On-Line Memory Compression for Embedded Systems

LEI YANG, ROBERT P. DICK
Northwestern University, Evanston
{l-yang,dickrp}@northwestern.edu
HARIS LEKATSAS, SRIMAT CHAKRADHAR
NEC Laboratories America, Princeton
{lekatsas, chak}@nec-labs.com

Memory is a scarce resource during embedded system design. Increasing memory often increases packaging costs, cooling costs, size, and power consumption. This paper presents 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. In addition to presenting a novel framework for dynamic data memory compression and in-RAM filesystem compression in embedded systems, this work identifies the software-based compression algorithms that are most appropriate for use in low-power embedded systems.

Categories and Subject Descriptors: D.4.2 [**Storage Management**]: Virtual memory; C.3 [**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

This is an extended version of an article that appeared in the International Conference on Hardware/Software Codesign and System Synthesis in September 2005 [Yang et al. 2005].

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 [Yokotsuka 2004]. 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 [hpi]. 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 work described in this paper 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 paper. 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.

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

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

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 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.

1.2 Technique 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?

This paper answers these questions and evaluates the soundness of CRAMES. 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 compress them. Multiple compression techniques and memory allocation methods were experimentally evaluated; 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 [emb]. 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.

1.3 Paper Organization

The rest of this paper is organized as follows. Section 2 summarizes related techniques and their contributions. Section 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 4 discusses the implementation of CRAMES as a Linux kernel module. Section 5 presents the experimental set-up, describes the workloads, and presents the experimental results in detail. Finally, Section 6 concludes the paper.

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 paper. The primary goal of filesystem compression is to save disk space and furthermore, to reduce disk I/O by transferring compressed data.

2.1 Compress Both Code and Data

IBM MXT [Tremaine et al. 2001] 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 [Benini et al. 2002] 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 [Kjelso et al. 1996] 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.

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 [Lekatsas et al. 2000; Xu et al. 2004; Bell et al. 1990; Shaw et al. 2003] 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 [thu 1995].

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.

2.3.1 *Compressed Caching.* Compressed caching [Douglis 1993; Russinovich and Cogswell 1996; Wilson et al. 1999; com] 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 [Douglis 1993] 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 [Kjelso et al. 1999] 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 [Russinovich and Cogswell 1996] presented a thorough analysis of the compression algorithms used in compressed caching. Wilson, Kaplan, and Smaragdakis [Wilson et al. 1999] 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.

2.3.2 *Swap Compression.* Swap Compression [Rizzo 1997; Cortes et al. 2000; Roy et al. 2001; Tuduce and Gross 2005] compresses swapped out pages and stores them in a software cache in RAM. Cortes, Eles, and Peng [Cortes et al. 2000] 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 [Roy et al. 2001] 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. [Rizzo 1997] 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.

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 [Cramfs] 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 metadata 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 [Cloop] is a Linux kernel module that enables compressed loopback [Bovet and Cesati 2002] 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 [CBD] 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.

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* [ram] 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 [Chen et al. 2003] 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.

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 5).

3. CRAMES DESIGN

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.

3.1 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 3.2.2, 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.

3.2 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 3.1.

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 1 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 1). 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.

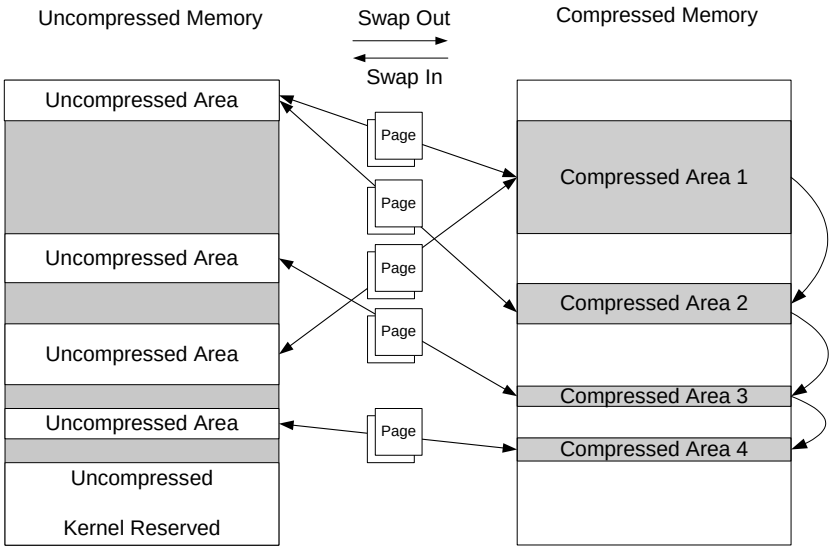

Fig. 1. Logical structure of the swapping mechanism on compressed RAM device. The two big boxes represent the same physical memory region.

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

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 I. Evaluated 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

Table II. Memory overhead of compression algorithms

be accomplished with existing techniques and tools, e.g., JFFS2 [Woodhouse 2001] with demand paging. These techniques may be used in combination with CRAMES.

3.2.2 *CRAMES and Block-based Data Compression .* To ensure good performance for CRAMES, appropriate compression algorithms must be identified and/or designed. Fortunately, 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 [lzo], and RLE (Run Length Encoding).

Figure 2 illustrates the compression ratios and execution times of the evaluated algorithms and Table II 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 appears to be the best block compression algorithm for dynamic data compression in low-power embedded systems due to its good 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 was chosen as the default compression algorithm in CRAMES.

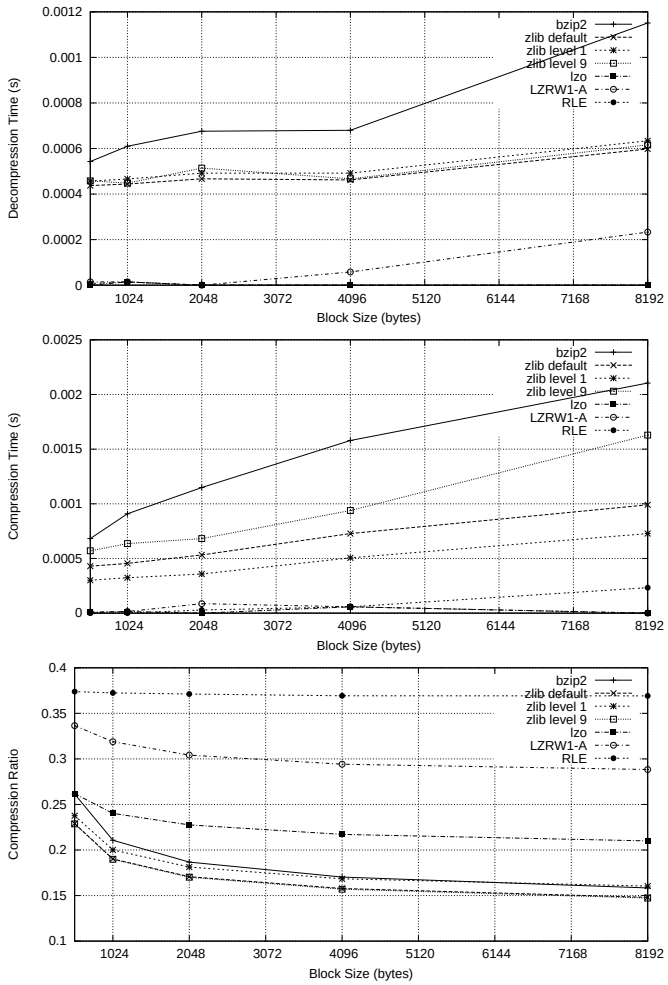

Fig. 2. Comparison of evaluated compression algorithms

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.

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

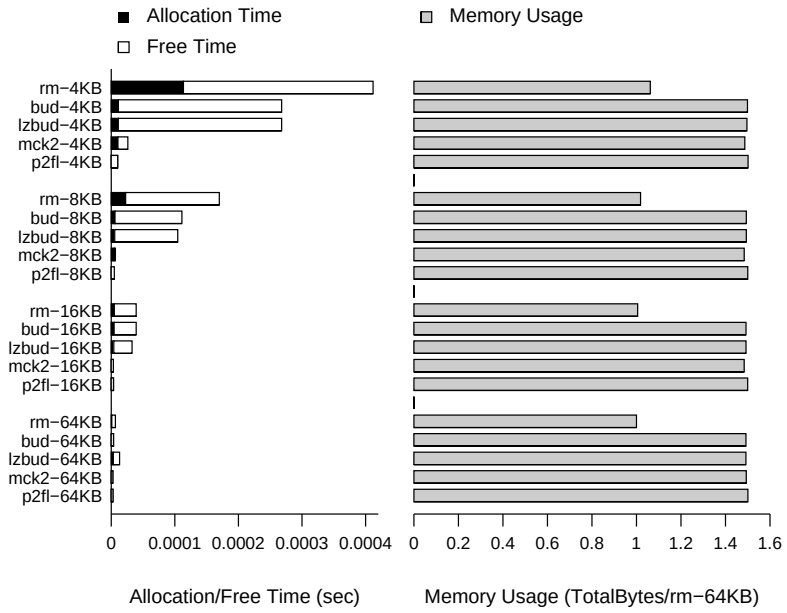

Fig. 3. Memory usage of evaluated memory allocation methods

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 [McKusick and Karels 1988] (mck2), Buddy System [Peterson and Norman 1977] (bud), and Lazy Buddy Algorithm [Lee and Barkley 1989; Barkley and Lee 1989] (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 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-4 KB` 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 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 Using CRAMES with the Filesystem

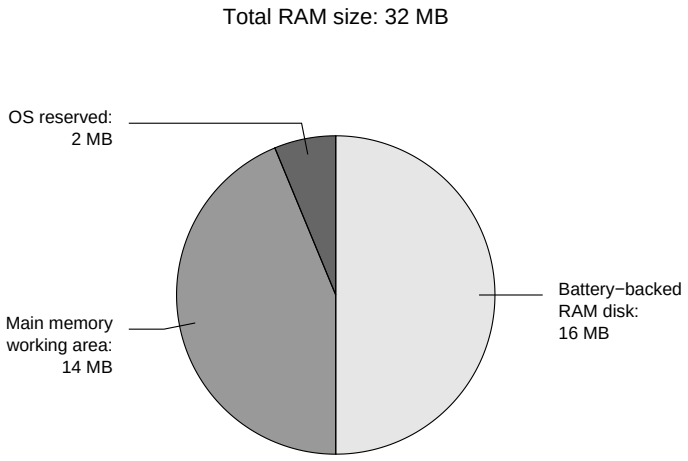

Fig. 4. RAM regions in a Sharp Zaurus SL5600

As illustrated in Figure 4, 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 paper, 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 [Cramfs] is a compressed filesystem targeting embedded systems, but it is read-only. JFFS2 [Woodhouse 2001] 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.

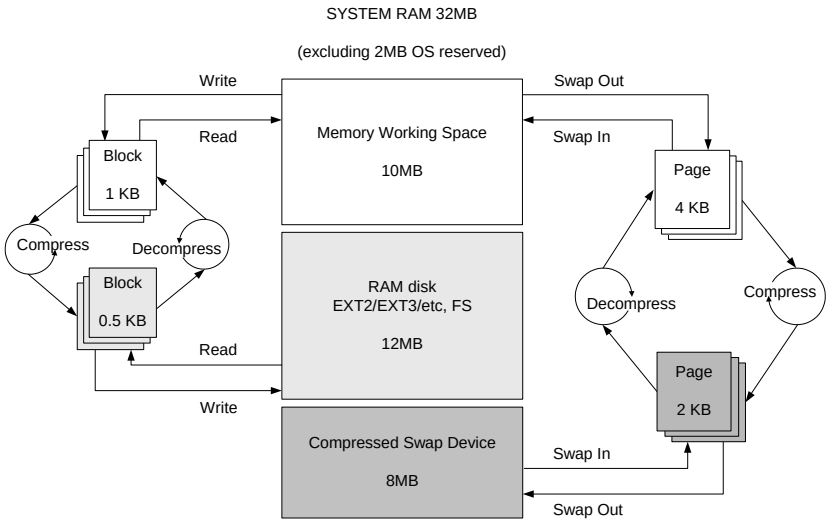

Fig. 5. A possible system RAM partition

Figure 5 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 $8 \div 0.5 = 16$ MB. The maximum total memory addressable is therefore $10 + 16 = 26$ MB. If the average compression ratio for the RAM disk is 60%, the total filesystem storage available for the system is $12 \div 0.6 = 20$ MB. Thus, the

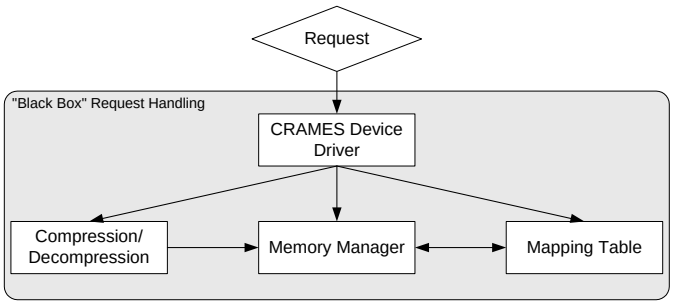

Fig. 6. CRAMES device request handling

maximum RAM capacity of the system is expanded to 26+20+2 = 48 MB with little degradation in performance or power consumption (as shown in Section 5). This example assumes fixed-size compressed RAM devices to simplify the explanation; however, CRAMES supports dynamic, automatic compressed RAM resizing.

4. CRAMES IMPLEMENTATION

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) 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.

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) 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 6 depicts the logical structure of the

[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.

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

Table III. Mapping table in CRAMES

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.

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 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.

- `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 III 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.

4.1.2 *Memory Manager .* Section 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 3.2.3, 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.

4.1.3 *Request Handling Flow.* Figure 7 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 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.

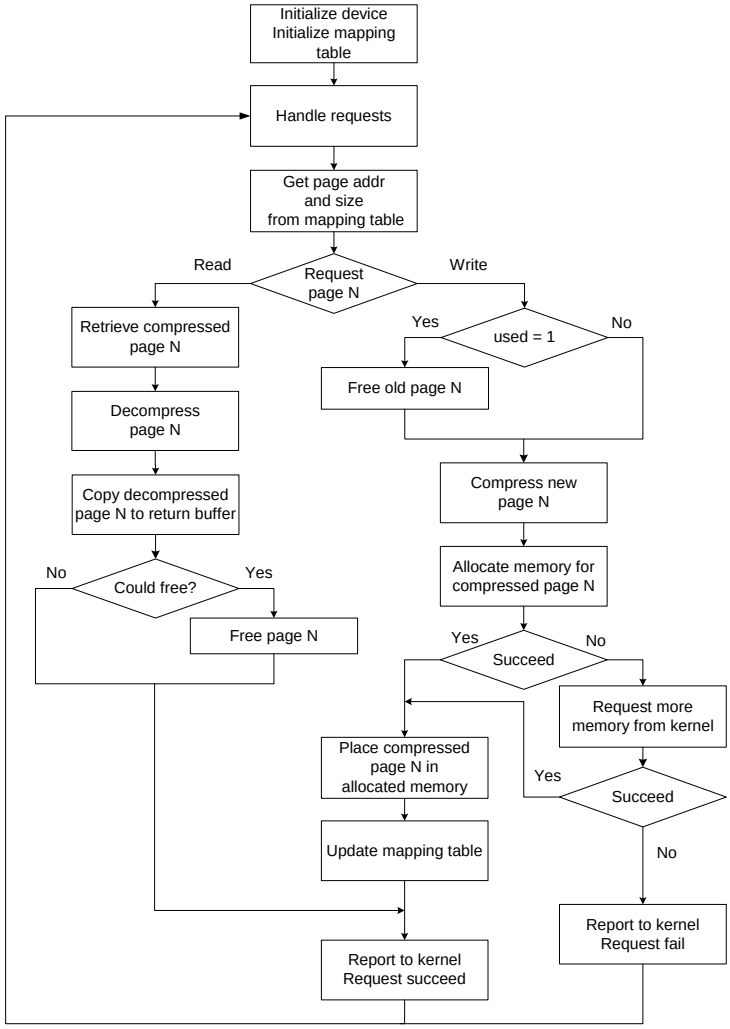

Fig. 7. Handling request in CRAMES device

4.2 CRAMES and RAM Disk Comparison

Figure 8 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.

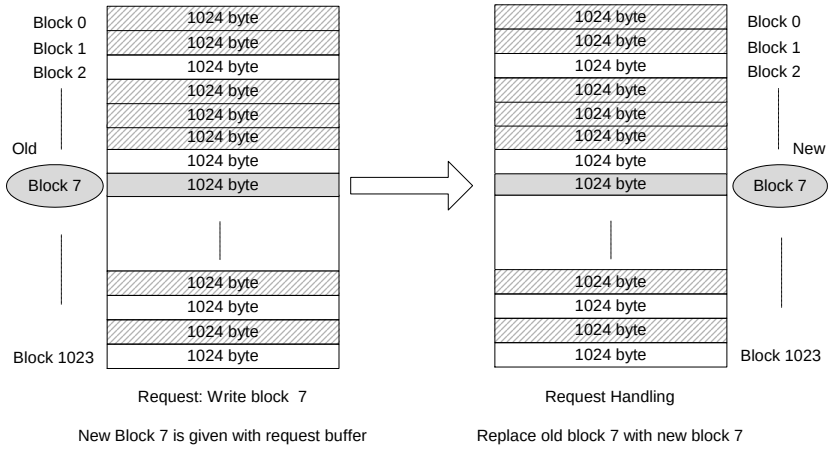

Fig. 8. Linear, fixed-size blocks in RAM disk

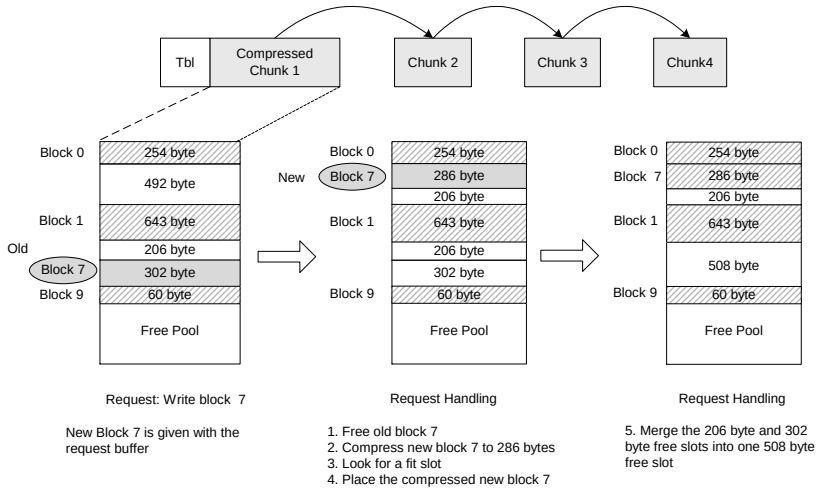

Fig. 9. Compressed blocks in CRAMES device

Figure 9 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,

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

Table IV. Performance, power consumption, and energy consumption for filesystem experiments

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.

5. CRAMES EVALUATION

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 [emb]. It has a 400 MHz Intel XScale PXA250 processor [xsc 2002], 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 Instruments 6034E data acquisition board attached to the PCI bus of a host workstation running Linux.

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 IV. 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 IV 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.

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.

5.2.1 *Evaluating CRAMES with the original RAM size.* The benchmarks used to evaluate CRAMES contain three applications from the Mediabench benchmark suite [Lee et al.], one matrix multiplication program with different matrix sizes, ten common GUI applications provided with Qtopia [qto] 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 V and Figure 10 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.

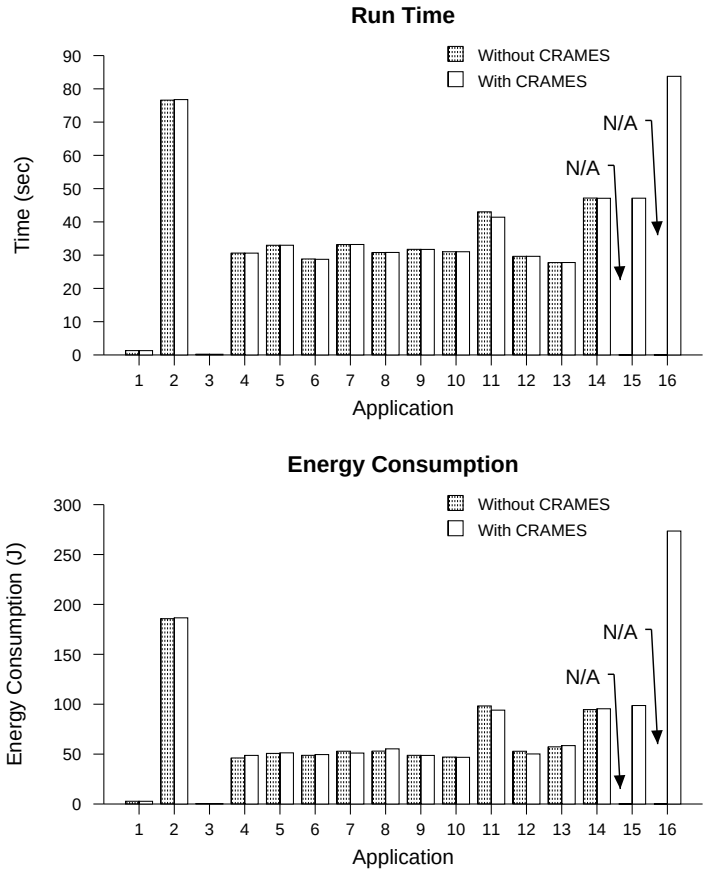

Fig. 10. Performance and energy consumption impact of using CRAMES for swapping

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.

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 5.2.1.

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

Table V. Performance, power consumption, energy consumption, and compression ratio for swapping experiments

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 [Bovet and Cesati 2002] and therefore is not compressed by CRAMES. This guarantees the fairness of our comparison.

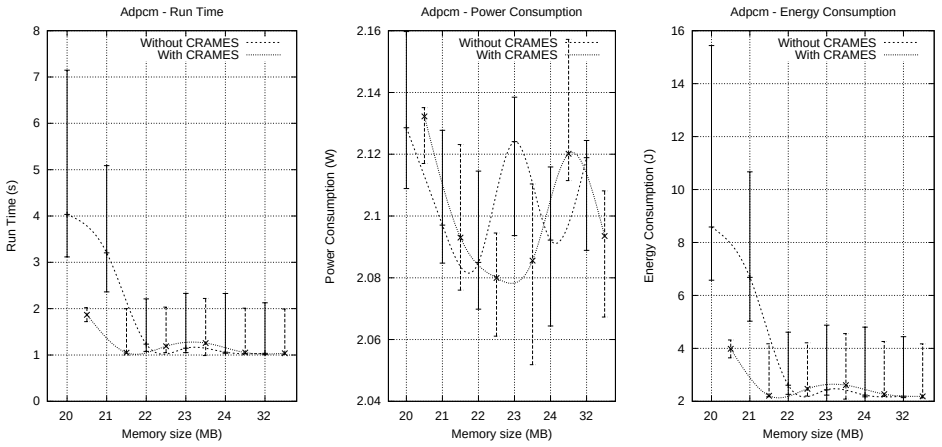

Fig. 11. Performance and energy Consumption of adpcm

Figure 11, 12, 13, and 14 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 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 11, 12, and 13 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

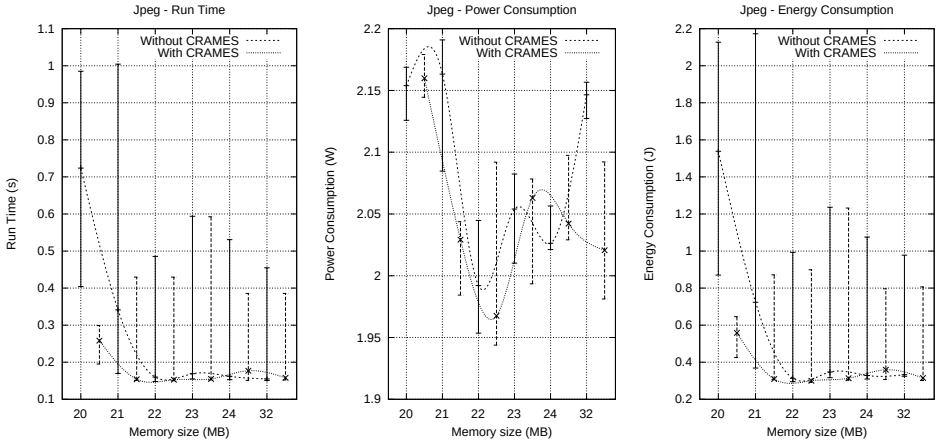

Fig. 12. Performance and energy consumption of jpeg

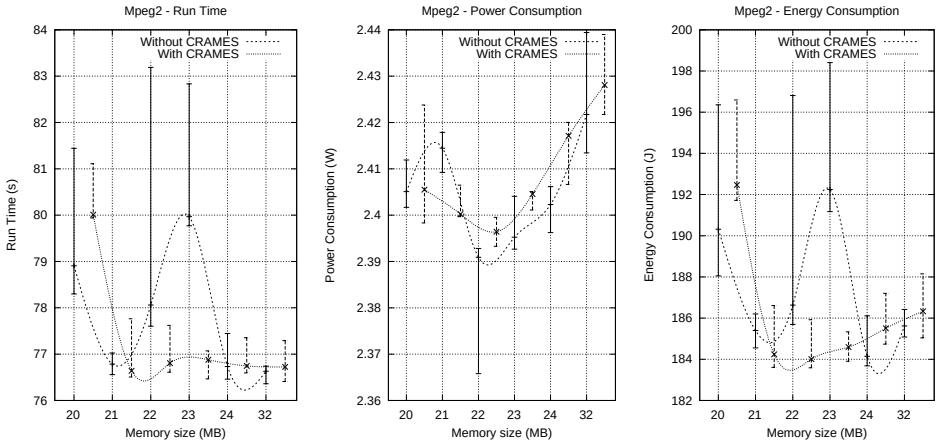

Fig. 13. Performance and energy Consumption of mpeg2

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, 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.

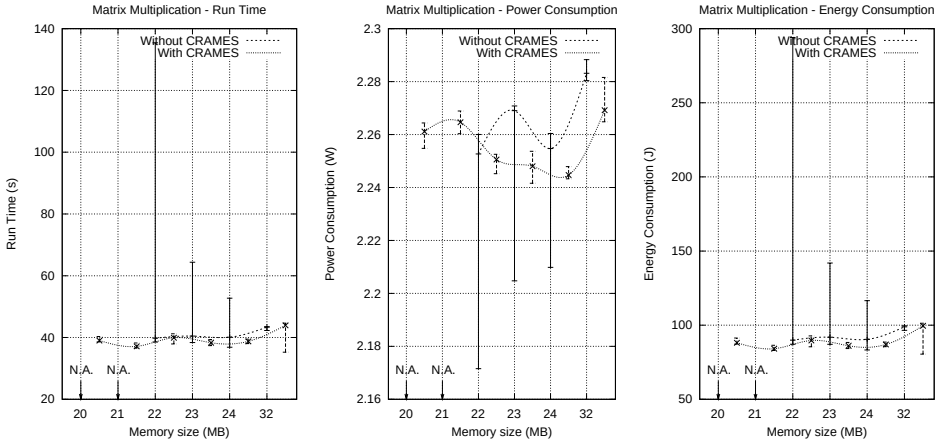

Fig. 14. Performance and energy consumption of matrix multiplication

Figure 14 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 [xsc 2002].

6. CONCLUSIONS

In this paper, 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.

REFERENCES

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

HP online shopping. http://www.shopping.hp.com.

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

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

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

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

1995. *An introduction to Thumb*.

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

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

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

Benini, L., Bruni, D., Macii, A., and Macii, E. 2002. Hardware-assisted data compression for energy minimization in systems with embedded processors. In *Proc. Design, Automation & Test in Europe Conf.*

Bovet, D. P. and Cesati, M. 2002. *Understanding the Linux Kernel*, Second ed. O'Reilly & Associates, Inc.

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

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

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

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

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

Douglis, F. 1993. The compression cache: Using on-line compression to extend physical memory. In *Proc. USENIX Conf.*

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

Kjelso, M., Gooch, M., and Jones, S. 1999. Performance evaluation of computer architectures with main memory data compression. In *J. Systems Architecture*. Vol. 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.

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

Lekatsas, H., Henkel, J., and Wolf, W. 2000. Code compression for low power embedded system design. In *Proc. Design Automation Conf.* 294–299.

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

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

Rizzo, L. 1997. A very fast algorithm for RAM compression. *Operating Systems Review 31,* 2 (Apr.), 36–45.

Roy, S., Kumar, R., and Prvulovic, M. 2001. Improving system performance with compressed memory. In *Proc. Parallel & Distributed Processing Symp.*

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 *Proc. International Conf. 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 Journal of Research and Development 45,* 2 (Mar.).

Tuduce, I. C. and Gross, T. 2005. Adaptive main memory compression. In *Proc. USENIX Conf.*

Wilson, P. R., Kaplan, S. F., and Smaragdakis, Y. 1999. The case for compressed caching in virtual memory systems. In *Proc. USENIX Conf.* 101–116.

Woodhouse, D. 2001. JFFS: The journalling flash file system. In *Ottawa Linux Symp.* RedHat Inc.

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

Yang, L., Dick, R. P., Lekatsas, H., and Chakradhar, S. 2005. CRAMES: Compressed RAM for embedded systems. In *Proc. Int. Conf. Hardware/Software Codesign and System Synthesis.*

Yokotsuka, M. 2004. Memory motivates cell-phone growth. *Wireless Systems Design.*

Received November 2005; revised May 2006; accepted August 2006