CRAMES: Compressed RAM for Embedded Systems

Lei Yang† Robert P. Dick† Haris Lekatsas‡ Srimat Chakradhar‡

†l-yang, dickrp@northwestern.edu
Northwestern University
Evanston, IL 60208

‡lekatsas, chak@nec-labs.com
NEC Laboratories America
Princeton, NJ 08540

ABSTRACT

Memory is a scarce resource in many embedded systems. Increasing memory often increases packaging and cooling costs, size, and energy consumption. This paper presents CRAMES, an efficient software-based RAM compression technique for embedded systems. The goal of CRAMES is to dramatically increase effective memory capacity without hardware 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. Execution time and energy consumption for a broad range of examples increase only slightly, by averages of 0.35% and 4.79%. In addition, this work identifies the software-based compression algorithms that are most appropriate for 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

Keywords

Embedded system, memory, compression

1. Introduction and Motivation

Modern embedded systems, e.g., personal digital assistants (PDAs) and mobile phones, are growing increasingly complex. 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 [1]. Although memory price has historically dropped with time, adding memory frequently results in increased packaging and cooling costs, size, and energy 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), hx2755 differs from hx2415 only by making 2.2 times as much memory available to the user [2]. In addition, as embedded systems support new applications, their working data sets often increase in size, exceeding original estimates of memory requirements. Redesigning hardware is not desirable as it may substantially increase time-to-market and design costs.

We propose a new software-based RAM compression technique, named CRAMES, that increases effective memory capacity without adding physical memory. RAM compression for embedded systems is a complex problem that raises several questions. Does the 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 the technique be evaluated for use in embedded systems?

This paper answers these questions and evaluates the quality of CRAMES. To minimize the performance and energy consumption impact, CRAMES takes advantage of the operating system (OS) virtual memory swapping mechanism to decide which pages to compress and when to compress them. Multiple compression techniques and memory allocation methods were experimentally evaluated; the most promising 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 from performance or energy consumption penalties as a result of its use. In addition to data set compression, CRAMES may also be used 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. This embedded system's architecture is similar to that of modern smart 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 evaluated our technique using well-known batch applications 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 the memory capacity with minimal performance and energy consumption costs.

CODES+ISSS'05, Sept. 19–21, 2005, Jersey City, New Jersey, USA.

The rest of this paper is organized as follows. Section 2 summarizes the contributions of related work. Section 3 describes the proposed memory compression technique and elaborates on the tradeoffs involved in the design of CRAMES. Important design principles are proposed for software-based RAM compression techniques. Section 4 discusses the implementation of CRAMES as a Linux kernel module. Section 5 describes the experimental setup, workloads, and experimental results in detail. Finally, Section 6 concludes the paper.

2. Related Work

Early techniques for reducing the RAM requirements of embedded systems were mostly hardware-based, i.e., they were implemented with, and relied on, special-purpose hardware. *Code compression* techniques [3,4] 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. *Main memory compression* techniques [5,6] insert a hardware compression/decompression unit between the cache and RAM. These 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 embedded systems.

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 target general-purpose systems with hard disks. *Compressed caching* [7–11] was proposed by a number of researchers to simultaneously handle both code memory compression and data memory compression. It improves system performance by decreasing the number of page faults serviced by hard disks, which have much longer access times than RAM. *Swap Compression* [12–15] compresses swapped pages and stores them in a cache. However, neither technique has been evaluated on embedded systems for which power consumption and performance are critically important.

In summary, despite the existence of memory compression techniques, few have seen use in commercial embedded systems for one or more of the following reasons: (1) they assume off-line compression and thus cannot 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 evaluated, or are unacceptable, for typical disk-less embedded systems.

The work described in this article makes the following main contributions: (1) unlike previous work, CRAMES handles both online data memory compression and in-RAM filesystem compression; (2) it requires no special hardware or system redesign; (3) the compression algorithm and memory allocation method are carefully selected to minimize performance and energy consumption overheads; and (4) CRAMES targets disk-less embedded systems. In summary, it greatly increases the RAM available to embedded systems with minimal performance and energy consumption costs (refer to Section 5).

3. CRAMES Design

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 working data set of one memory-intensive process (or several such processes) increases until it 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, pages within the main memory working area are compressed and moved to the compressed area so that the process may continue running. When a compressed page is later required by a process, the kernel quickly locates that page, decompresses it, and copies it back to the main memory working area, allowing the process to continue executing.

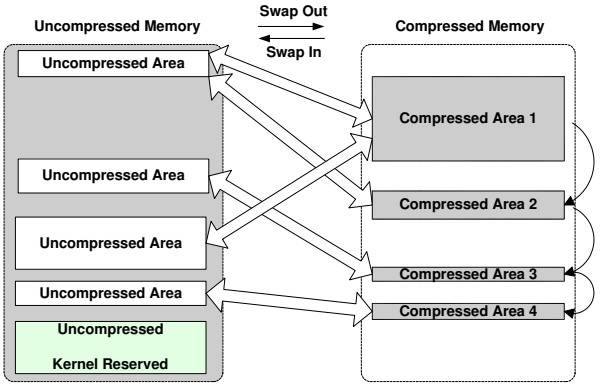


Figure 1: Swapping between uncompressed and compressed RAM

3.1. Design Principles

The goal of CRAMES is to 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.* To guarantee correct operation, pages must be compressed when the address space of active processes exceeds the main memory working area.
2. *Use a performance and energy efficient compression algorithm with low compression ratio*[1] *and memory overhead.* This is crucial to enable CRAMES to increase the amount of usable memory with small performance and energy consumption penalties.
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.
4. *Dynamically adjust the size of compressed area.* The compressed area must be large enough, when necessary, to provide applications with additional memory. However, it should stay out of the way when applications do not require additional memory to avoid having a negative impact on performance and energy consumption.
5. *Minimize the memory overhead.* CRAMES must minimize the memory overhead of compression, fragmentation, and indexing compressed pages.

3.2. Design Overview

This section provides an overview of CRAMES. Three closely related components are briefly introduced: OS virtual memory swapping, block-based data compression, and kernel memory allocation. We then 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, the least recently used data pages are swapped out from memory to, conventionally, hard disks. Swapping allows applications, or sets of applications, to execute even when physical memory 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 uncompressed areas (white) and compressed swap areas (gray), each with non-uniform sizes. Pages are swapped out from uncompressed areas to compressed areas. Note that there is no one-to-one correspondence between a compressed area and an uncompressed area.

The compressed RAM device varies its memory usage over time according to memory requirements. Unlike conventional swap devices, which are typically disk partitions or files, the compressed

[1]Compression Ratio gives a measure of the compression achieved by one compression algorithm on a page of data. It is compressed page size divided by original page size.

Table 1: Memory overhead of 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

RAM device does not have a 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 successfully allocated, the new 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. 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. Note that since a copy of a program's code is kept in its executable file, code pages need not be copied to the swap area or written back to the executable file because they may not be modified. Therefore, swapping is not useful for code compression.

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. 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 [16], and RLE (Run Length Encoding).

Figure 2 illustrate the compression ratios and execution times of the evaluated algorithms and Table 1 gives their memory requirements. For these comparisons, the source file for compression is a dump of pages swapped out from a workstation running SuSE Linux 9.0, which was later divided into uniform-sized blocks to perform block-based compression. The compression ratios decrease with the increase of block size because more similarity is available within a larger block. 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 high enough to starve applications in many embedded systems. RLE offers fast compression and decompression, requires almost no memory except for a few indexing bytes, and has a high compression ratio. 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, low working memory requirements for compression, no memory requirement for decompression [16], high compression speed, and high decompression speed. Therefore, LZO was chosen as the default compression algorithm in CRAMES.

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 coalesced when appropriate. These problems are closely related to the *kernel memory allocation* (KMA) problem. The memory management subsystem maintains mappings from virtual pages to the actual location of data in physical memory, allowing it to satisfy requests for virtually contiguous memory by allocating physically non-contiguous pages. In addition, the kernel maintains a linked list of free pages. Pages are removed from the free list when they are allocated, and returned to the free list when they are released.

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 same swapped data file used to evaluate compression algorithms: resource map allocator (rm), power-of-two freelists (p2fl), McKusick-Karels allocator (mck2), buddy system (bud), and lazy buddy algorithm (lzbud) [17]. As observed for block-based compression algorithms, there is a tradeoff between algorithm quality and performance, i.e., algorithms with excellent memory utilization achieve it at the cost of 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-4KB` stands for resource map allocator with a chunk size of 4 KB. Recall that the CRAMES memory manager requests memory from the kernel in linked chunks in order to dynamically increase and decrease the size of the compressed memory area. Although a 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 was selected as the default allocation method for CRAMES. 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.

3.3. Using CRAMES with the Filesystem

The Sharp Zaurus SL-5600 provides an example of a widely-used portable embedded system. It has 32 MB RAM, only 7.8 MB of which are available for user applications and system background processes. A significant portion (69% or 20 MB) of RAM is used to create a battery-backed RAM disk, i.e., a common RAM device without compression, for the filesystem.

Although the design of compressed filesystems has been studied extensively in recent years, no solution exists for readable/writable RAM disks with arbitrary filesystem type. Cramfs [18] is a read-only compressed filesystem targeting embedded systems. The Linux e2compr [19] patch provides transparent compression and decompression only for the second extended (ext2) filesystem. JFFS2 [20] is a compressed, readable and writable filesystem, but it is for use with flash memory rather than RAM disks. Using JFFS2 on RAM disks would require an intermediate driver and introduce unnecessary performance overhead resulting from flash-specific journaling techniques. Therefore, it is desirable for a memory compression technique to support the compression of RAM disks used for filesystems in addition to the compression of data in main memory. Although this is not its primary goal, CRAMES supports compressed RAM disks containing any type of existing filesystem.

4. CRAMES Implementation

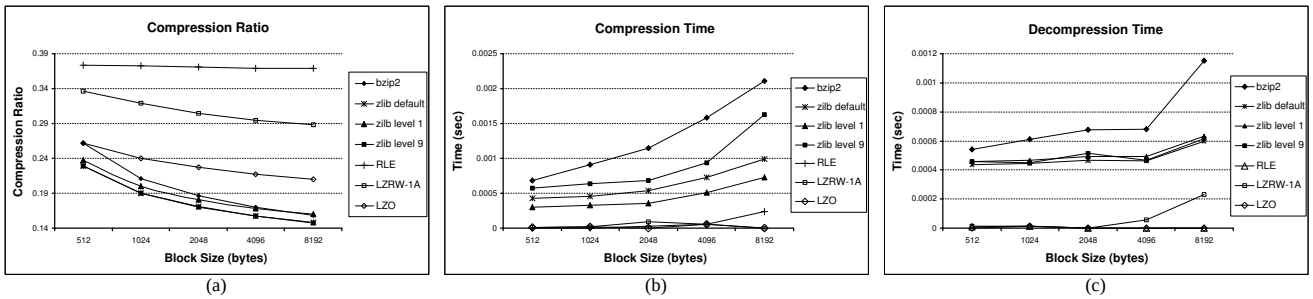
CRAMES has been implemented and evaluated as a loadable module for the Linux 2.4 kernel. The module is a special block device[2] using system RAM. It may serve as both a swap device and a storage area for filesystems. Although the block size for a swap device is 4 KB, i.e., the page size in Linux, the block size of filesystem storage areas may vary. This section describes the structure of a CRAMES device and focuses on its use as a swap device.

4.1. CRAMES Request Handling

CRAMES is a special block device. It must therefore register with the kernel to make itself accessible. During registration, it is necessary to report (1) block size and number of blocks[3], i.e., capacity, and (2) a request handling function, that the kernel calls when there is a read/write request for this device. CRAMES reports an estimated maximum capacity to the kernel, although its actual storage is usually substantially smaller. It enables on-the-fly data compression and decompression via its request handling procedure, which

[2] A block device is a random access device that stores and retrieves data in blocks.

[3] 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.



(a) (b) (c)

Figure 2: (a) Compression ratios, (b) compression times, and (c) decompression times of evaluated algorithms

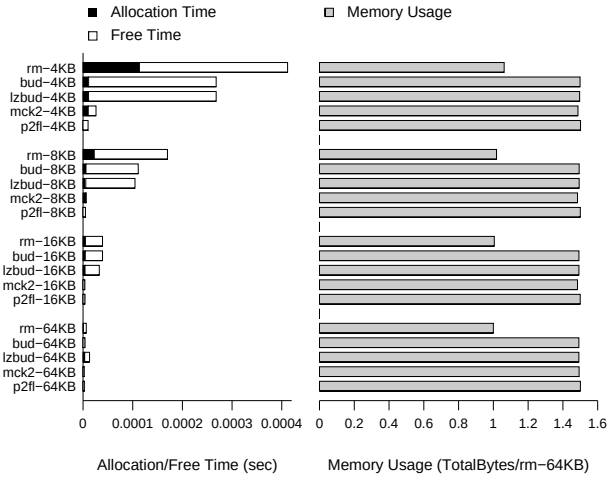


Figure 3: Memory usage of evaluated memory allocation methods

consists of four steps: (1) compressing a block that is written to the device or decompressing a block that is read from the device, (2) allocating memory for a compressed block or locating a compressed block with an index number, (3) managing the mapping table, and (4) merging free slots when possible.

Data in a block device are always requested by their 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 block indices to addresses in virtual memory, CRAMES maintains a *mapping table*, which may be directly-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 enough (at most 16 KB) and fast.

Regardless of the type of mapping table, 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 a block is not compressible 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.

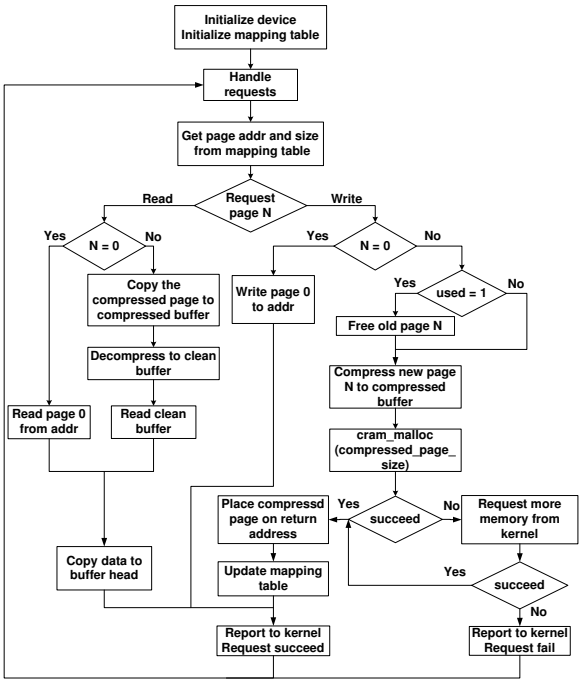


Figure 4: Handling request in CRAMES device

- `addr` records the actual address of a block.
- `size` keeps the compressed size of a block.

Figure 4 illustrates the flow of 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 and 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, request that the CRAMES memory manager allocate a slot of the compressed size for the new page 7, and places the compressed page 7 into the memory region allocated. In the Linux kernel, the first page of a swap device is used to persistently store information about the swap area. Therefore, this page is not compressed by CRAMES and is always placed at the beginning of the device memory space. As a result, in Figure 4, page 0 need not be decompressed or compressed first when it is read or written.

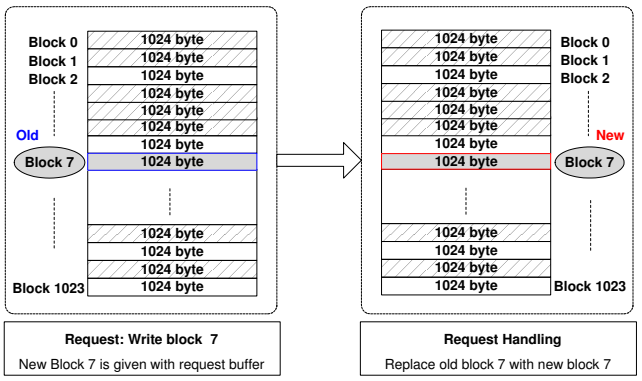


Figure 5: Linear, fixed-size blocks in RAM disk

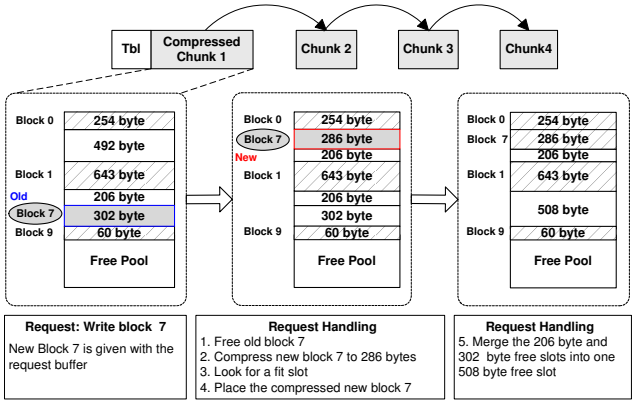


Figure 6: Compressed blocks in CRAMES device

4.2. CRAMES and RAM Disk Comparison

Figure 5 illustrates the logical structure and request handling of a RAM disk[4]. 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 asks from the kernel for a virtually contiguous memory region, which is then divided into uniform fixed-size blocks. When the RAM disk receives a read request for a block, it first locates that 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 the block, then replaces the data in that block with the data in the request buffer.

Figure 6 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 blocks and white areas represent free blocks. 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 need not be divided uniformly because the sizes of compressed blocks may differ due to the dependence of compression ratio on the specific data in each block. When all compressed blocks in a compressed chunk are free, CRAMES frees the entire chunk to the system. This allows the size of a CRAMES device to dynamically increase and decrease during operation, thereby adapting to the data memory requirements of currently running applications. This dynamic adjustment allows CRAMES to support (sets of) applications that would not run without the technique but prevents performance and energy consumption penalties for applications that are capable of running without data compression. When a CRAMES device receives a read request for a block, it looks up the block index in its mapping table, locates the block, 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.

[4] A RAM disk is an in-RAM block device that acts as if it is a hard disk.

Table 2: Timing, power, and energy for filesystem experiments

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

5. CRAMES Evaluation

This section presents energy consumption and performance 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. It has a 400 MHz Intel XScale PXA250 processor, 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. Measurements were taken using a National Instruments 6034E data acquisition board attached to the PCI bus of a host workstation running Linux. 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. This resistor was designed for current sensing applications.

5.1. Using CRAMES for Filesystem on Zaurus PDA

CRAMES was used to create a compressed RAM device for the EXT2 filesystem on a Zaurus SL-5600 PDA. We compared the execution time and energy consumption of this device with that of the EXT2 filesystem on a common RAM disk and observed an average compression ratio of 63% for the CRAMES device. In addition, Table 2 illustrates that the increases in execution time and energy consumption were small: on average 8.4% and 5.2%, respectively.

5.2. Using CRAMES for Swapping on Zaurus

The benchmarks used to evaluate CRAMES contain four applications from the Mediabench benchmark suite [21], one matrix multiplication program with different matrix sizes, ten common GUI applications provided with Qtopia for Zaurus PDAs, and combinations of the above 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 (software) touchscreen device with a named FIFO 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 were tested with and without CRAMES. They 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 Opera and Quasar as well as simultaneously running large matrix multiplication and media player. Table 3 shows that, for the first category, there are seldom any performance, power, or energy penalties be-

Table 3: Timing, power, energy, and compression ratio for swapping experiments

	Application	Description	Size (KB)		Time (s)		Power (W)		Energy (J)		Swap	Comp
			Data	Code	without	w. CRAMES	without	w. CRAMES	without	w. CRAMES	(bytes)	ratio
1	Adpcm	MB: Speech compression	24	4	1.25	1.31	0.38	0.49	0.54	0.79	0	n.a.
2	Mpeg2	MB: Video CODEC	416	48	71.74	71.71	1.16	1.17	82.95	84.10	0	n.a.
3	Jpeg	MB: Image encoding	176	72	0.51	0.49	1.87	2.04	0.95	0.99	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	1004	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	2948	4	52.53	55.06	2.18	2.18	114.69	119.82	129461	0.33
12	Opera Browser	GUI: Web browser	1728	3972	29.65	29.65	1.78	1.69	52.86	50.16	454585	0.40
13	Primtest	GUI: Java Multi-thread	2848	1364	27.77	27.79	2.06	2.11	57.30	58.52	497593	0.39
14	Quasar	GUI: Java Multi-thread	4192	1364	47.16	47.10	2.01	2.03	94.63	95.43	449224	0.43
15	Opera & Quasar	GUI & GUI combination	6104	5336	n.a.	47.12	n.a.	2.09	n.a.	98.68	992561	0.40
16	Matrix (800) & Media Player	Batch & GUI combination	11600	168	n.a.	83.77	n.a.	3.27	n.a.	273.55	832642	0.34

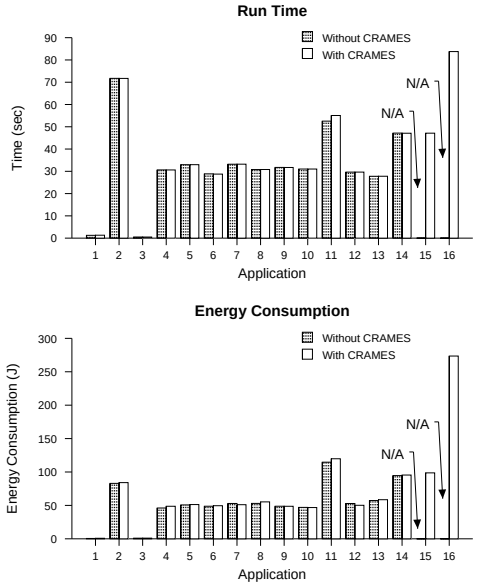


Figure 7: Performance and energy consumption impact of using CRAMES for swapping

cause pages need not be swapped out. For the second category, there are only minor penalties: on average 0.35% for performance, 3.26% for power consumption, and 4.79% for energy consumption. The penalties result from the loss of the small amount of RAM CRAMES reserves for the initial compressed area, which was originally available to the applications. 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 to increase usable memory.

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 and 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 (on average 0.35% and 4.79%, respectively). In addition, CRAMES supports in-RAM compressed filesystems of any 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; it enables the execution of applications for which memory requirements exceed physical RAM. Moreover, it will allow hardware design to be optimized for the typical memory requirements of applications while also supporting (sets of) applications with larger data sets. We plan to publicly release the Linux kernel module implementation of CRAMES for academic and personal use.

7. Acknowledgements

This work was supported in part by NEC Labs America and in part by the NSF in under award CCR-0347941. We would like to thank Peter Dinda for his advice on GUI input monitoring, Hui Ding for assisting us with user input traces, as well as the anonymous reviewers, Peter Dinda, and Lawrence Henschen for their constructive suggestions.

8. References

[1] M. Yokotsuka, "Memory motivates cell-phone growth," *Wireless Systems Design*, Apr. 2004.
[2] "HP online shopping," http://www.shopping.hp.com.
[3] H. Lekatsas, J. Henkel, and W. Wolf, "Code compression for low power embedded system design," in *Proc. Design Automation Conf.*, June 2000, pp. 294–299.
[4] X. H. Xu, C. T. Clarke, and S. R. Jones, "High performance code compression architecture for the embedded ARM/Thumb processor," in *Proc. Conf. Computing Frontiers*, Apr. 2004, pp. 451–456.
[5] B. Tremaine, *et al.*, "IBM memory expansion technology," *IBM Journal of Research and Development*, vol. 45, no. 2, Mar. 2001.
[6] L. Benini, *et al.*, "Hardware-assisted data compression for energy minimization in systems with embedded processors," in *Proc. Design, Automation & Test in Europe Conf.*, Mar. 2002.
[7] F. Douglis, "The compression cache: Using on-line compression to extend physical memory," in *Proc. USENIX Conf.*, Jan. 1993.
[8] M. Russinovich and B. Cogswell, "RAM compression analysis," Feb. 1996, http://ftp.uni-mannheim.de/info/OReilly/windows/-win95.update/model.html.
[9] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis, "The case for compressed caching in virtual memory systems," in *Proc. USENIX Conf.*, 1999, pp. 101–116.
[10] M. Kjelso, M. Gooch, and S. Jones, "Performance evaluation of computer architectures with main memory data compression," in *J. Systems Architecture*, vol. 45, 1999, pp. 571–590.
[11] "Compressed Caching in Linux Virtual Memory," http://linuxcompressed.sourceforge.net.
[12] "RAM Doubler," http://www.lowtek.com/maxram/rd.html.
[13] T. Cortes, Y. Becerra, and R. Cervera, "Swap compression: Resurrecting old ideas," in *Software-Practice and Experience Journal*, no. 30, June 2000, pp. 567–587.
[14] S. Roy, R. Kumar, and M. Prvulovic, "Improving system performance with compressed memory," in *Proc. Parallel & Distributed Processing Symp.*, Apr. 2001.
[15] I. C. Tuduce and T. Gross, "Adaptive main memory compression," in *Proc. USENIX Conf.*, Apr. 2005.
[16] "LZO real-time data compression library," http://www.oberhumer.com/opensource/lzo.
[17] U. Vahalia, *UNIX internals: the new frontiers*. Prentice Hall, 1996.
[18] "Cramfs: Cram a filesystem onto a small ROM," http://sourceforge.net/projects/cramfs.
[19] "Transparent compression for the ext2 filesystem," http://e2compr.sourceforge.net.
[20] D. Woodhouse, "JFFS: The journalling flash file system," in *Ottawa Linux Symp.* RedHat Inc., 2001.
[21] C. Lee, M. Potkonjak, and W. H. M. Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communications systems," http://cares.icsl.ucla.edu/MediaBench.