Benini and Bruni [2002] proposed a similar architecture for memory energy reduction in embedded systems. In their system, uncompressed cache lines are compressed before they are written back to main memory, and decompressed when data are transferred back to cache. They presented two compression algorithms and their hardware implementations. Simulation results indicate that the average energy savings of the cache–memory subsystem range from 4.2% to 35.2%, depending on the selected compression algorithm.

Moore [2003] proposed a hardware-assisted compression infrastructure primarily for the purpose of minimizing communication bandwidth requirements. He is the first to suggest that this technique may be integrated with the virtual memory system in embedded systems, but did not present a method of doing this.

Kjelso et al. [1996] showed simulation results indicating that hardware-based memory compression can substantially improve performance by eliminating paging due to insufficient memory. They also described the design and hardware implementation of their compression algorithm, called X-match, for in-memory data.

2.2 Software-Based Memory Compression

Most previous work on software-based memory compression falls into two main categories: compressed caching and swap compression. Both have the main goal of improving system performance and both target general-purpose systems with hard disks.

Compressed caching [Douglis 1993; Wilson et al. 1999; Russinovich and Cogswell 1996; Compressed Caching] introduces a software-based cache to the virtual memory system that uses part of the memory to store data in compressed format. The objective is to improve system performance by decreasing the number of page faults that must be serviced by hard disks, which have much longer access times than RAM. Early work by Douglis [1993] proposed a software-based compressed cache, which uses part of the memory to store data in LZRW1 compressed format. Russinovich and Cogswell [1996] presented a thorough analysis of the compression algorithms used in compressed caching.

A study on compressed caching by Kjelso et al. [1999] presented a performance model for compressed caching. It demonstrated that system performance

[1]Compression ratio gives a measure of the compression achieved by a compression algorithm. It is defined as compressed data size divided by original data size. Therefore, a low compression ratio indicates better performance than a high one.

may be improved by up to a factor of two when using software-based memory compression instead of paging, while hardware assisted memory compression can improve performance by up to an order of magnitude. They addressed the problem of memory management for variable-size compressed pages. Their experiments used the LZRW1 compression algorithm.

Wilson et al. [1999] 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.

Swap compression [Rizzo 1997; Tuduce and Gross 2005; Cortes et al. 2000; Roy et al. 2001] compresses swapped pages and stores them in a memory region that acts as a cache between memory and disk. Cortes et al. [2000] proposed 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 objective.

Roy and Prvulovic [2001] proposed a memory compression mechanism for Linux with the goal of improving performance. Again, their approach targets systems with hard disks. They reported speedups from 5% to 250% depending on the application. They did not consider the use of this technique in systems without disks, that is, most embedded systems.

Both compressed caching and swap compression require a backing store, for example, a hard disk. When the compressed area is filled up, its LRU pages must be sent to the backing store so that free memory in the compressed area can be kept above a configurable threshold. Unlike hardware-based main memory compression techniques, both compressed caching and swap compression are unable to compress code in memory. Neither of these two techniques was designed or evaluated for use in embedded systems. There is some evidence to suggest they would not be appropriate for this application. For example, they generally use compression algorithms that impose high overheads, and require disk as a backing store.

2.3 Compression Algorithms for In-RAM Data

Compression techniques, both lossy and lossless, are widely used in all fields of information processing. Compression ratio and compression–decompression speed vary greatly depending on the algorithm and specific data. Online memory compression requires lossless compression algorithms. Unfortunately, many existing lossless algorithms are not suitable for this application. There are three main reasons for this. First, the algorithm must perform well operating on data blocks 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. Second, to minimize performance penalty, compression and decompression must be extremely fast. Many existing compression algorithm implementations, for example, gzip and bzip2, provide very good compression ratios but are too slow for use in online memory compression. Finally, the algorithm must require little memory to operate on embedded systems with tight memory constraints. bzip2 requires over 7MB memory to compress and over 3MB to decompress.

These memory requirements are too high for many embedded systems, which frequently have 32MB of RAM or less.

LZO [Oberhumer] is a very fast general-purpose lossless compression algorithm that works well on memory data. It is an asymmetric compression algorithm: decompression is faster than compression. LZO compresses a block of data into dictionary matches and nonmatching literals. LZO adjusts to long matches and long literal runs so that it produces good results on highly redundant data and deals acceptably with incompressible data. When dealing with incompressible data, LZO expands the input block by a maximum of 16 bytes per 1,024 bytes of input data. LZO requires requires 64KB of memory for compression[2] and no additional memory for decompression.

Rizzo [1997] attempted to analyze the frequent patterns present in memory data. He concluded that zero-valued data are the most frequent and that the next frequent symbols are difficult to determine and very content-dependent. He therefore proposed a software-based algorithm that compresses in-RAM data by exploiting only the high frequency of zero-valued data. In order to improve speed, the algorithm uses W-bit Huffman encoding, where W is the size of a machine-word. Rizzo's algorithm is faster than our proposed algorithm presented in this article, but has a much worse compression ratio (on average 64% on our test data in comparison to 44% for PBPM) and therefore was not considered a viable candidate for CRAMES. In Section 4.1, 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. [1996] designed the X-Match hardware compression algorithm that maintains a dictionary of previously seen data 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. An incoming word can have all bytes match a dictionary entry (full match), or it can have at least any two of the four bytes match exactly with a dictionary entry (partial match), with the bytes that do not match being transmitted literally. Data elements which do not produce a match are transmitted literally prefixed by a single bit.

IBM's MXT technology [Tremaine et al. 2001] is a main memory compression technique based on a parallelized hardware implementation of a derivative of the Lempel-Ziv (LZ77) sequential algorithm [Ziv and Lempel 1977]. With this implementation, the uncompressed data block is partitioned into a number of 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. [1999] 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 match a complete 32-bit dictionary entry, match only the upper 22 bits, or fail to match any dictionary entry. As a special case,

[2]There is also a compression level that requires 8KB memory but has a poorer compression ratio.

the word is first checked to see if it is all zeroes, that is, matches a full-word zero. Like Rizzo's algorithm, the WKdm algorithm exploits the most 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 exploit other frequent patterns in the word itself and therefore has a compression ratio ranging from 47% to 50% for in-RAM data. In most cases, the compression ratio of WKdm is inferior to that of LZO.

3. OVERVIEW OF CRAMES

This section briefly introduces the design and implementation of CRAMES to provide readers a context for the techniques presented in this article. To increase available memory to embedded systems, CRAMES selectively compresses data pages when the working datasets of running processes exceed physical RAM capacity. When data in a compressed page are later required by a process, CRAMES locates that page, decompresses it, and copies it back to the main memory working area, allowing the process to continue executing. In summary, applications that would normally be unable to run to completion on an embedded system with insufficient RAM can execute correctly with the help of CRAMES. In addition to dataset compression, CRAMES also supports compression of any type of in-RAM filesystem (commonly referred to as RAM disk).

3.1 Design Challenges

The goal of CRAMES is to significantly increase available memory with minimal performance and energy penalties, and without requiring additional hardware. Next, we summarize the primary challenges for CRAMES and our methods of overcoming them.

(1) *Selection and scheduling of pages for compression and decompression*. This is essential to guarantee correct operation. 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 might occur when a memory request cannot be satisfied. In summary, the Least Recently Used (LRU) pages should be compressed when the memory usage of active processes exceeds the main memory working area.

In order to minimize performance and energy consumption impact, CRAMES uses the OS virtual memory swapping mechanism to decide which data pages to compress and when to perform compression and decompression. Figure 1 illustrates the data flow path for CRAMES. Note that virtual memory contains both uncompressed areas (white) and compressed areas (gray). CRAMES requires a Memory Management Unit (MMU). However, no special-purpose hardware is required.

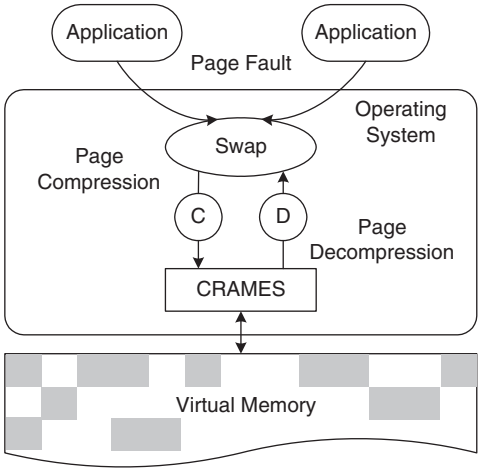

Fig. 1. Overview of CRAMES.

(2) *Design of an efficient compression algorithm*. Compressing and decompressing pages and moving them between uncompressed memory and compressed memory consumes time and energy. Profiling results indicate that compression and decompression are responsible for the main performance penalty in CRAMES. Using a high-quality compression algorithm is thus crucial to ensure the practical use of CRAMES. The compression algorithm must have excellent performance and low energy consumption. It must also provide a good compression ratio and small working space memory requirements, to substantially increase the amount of usable memory.
There is a trade-off between compression speed and compression ratio. In general, slower compression algorithms have better compression ratios; faster compression algorithms have poorer compression ratios. We compared implementations of existing data compression algorithms that span a range of compression ratios and execution times: bzip2, zlib (at level 1, 6, and 9), LZRW1-A, LZO, and RLE (Run Length Encoding). 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 appears to be most appropriate for online memory compression because of its good all-around performance. However, as described in Section 1.2, when the system is under tight memory constraints, LZO compression may degrade application performance significantly. In Section 4, we describe our efforts to design a compression algorithm more appropriate for use in online memory compression and demonstrate its effectiveness in reducing the performance penalty of CRAMES.

(3) *Dynamic management of the compressed memory*. Since sizes of compressed pages vary widely, efficiently distributing and locating data in compressed memory is challenging. Compression transforms the simple 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 *Kernel Memory Allocation* (KMA) problem [Vahalia 1996]. The compressed memory manager must be fast and high quality, that is, it must minimize waste resulting from fragmentation.

In addition to handling memory allocation, the compressed memory manager must also dynamically change the size of the compressed area based on the amount of memory required by currently executing applications. 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.

The CRAMES memory manager builds upon methods used in KMA. In order to identify the most appropriate memory allocation method, we implemented and evaluated several classical memory allocators. We decided to use the *Resource Map* allocator [Vahalia 1996], which achieved the best trade-off of allocation speed and memory usage.

(4) *Minimizing the performance and memory overhead*. In practice, a software-based memory compression technique must have high performance to permit use in real embedded systems. CRAMES must minimize the performance overhead introduced by compression, decompression, and memory management. Furthermore, compressed data pages may vary in size and cut memory into fragments with different sizes. CRAMES must also minimize the memory overhead of compression, fragmentation, and indexing compressed pages to ensure an improvement in physical memory capacity.

3.2 Implementation and Evaluation

CRAMES has been implemented as a loadable Linux kernel module for maximum portability and modularity. It can easily be ported to other modern OSs that support virtual memory. The module is a special block device (i.e., a random access device that stores and retrieves data in blocks) using physical memory. It may serve as both a swap device and a storage area for filesystems.

The memory space in a CRAMES device consists of several virtually contiguous memory chunks that are maintained in a linked list. Each chunk is divided into blocks with potentially different sizes. 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. 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. When a CRAMES device receives a read request for a block, it locates the block using an index 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.

Freeing compressed data belonging to terminated processes posed a challenge in the design of CRAMES. A conventional swap device need not be notified when processes terminate and data are freed. However, it was useful to add a mechanism to notify CRAMES when data should be freed because they belong to a terminated process. Otherwise, such data may remain in memory in compressed format indefinitely, potentially degrading memory utilization and performance. We solved this problem by making small modifications to the Linux kernel. In particular, whenever the kernel attempts to free a page located in the swap device, the modified *swap_entry_free* function in *swapfile.c* calls a CRAMES function to eliminate that page from compressed memory, thus ensuring the swap device only holds data that belong to nondefunct processes.

CRAMES was first evaluated on a battery-powered PDA using widely used noninteractive benchmarks as well as interactive applications with Graphical User Interfaces (GUIs). Experimental results show that CRAMES is capable of dramatically increasing memory capacity with small performance and power consumption costs. A next-generation cellphone prototype board was later used to aid development. Using this platform, we were able to run real cellphone applications and performed extensive testing to fine-tune CRAMES. During the final stages of development, the memory management system of CRAMES was greatly improved to alleviate potential fragmentation problems. A product development team at NEC Japan took over development and did rigorous testing under real network conditions. In June 2007, sales of cellphones running CRAMES started in Europe and Japan. The first such cellphone was the NEC FOMA 904i.

4. PATTERN-BASED PARTIAL MATCH COMPRESSION

We first considered using the LZO [Oberhumer] algorithm to compress data in memory. LZO is significantly faster than many other general-purpose compression algorithms, such as LZW series algorithms, zlib, and bzip2. However, it is not designed for memory compression and therefore does not fully exploit the regularities of in-RAM data. In addition, LZO requires 64KB of working memory for compression, a significant overhead on many memory-constrained embedded systems. LZO provides good compression ratio and performance in many applications. However, better results are possible for online memory compression. We now analyze the regularities of in-RAM data and describe a new algorithm, named PBPM, which is extremely fast and well-suited for memory compression.

The PBPM algorithm is designed based on the observation that frequently encountered data patterns can be encoded with fewer bits to save space. Scanning through the input data a word (32bits) 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) very frequent patterns are encoded using special bit sequences that are much shorter than the original data, (2) patterns that do not fall into this category and are found 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 for later use and the original word contents are sent to output.

We adopted a systematic approach when designing PBPM. Based on our observations on the properties of in-RAM data, we first identified the key components of the algorithm: exploiting data patterns, dictionary maintenance, and pattern encoding. We then statistically analyzed the data to identify the most frequent patterns, and developed a prototype in Python. Afterwards, we adjusted parameters such as the size and layout of dictionary and coding scheme to optimize the compression ratio. Finally we implemented the algorithm in C and used a number of software acceleration techniques to optimize its performance. Different applications were used during the development of the algorithm and its final evaluation in CRAMES. We believe this systematic approach is useful when designing compression algorithms customized to datasets with particular statistical properties.

4.1 In-RAM Data Patterns

Unlike general-purpose algorithms designed for text data, a special-purpose algorithm designed for memory compression must fully exploit the regularities of in-RAM data. 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 64MB swap data file from a workstation running SuSE Linux 9.0. Various applications were executed to exhaust physical memory and trigger swapping. Figure 2 shows the relative frequencies of patterns we evaluated. Next, we specify conventions for 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 it with four symbols, each representing a byte. A "z" represents a zero byte, an "x" represents an arbitrary byte, and an "m" represents a byte that matches a dictionary entry. Following this convention, "zzzz" indicates an all-zero word; "mmmx" indicates a partial match with a 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 table, containing random indices within the range of the dictionary. The decision to base hashing on the third byte was made to achieve decent hashing quality with low computational overhead. Based on this hash function, we only

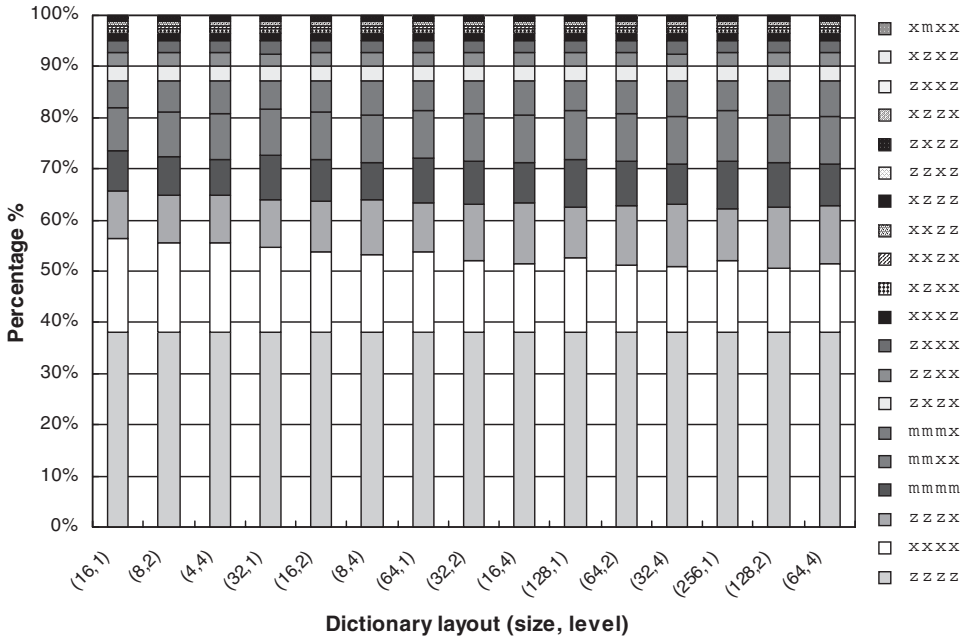

Fig. 2. Frequent pattern histogram.

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 and layouts, for example, 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 provides 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 2 illustrates, zero words, "zzzz", are the most frequent compressible pattern, followed by one byte positive sign-extended words "zzzx". Other zero-related patterns are relatively less frequent. 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

Table I. 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%

set-associative dictionary works no better than a two-way set-associative dictionary in this application.

4.2 The PBPM Compression Algorithm

The PBPM compression and decompression algorithms are presented in Algorithm 1. Based on our analysis, we selected the most frequent patterns and the most effective dictionary layouts for PBPM. The patterns and coding schemes are summarized in Table I, which also reports the actual frequencies of the listed patterns in our swap data file. In Algorithm 1 and column "Output" of Table I, "B" represents a byte and "b" represents a bit.

PBPM maintains a small two-way set-associative dictionary (*DICT*[ ] in Algorithm 1) of 16 recently seen words. An incoming word can fully match a dictionary entry, or match only the highest three 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 [Tremaine et al. 2001; Wilson et al. 1999], we have experimentally determined that considering only byte-aligned partial matches is sufficient to exploit the partial similarities among in-RAM data while permitting 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 criteria the word meets:

(1) Is it a "zzzz"?
(2) If not, is it a "zzzx"?
(3) If not, is it a "zxzx"?

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. 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 I, and adds entries to the dictionary upon a partial match or dictionary miss. Therefore, the dictionary can be reconstructed during decompression and does not need to be stored together with the compressed data.

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
```

4.3 Software Acceleration

During the compression process, depending on different patterns, an input word may be converted to one of the following outputs (please refer to Table I).

—Two-bit code (zzzz),
—Two-bit code followed by four-byte data (xxxx),
—Two-bit code followed by four-bit index (mmmm),

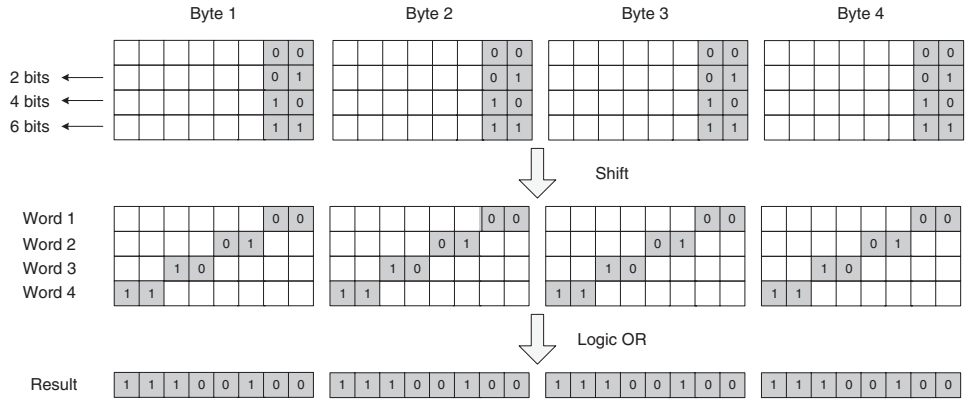

Fig. 3. Fast word shift operation.

—Four-bit code followed by one-byte data (zzzx),
—Four-bit code followed by four-bit index and two-byte data (mmxx),
—Four-bit code followed by four-bit index and one-byte data (mmmx), or
—Four-bit code followed by two-byte data (zxzx).

The output of all compressed words can be stored in one flat array, or *tape*. For example, for pattern "mmmm", a two-bit code is first sent to the output tape, followed by a four-bit index. However, this may not be the most efficient implementation due to alignment problems introduced by nonuniform symbol lengths. In our implementation, we used separate tapes for code, index, and data. Because the code length may be either two-bit or four-bit, we have two tapes for codes, each of which consists of two-bit sequences. As a consequence, the first two bits of a four-bit code are sent to T_{code1} and the second two bits are sent to T_{code2}; meanwhile, the four-bit codes are always sent to T_{index} and data are always sent to T_{data}. To reduce the number of memory copies, we place the data tape at a fixed position in the output buffer. Other tapes are appended via copying to the data tape at the end of compression process to minimize runtime. The data tape is usually the longest among all tapes. Therefore, this arrangement of tapes results in the shortest runtime. Please note that since data length may vary for different patterns, the data tape T_{data} consists of one-byte, two-byte, and four-byte sequences, which may cause the word alignment problem. We will discuss this problem in more detail in Section 4.4.

To allow fast bit operations, we used the following acceleration technique for the code tapes and the index tape. Code tapes always consist of two-bit sequences. Therefore, during compression we store each two-bit code in the lowest two bits of a byte. After all codes are collected, we pack the codes four words at a time by shifting the second word by two bits, the third word by four bits, and the fourth word by six bits. Then we perform a logical "or" of these four words, as illustrated in Figure 3. This scheme minimizes the total number of shifts required to pack all two-bit sequences because four-byte shifts may be carried out in parallel on 32-bit architectures. A similar technique is applied to

Table II. Performance Comparison of Two Alignment Schemes

	Non-aligned PBPM (μs)		Separate aligned tapes (μs)		Single aligned tape (μs)	
	Compress	Decompress	Compress	Decompress	Compress	Decompress
average	12.99	10.94	14.94	10.94	16.89	18.26
stdev.	3.05	3.08	1.79	2.20	3.77	4.44

the index tape, which contains four-bit sequences. During decompression, the procedure is reversed.

4.4 Word Alignment

Some processors (e.g., many ARM and PowerPC processors) do not support high-performance loads or stores of nonaligned data objects; for example, an access to a four-byte word with an address that is not evenly divisible by four may be illegal or impose substantial performance penalties. In the best case, such accesses trigger interrupts that must be handled by the processor support library, greatly degrading performance. As mentioned in the previous section, the data tape consists of sequences with different lengths, that is, one-byte, two-byte, and four-byte. This poses no problem for processors that support high-performance misaligned accesses. However, it may impose substantial overhead for other architectures commonly used in embedded systems. To solve this problem, we implemented two alignment schemes and compared their performance.

(1) *Separate aligned tapes*. Instead of having one data tape, we maintain separate data tapes for one-byte, two-byte, and four-byte data. One of the three data tapes is placed at a fixed position in the output buffer. At the end of the compression process the other two tapes are copied to the end of the first data tape. As long as the later two tapes are short, little copying is required and the performance overhead of this approach is low. Based on our analysis, the longest data tape is generally the four-byte data tape because the no-match pattern is the second most frequent pattern.

(2) *Single word-aligned tape*. Only one flat data tape is maintained. However, data are written into this tape in a word-aligned manner. Each word of this tape consists of one of the following: four single bytes, two two-byte sequences, or one four-byte word. Three additional pointers are maintained to record the positions of the next available one-byte, two-byte, and four-byte locations. Data are written to the tape at the location indicated by the corresponding pointer depending on the length, that is, one-byte, two-byte, or four-byte. This implies that data may be written to the tape out-of-order. The same pointers are maintained and the same procedure is followed during decompression. Every time data are written to the tape, the corresponding pointer must be checked and updated, reducing performance. However, it is never necessary to copy tapes to new positions when using this technique.

We compared these alignment techniques against the original PBPM algorithm, which does not consider the alignment problem. The three versions were executed on a Linux Workstation with a 2GHz AMD Athlon XP 2800+ processor. This processor does not require word-aligned load and store instructions and hence does not incur exceptions on misaligned memory accesses. Table II

shows that the first technique, which maintains separate data tapes, imposes smaller performance overhead than the second one, which maintains a single word-aligned data tape. Compared to the original implementation of PBPM, on average, the first technique increases compression time by 15% and does not affect decompression time. The second technique increases compression time by 30% and decompression time by 67%. Therefore, for architectures that suffer high performance overheads on misaligned accesses, the copying-based alternative implementation should be used; otherwise, the original implementation of PBPM should be used.

5. ADAPTIVE COMPRESSED MEMORY MANAGEMENT

As described in Section 3, CRAMES compresses the swapped-out data a page at a time and stores them in a special compressed RAM device. Upon initialization, the compressed RAM device only requests a small memory chunk; as system memory requirements grow, additional memory chunks are requested. The compressed RAM device also dynamically decreases its size when compressed pages are freed. A compressed page may be freed in two circumstances: (1) the page only belongs to one running process 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 preceding dynamic memory allocation strategy works well when the system is not under extreme memory pressure. However, its performance degrades when memory is dangerously low: 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 one page of physical memory for its compressed memory region, it would be able to swap out multiple pages, allowing the application to proceed. However, CRAMES is in contention with the application, resulting in an *online memory compression deadlock*, that is, a situation in which both the CRAMES and other application processes wait for memory resources that can only be made available by progress of the other process.

We will use an example to illustrate online memory compression deadlock. In Figure 4, assuming an embedded system with ten pages of physical RAM, process A and process B each have 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 dataset of process A starts to grow and it requests one more page. Since only a single page of memory is now available, the kernel starts swapping. Page `b1` from process B is swapped out and compressed by CRAMES. After compression, the size of `b1` is reduced to 80% of its original size. (2) Process A requests one more page. The kernel continues its attempt to swap out pages from process B.

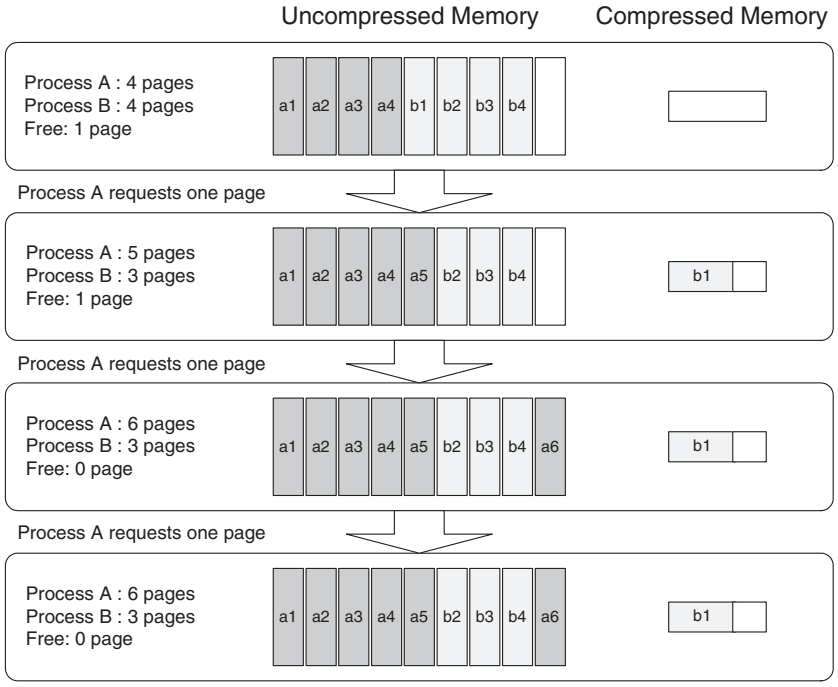

Fig. 4. Online memory compression deadlock.

Unfortunately, since the compression ratio of the last swapped-out page `b1` was high, none of the pages of process B can fit into the compressed area. Therefore, the kernel allocates the last free page in the uncompressed area to process A. (3) Process A requests one more page. If CRAMES were able to allocate even a single page from the uncompressed area, it could compress multiple pages and proceed. However, no additional free pages are available because process A reserved the last free page in the previous step. This results in a deadlock scenario. The allocation request from process A is in contention with CRAMES for memory pages and neither may proceed.

To avoid online memory compression deadlock, a compressed RAM device must predictively request additional memory, that is, it cannot wait until no allocated chunks have space for incoming data. This subtle issue comprises one of the differences 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 full, so that the compressed memory is always available to applications.

We propose the following scheme to prevent online 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 be denied if the system memory is dangerously low. However, even if this first request is denied, subsequent invocations of CRAMES will generate

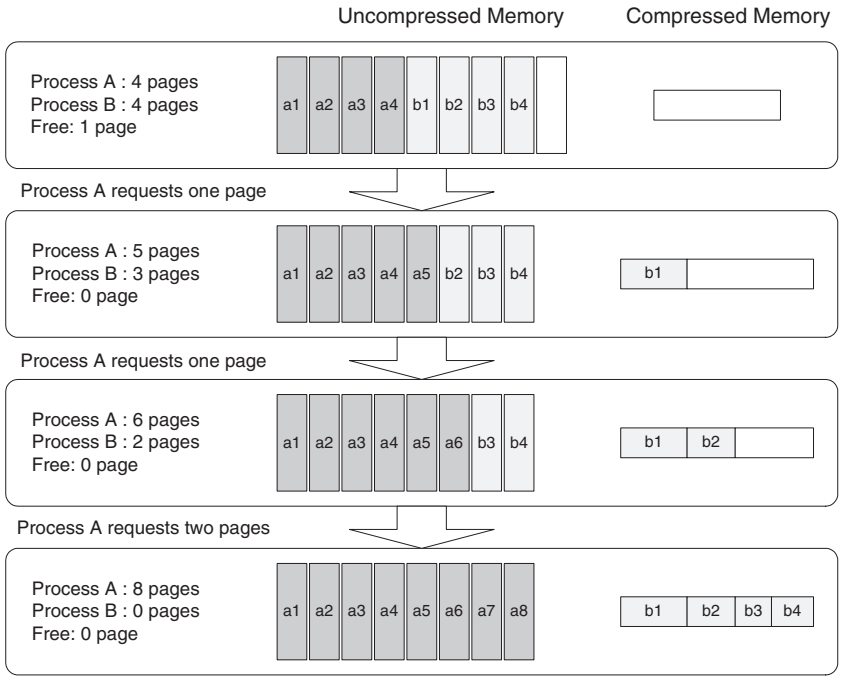

Fig. 5. Deadlock avoided.

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 generally succeed eventually.

Figure 5 illustrates the operation of CRAMES, which preemptively allocates memory in order to avoid online memory compression deadlock. After the first page `b1` from process B is swapped out and compressed, CRAMES realizes that the total compressed area is 80% full, exceeding the fill ratio. As a consequence, CRAMES requests pages from the kernel immediately. Since this allocation request occurs before that of 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 is able to compress the rest of the pages from process B because the available memory in the compressed area is sufficient. Therefore, the subsequent allocation requests from process A are successful.

We have experimented with different fill ratios and found that 7/8 is sufficient to allow the maximum improvement in application available memory. This ratio also results in little memory waste. Evaluation of this method on a portable embedded system is presented in Section 6.2.

6. EVALUATION

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

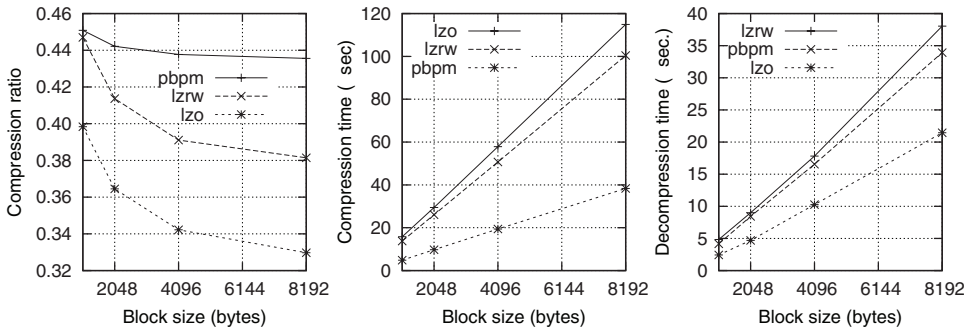

Fig. 6. Compression ratios and speeds of PBPM, LZO, and LZRW.

(1) Does PBPM provide a competitive compression ratio yet have 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. It has a 400 MHz Intel XScale 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. We replaced the SL-5600 battery with an Agilent E3611A direct-current power supply. Current was computed by measuring the voltage across a 250 mΩ, 5W, Ohmite Lo-Mite 15FR025 molded silicone wire element resistor in series with the power supply. This resistor was designed for current sensing applications. Voltage measurements were taken using a National Instruments 6034E data acquisition board attached to the PCI bus of a host workstation running Linux.

6.1 Quality and Speed of the PBPM Algorithm

We compared the compression ratio and speed of the PBPM algorithm with two other compression algorithms that have been used for online memory compression: LZO and LZRW. Both LZO and LZRW have different levels of compression that provide different compression ratios and speeds. We compared PBPM with the fastest mode of LZO and LZRW1-A, which is among the fastest of the available LZRW algorithms.

Figure 6 illustrates the compression ratios (compressed block size divided by original block size) and execution times of the 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 online memory compression is a symmetric application, that is, a memory page is decompressed exactly once

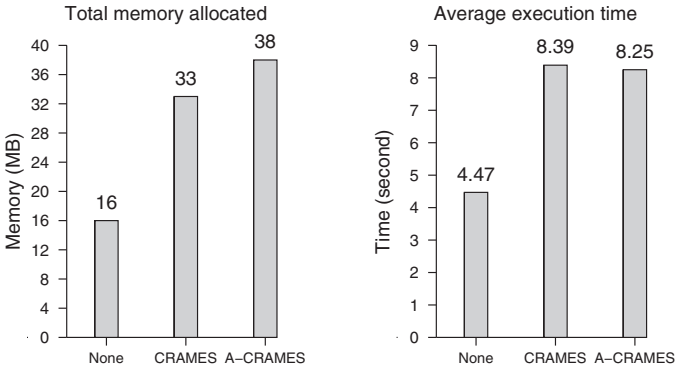

Fig. 7. Performance of A-CRAMES.

every time it is compressed. Therefore, the overall, symmetric performance of a compression algorithm is the critical metric. For online memory compression, the block size is page size, that is, 4,096 KB. At the block size of 4,096 KB, PBPM achieves a 200% speedup over both LZO and LZRW. The compression ratio achieved by PBPM (44%) is competitive with that of LZO (34%) and LZRW (39%). We believe that PBPM is especially suitable for online memory compression because of its high symmetric compression speed and good compression ratio when used on swap data.

6.2 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 memory in 1 MB blocks. The memory allocated is then filled with randomized values and mixed with zero runs with similar patterns to those observed in real swap traces. Requests repeat until a request fails. Memeater was executed on a Zaurus with 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 7 presents the total memory allocated and average execution times (from five samples) under the three system settings. Without CRAMES, the system could only provide 16 MB of memory to memeater. With CRAMES, 33 MB of memory were provided and the execution time was linear to the amount of memory allocated, that is, no delay was observed. Furthermore, when adaptive memory management is enabled (A-CRAMES), 38 MB of memory (13% more) were allocated without performance penalty. These results support our claim in Section 5 that A-CRAMES helps to prevent online memory compression deadlock.

6.3 Probability of Running out of Memory

The overall memory compression ratio is influenced by the running applications. It is possible that under some workloads, a set of applications will write

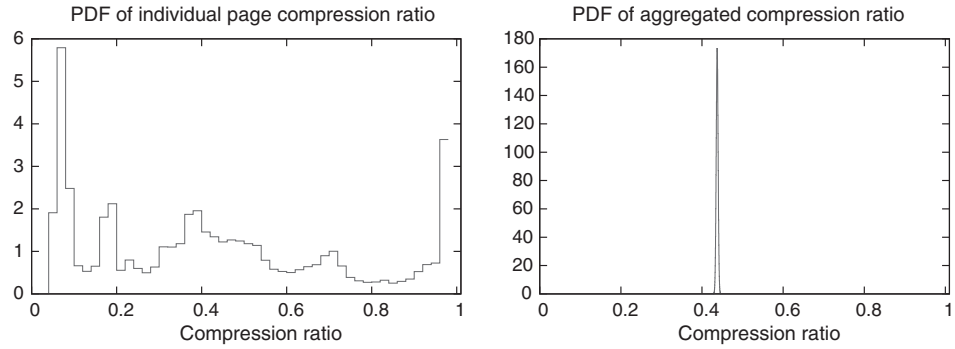

Fig. 8. PDFs of individual page compression ratios and aggregated compression ratios.

relatively incompressible data to memory, preventing CRAMES and PBPM from achieving the predicted aggregate system-wide compression ratio. This would prevent running applications from allocating additional memory. The probability of this happening is equivalent to the probability of the aggregate compression ratio of in-RAM data exceeding the target compression ratio when deciding the amount of physical RAM in the embedded system. We have taken two approaches to estimate the potential danger of this phenomenon. First, we ran numerous sets of applications on the target system to determine stability. Second, we analytically estimated the probability of the target compression ratio being exceeded. We will now describe this analysis.

The probability of exceeding the target compression ratio can be approximated using statistical techniques. We first estimated the Probability Density Functions (PDFs) of page compression ratios based on trace data extracted when using CRAMES for a wide range of applications and running PBPM on each page. As long as the correlation between the compression ratios of independent pages is weak and the mean and variance of the random process are finite, one can determine the aggregate system-wide compression ratio PDF by starting from one-page compression ratio PDF and repeatedly convolving by the same function. This process should be repeated for each page of compressed memory in the system. In practice, the number of pages in a system is large, for example, 16,384 4KB pages for 32MB of compressed data, assuming an average compression ratio of 50%. Note that as the number of convolutions becomes large, the resulting probability distribution takes on an approximately Gaussian distribution with low variance [Leon-Garcia 1989]. It is important to note that the aggregate distribution will have low variance regardless of variance of the individual pages due to the Law of Large Numbers.

Figure 8 shows the probability distribution of independent compression ratio and aggregated compression ratio on our trace data. The results indicate that the probability of the aggregated compression ratio exceeding our estimate is extremely low. In fact, the probability of the aggregate compression ratio being within 2% of 43.8% is 100.0000%. The probability of exceeding our target compression ratio of 50% can be estimated as the area under the aggregate PDF for compression ratios higher than 50%. This probability is 3.40×10^{-158}. In

summary, although there can be no guarantee that a particular set of applications produce pages with an aggregate compression ratio below a particular target compression ratio, experiments and analysis indicate that this is unlikely to pose a problem for CRAMES and PBPM. If a problem does occur, it will most likely be due to violation of the assumption of page compression ratio independence. We did not encounter problems during experiments, testing, or commercialization.

6.4 Overall Performance of CRAMES

We have experimentally demonstrated 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 [Yang et al. 2005]. This implies that with CRAMES, the same hardware platform can easily support new applications with larger datasets. This also implies that an embedded system could be designed with less RAM and still support desired applications, with some potential performance and energy consumption overheads. When the system is under substantial memory pressure, PBPM and adaptive memory management minimize these overheads.

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. The memory allocated to the kernel module cannot be swapped out and therefore is not compressed by CRAMES. This allows a fair comparison. With reduced physical RAM, we measured and compared the runtimes, power consumptions, and energy consumptions of four batch benchmarks, that is, three applications from MediaBench [Lee et al.] (Adpcm, Jpeg, and Mpeg2) and a 512 by 512 matrix multiplication application. Table III shows the performance numbers of benchmarks running without compression, with LZO compression, and with PBPM compression under different memory constraints. Note that adaptive memory management was enabled for both LZO and PBPM to ensure a fair comparison. In our experiments, each benchmark was executed five times; the average results are reported. The 90% confidence interval for each case ranges from 0.09% to 19.87% of the reported values.

As shown in Table III, both LZO and PBPM impose only small power consumption overheads on the applications. When the system memory is not reduced dramatically, the performance overheads of both compression algorithms are insignificant. However, the performance difference between LZO and PBPM becomes obvious when the system is under tight memory constraints. In particular, when application available RAM was reduced from 20MB to 8MB, without CRAMES all benchmarks suffered from significant performance degradation; the 512 by 512 matrix multiplication could not even execute due to memory constraints. However, with CRAMES, all benchmarks were able to execute with only slight performance and energy consumption penalties. Compared to the base case in which application available RAM is 20MB and CRAMES is not used, PBPM compression results in an average performance penalty of 0.2%

Table III. Performance of CRAMES with PBPM and Adaptive Allocation

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 (seconds)												
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 (Watts)												
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 (Joules)												
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

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%.

Interestingly, we observe that when application available RAM was 20MB and CRAMES was used, the performance of matrix multiplication was actually improved by 5% on average, comparing to the base case. We believe the cause of the improved performance is as follows. Unlike the other three applications, the memory requirements of 512 by 512 matrix multiplication imposes great memory pressure on the system. Without compression, to keep the available memory above a certain threshold, the kernel must reclaim memory from buffer caches by either using clean pages or evicting dirty pages, which results in significant performance overhead. However, CRAMES responds to reduced memory by compressing pages with minimum overheads. Therefore, when the matrix multiplication program is executing, compression keeps the free memory in the system high enough so that the kernel needs to do little reclamation from buffer caches.

7. CONCLUSIONS

High-performance OS-controlled memory compression can assist embedded system designers to optimize hardware design for typical software memory requirements while also supporting applications with larger datasets. In this article, we presented PBPM, an efficient compression algorithm for use in OS-controlled memory compression. PBPM has compression ratios that are competitive with existing algorithms used in online memory compression. However,

its performance is significantly better when system memory is under tight constraints. We also presented an adaptive compressed memory management scheme to prevent online memory compression deadlock, thereby further increasing the amount of usable memory. Experimental results indicate that using these two techniques in our memory compression framework allows applications to execute with only slight penalties even when available RAM is reduced to 40% of its original size. These benefits require no changes to applications or hardware.

REFERENCES

BENINI, L., BRUNI, D., MACII, A., AND MACII, E. 2002. Hardware-Assisted data compression for energy minimization in systems with embedded processors. In *Proceedings of the Design, Automation and Test in Europe Conference*.

BENINI, L., BRUNI, D., MACII, A., AND MACII, E. 2004. Memory energy minimization by data compression: Algorithms, architectures and implementation. *IEEE Trans. VLSI Syst. 12,* 3, 255–267.

COMPRESSED CACHING. Compressed caching in Linux virtual memory. `http://linuxcompressed.sourceforge.net`.

CORTES, T., BECERRA, Y., AND CERVERA, R. 2000. Swap compression: Resurrecting old ideas. *Softw. Pract. Exper. J. 30*, 567–587.

DOUGLIS, F. 1993. The compression cache: Using on-line compression to extend physical memory. In *Proceedings of the USENIX Conference*. 519–529.

KJELSO, M., GOOCH, M., AND JONES, S. 1996. Design and performance of a main memory hardware data compressor. In *Proceedings of the Euromicro Conference*. 423–430.

KJELSO, M., GOOCH, M., AND JONES, S. 1999. Performance evaluation of computer architectures with main memory data compression. *J. Syst. Archit. 45*, 571–590.

LEE, C., POTKONJAK, M., AND SMITH, W. H. M. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. `http://cares.icsl.ucla.edu/MediaBench`.

LEFURGY, C., PICCININNI, E., AND MUDGE, T. N. 2000. Reducing code size with run-time decompression. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. 218.

LEKATSAS, H., HENKEL, J., AND WOLF, W. 2000. Code compression for low power embedded system design. In *Proceedings of the Design Automation Conference*. 294–299.

LEON-GARCIA, A. 1989. *Probability and Random Processes for Electrical Engineering*. Addison-Wesley.

MOORE, K. E. 2003. Compressing memory management in a device. U.S. patent, Hewlett-Packard Development Company, LP. May.

OBERHUMER, M. F. LZO real-time data compression library. `http://www.oberhumer.com/opensource/lzo`.

RIZZO, L. 1997. A very fast algorithm for RAM compression. *Oper. Syst. Rev. 31,* 2, 36–45.

ROY, S., KUMAR, R., AND PRVULOVIC, M. 2001. Improving system performance with compressed memory. In *Proceedings of the Parallel and Distributed Processing Symposium*.

RUSSINOVICH, M. AND COGSWELL, B. 1996. RAM compression analysis. `http://ftp.uni-mannheim.de/info/OReilly/windows/win95.update/model.html`.

SHAW, C., CHATTERJI, D., SEN, P. M. S., ROY, B. N., AND CHAUDURI, P. P. 2003. A pipeline architecture for encompression (encryption + compression) technology. In *Proceedings of the International Conference on VLSI Design*.

TREMAINE, B., FRANASZEK, P. A., ROBINSON, J. T., SCHULZ, C. O., SMITH, T. B., WAZLOWSKI, M., AND BLAND, P. M. 2001. IBM memory expansion technology. *IBM J. Res. Devel. 45,* 2, 271–285.

TUDUCE, I. C. AND GROSS, T. 2005. Adaptive main memory compression. In *Proceedings of the USENIX Conference* 237–250.

VAHALIA, U. 1996. *UNIX Internals: The New Frontiers*. Prentice-Hall, NJ.

WILSON, P. R., KAPLAN, S. F., AND SMARAGDAKIS, Y. 1999. The case for compressed caching in virtual memory systems. In *Proceedings of the USENIX Conference*. 101–116.

XU, X. H., CLARKE, C. T., AND JONES, S. R. 2004. High performance code compression architecture for the embedded ARM/Thumb processor. In *Proceedings of the Conference on Computing Frontiers*. 451–456.

YANG, L., DICK, R. P., LEKATSAS, H., AND CHAKRADHAR, S. 2005. CRAMES: Compressed RAM for embedded systems. In *Proceedings of the International Conference Hardware/Software Codesign and System Synthesis*.

YANG, L., LEKATSAS, H., AND DICK, R. P. 2006. High-performance operating system controlled memory compression. In *Proceedings of the Design Automation Conference*. 701–704.

ZIV, J. AND LEMPEL, A. 1977. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory 23,* 3, 337–343.

Received February 2007; revised July 2007; accepted July 2008