

High-Performance Operating System Controlled Memory Compression

Lei Yang†

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

Haris Lekatsas‡

Robert P. Dick†

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

ABSTRACT

This article describes a new software-based on-line memory compression algorithm for embedded systems and presents a method of adaptively managing the uncompressed and compressed memory regions during application execution. The primary goal of this work is to save memory in disk-less embedded systems, resulting in greater functionality, smaller size, and lower overall cost, without modifying applications or hardware. In comparison with algorithms that are commonly used in on-line memory compression, our new algorithm has a comparable compression ratio but is twice as fast. The adaptive memory management scheme effectively responds to the predicted needs of applications and prevents on-line memory compression deadlock, permitting reliable and efficient compression for a wide range of applications. We have evaluated our technique on an embedded portable device and have found that the memory available to applications can be increased by 150%, allowing the execution of applications with larger working data sets, or allowing existing applications to run with less physical memory.

Categories and Subject Descriptors: C.3 [Special-Purpose and Application-Based Systems]: Real-time and embedded systems

General Terms: Algorithms, Management, Performance

Keywords: Virtual memory, Compression

1. Introduction

Every year, embedded system designers pack more functionality into less space, while meeting tight performance and power consumption constraints. Software running on some embedded devices, such as cellular phones and PDAs, is becoming increasingly complicated, requiring faster processors and more memory to execute. In this work, we propose a method of increasing functionality without changes to hardware or applications by making better use of physical memory via on-line memory compression.

Our existing memory compression framework, CRAMES [1], takes advantage of an operating system's (OS's) virtual memory infrastructure by storing swapped-out pages in compressed format. It dynamically adjusts the size of the compressed RAM area, protecting applications capable of running without it from performance or energy consumption penalties. This memory compression technique has been implemented as a loadable module for the Linux kernel and evaluated on a battery-powered embedded system. Experimental results indicate that it is capable of doubling the amount of RAM available to applications. When used for this purpose, it has little or no impact on the execution times and energy consumptions of applications capable of running without CRAMES.

Although this version of CRAMES that used existing compression algorithms greatly increased usable memory for a wide range of applications, using it to cut physical memory to 40% increased application execution time by 29%, in the worst case. This conflicted with our goal of dramatically increasing the memory avail-

able to applications without changing applications, without changing hardware, and without significant performance degradation, for all realistic embedded applications. In this article, we describe two techniques to further improve the performance of on-line software-based memory compression.

We present a very fast, high-quality compression algorithm for working data set pages. This algorithm, named pattern-based partial match (PBPM), explores frequent patterns that occur within each word of memory and takes advantage of the similarities among words by keeping a small two-way, hashed, set associative dictionary that is managed with a least-recently used (LRU) replacement policy. PBPM has a compression ratio that is competitive with the best compression algorithms of the Lempel-Ziv family [2] while exhibiting lower run-time and memory overhead.

In addition, we present an adaptive compressed memory management technique that predictively allocates memory for compressed data. Experimental results show that our new pre-allocation method is able to further increase available memory to applications by up to 13% compared to the same system without pre-allocation.

2. Related work

A number of previous approaches incorporated compression into the memory hierarchy for different goals. *Main memory compression* techniques [3] insert a hardware compression/decompression unit between cache and RAM. Data is stored uncompressed in cache, and compressed on-the-fly when transferred to memory. *Code compression* techniques [4] store instructions in compressed format and decompress them during execution. Compression is usually done off-line and can be slow, while decompression is done during execution, usually by special hardware, and must be very fast.

Compressed caching [5,6] introduces a software cache to the virtual memory system that uses part of the memory to store data in compressed format. *Swap Compression* [7,8] compresses swapped pages and stores them in a memory region that acts as a cache between memory and disk. The primary objective of both techniques is to improve system performance by decreasing the number of page faults that must be serviced by hard disks. Both techniques require a backing store, i.e., hard disks, when the compressed cache is filled up. Unlike hardware-based main memory compression techniques, neither compressed caching nor swap compression is able to compress code in memory.

IBM's MXT technology [3] used a hardware parallelized derivative of LZ77 [2]. Kjelso, et al. [9] designed X-Match, a hardware-based dictionary coding algorithm. Wilson [6] proposed WKdm, a software-based dictionary coding algorithm. Rizzo [7] presented a software-based algorithm that compresses in-RAM data by exploiting the high frequency of zero-valued data.

3. Overview of CRAMES

This section briefly introduces the design and implementation of CRAMES to provide readers a context for the techniques presented in this paper. To increase available memory to embedded systems, CRAMES selectively compresses data pages when the working data sets of processes exceed physical RAM capacity. When data in a compressed page is later required by a process, CRAMES locates that page, decompresses it, and copies it back to the main memory working area, allowing the process to continue executing. In order to minimize performance and energy consumption impact, CRAMES takes advantage of the OS virtual memory swapping mechanism to decide which data pages to compress and when to perform compression and decompression. CRAMES requires an MMU. However, no other special-purpose hardware is required.

This work is supported in part by NEC Laboratories America and in part by the NSF under award CNS-0347941.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DAC 2006, July 24–28, 2006, San Francisco, California, USA.

Copyright 2006 ACM 1-59593-381-6/06/0007 ...\$5.00.

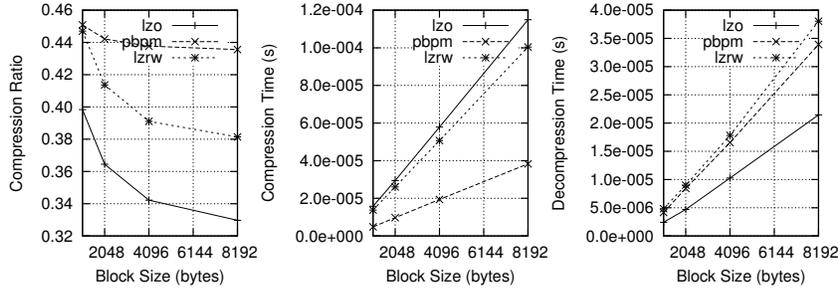


Figure 2: Compression ratios and speeds of PBPM, LZO, and LZRW

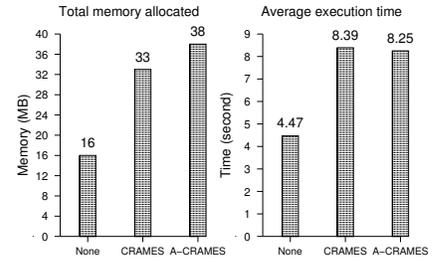


Figure 3: Performance of A-CRAMES

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

```

Require: IN, OUT word stream
Require: TAPE, INDX bit stream
Require: DATA byte stream
1: for word in range of IN do
2:   if word = zzzz then
3:     TAPE ← 00
4:   else if word = zzzx then
5:     TAPE ← 1100
6:     DATA ← B
7:   else if word = zxzx then
8:     TAPE ← 1111
9:     DATA ← BB
10:  else
11:    mnumm ← DICT[hash(word)]
12:    if word = mnumm then
13:      TAPE ← 10
14:      INDX ← bbbb
15:    else if word = mnmxx then
16:      TAPE ← 1110
17:      INDX ← bbbb
18:      DATA ← B
19:      Insert word to DICT
20:    else if word = mnmxx then
21:      TAPE ← 1101
22:      INDX ← bbbb
23:      DATA ← BB
24:      Insert word to DICT
25:    else
26:      TAPE ← 01
27:      DATA ← BBBB
28:      Insert word to DICT
29:    end if
30:  end if
31: end for
32: OUT ← Pack(TAPE, DATA, INDX)

Require: IN, OUT word stream
Require: TAPE, INDX bit stream
Require: DATA byte stream
1: Unpack(OUT)
2: for code in range of TAPE do
3:   if code = 00 then
4:     OUT ← zzzz
5:   else if code = 1100 then
6:     B ← DATA
7:     OUT ← zzzB
8:   else if code = 1111 then
9:     BB ← DATA
10:    OUT ← zBzB
11:   else if code = 10 then
12:    bbbb ← INDX
13:    OUT ← DICT[bbbb]
14:   else if code = 1110 then
15:    bbbb ← INDX
16:    mnumm ← DICT[bbbb]
17:    B ← DATA
18:    OUT ← mnmB
19:    Insert mnmB to DICT
20:   else if code = 1101 then
21:    bbbb ← INDX
22:    mnumm ← DICT[bbbb]
23:    BB ← DATA
24:    OUT ← mnmBB
25:    Insert mnmBB to DICT
26:   else if code = 01 then
27:    BBBB ← DATA
28:    OUT ← BBBB
29:    Insert BBBB to DICT
30:  end if
31: end for

```

5. Adaptive compressed memory management

As described in Section 3, CRAMES compresses the swapped-out data a page (4,096 bytes) at a time and stores them in a special compressed RAM device. Upon initialization, the compressed RAM device only requests a small memory chunk (usually 64 KB) and requests additional memory chunks as system memory requirements grow. Therefore, the size of a compressed RAM device dynamically adjusts during operation, adapting to the data requirements of the currently running applications.

The above dynamic memory allocation strategy works well when the system is not under extreme memory pressure. However, it suffers from a performance penalty when the system is dangerously low on memory: applications and CRAMES compete for remaining physical memory and there is no guarantee that an allocation request will be satisfied. If physical memory is nearly exhausted, an application may be unable to allocate additional memory. If CRAMES were able to allocate even a page of physical memory for its compressed memory region, it would be able to swap out (on average) two pages, allowing the application to proceed. However, CRAMES is in contention with the application, resulting in a scenario we call *on-line memory compression deadlock*.

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

We propose the following scheme to prevent on-line memory compression deadlock. CRAMES monitors the compressed area utilization and requests the allocation of a new memory chunk based on the saturation of current memory chunks. When the total amount of memory in the compressed area is above a predefined *fill ratio*, CRAMES requests a new chunk from kernel. This request may also be denied if the system memory is dangerously low. However, even if this first request is denied, subsequent invocations of CRAMES will generate additional requests. After low-memory conditions cause applications to swap out pages to the compressed RAM device, more memory will be available and the preemptive compressed RAM device allocation requests will finally be successful. We have experimented with different fill ratios and found that 7/8 is sufficient to permit successful requests of compressed RAM device memory allocation in all tested applications. This ratio also results in little memory waste. Evaluation of this method on a portable embedded system is presented in Section 6.2.

6. Evaluation

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

1. Does PBPM provide a comparable compression ratio yet have even lower performance costs than existing algorithms?
2. Does adaptive memory management enable CRAMES to provide more memory to applications when system RAM is tightly constrained?
3. What is the overall performance of CRAMES using PBPM and adaptive memory management?

To evaluate the proposed techniques, we used a Sharp Zaurus SL-5600 PDA. This battery-powered embedded system runs an embedded version of Linux, has a 400 MHz Intel XScale PXA250 processor, 32 MB of flash memory, and 32 MB of RAM. In our current system configuration, 12 MB of RAM are used for uncompressed, battery-backed filesystem storage and 20 MB are available to kernel and user applications.

6.1. Quality and speed of the PBPM algorithm

We evaluated the compression ratio and speed of the PBPM algorithm compared to two other compression algorithms that have been used for on-line memory compression: LZO and LZRW. Figure 2 illustrates the compression ratios (compressed block size divided by original block size) and execution times of evaluated algorithms. For these comparisons, the source file for compression is the swap data file (divided into uniform-sized blocks) used to identify the frequent patterns in memory. The evaluation was performed on a Linux Workstation with a 2.40 GHz Intel Pentium 4 processor. Note that OS-controlled on-line memory compression is a symmetric application, i.e., a memory page is decompressed exactly once every time it is compressed. Therefore, the overall, symmetric, performance of a compression algorithm is the critical performance metric. Overall, PBPM achieves a 200% speedup over LZO and LZRW. Yet the compression ratio achieved by PBPM is comparable with that of LZO and LZRW. We believe that PBPM is especially suitable for on-line memory compression because of its extremely fast symmetric compression and good compression ratio.

Benchmark Description	RAM (MB)	Adpcm			Jpeg			Mpeg2			Matrix Mul.		
		w.o.	LZO	PBPM	w.o.	LZO	PBPM	w.o.	LZO	PBPM	w.o.	LZO	PBPM
Execution Time (s)													
Adpcm: Speech compression Source: MediaBench Data size: 24 KB Code size: 4 KB	8	4.83	1.69	1.43	0.71	0.26	0.23	79.35	80.30	77.96	n.a	39.26	38.68
	9	3.69	1.35	1.26	0.44	0.21	0.21	76.80	76.83	74.04	n.a	37.40	38.24
	10	1.41	1.34	1.36	0.23	0.21	0.21	79.06	76.93	75.32	59.11	39.56	37.18
	11	1.37	1.40	1.40	0.26	0.25	0.21	80.57	76.81	76.83	44.44	38.42	42.65
	12	1.37	1.31	1.32	0.24	0.21	0.19	76.79	76.94	76.95	41.72	38.73	43.96
	20	1.31	1.30	1.30	0.23	0.21	0.22	76.60	76.77	76.76	43.02	41.41	42.97
Power Consumption (W)													
Jpeg: Image encoding Source: MediaBench Data size: 176 KB Code size: 72 KB	8	2.13	2.13	2.13	2.15	2.16	2.15	2.41	2.41	2.51	n.a	2.26	2.29
	9	2.10	2.10	2.13	2.15	2.02	2.07	2.41	2.40	2.50	n.a	2.26	2.29
	10	2.09	2.10	2.09	2.00	1.99	2.04	2.39	2.40	2.48	2.24	2.25	2.29
	11	2.12	2.09	2.13	2.05	2.04	2.07	2.40	2.40	2.50	2.26	2.25	2.29
	12	2.09	2.13	2.11	2.03	2.05	2.10	2.40	2.41	2.55	2.25	2.25	2.29
	20	2.11	2.09	2.18	2.15	2.02	2.24	2.42	2.43	2.57	2.28	2.27	2.29
Energy Consumption (J)													
Mpeg2: Video CODEC Source: MediaBench Data size: 416 KB Code size: 48 KB	8	10.34	3.60	3.04	1.51	0.56	0.49	190.99	193.42	195.71	n.a	88.74	88.62
	9	7.75	2.84	2.68	0.94	0.42	0.43	185.38	184.55	185.10	n.a	84.70	87.64
	10	2.94	2.79	2.85	0.47	0.42	0.42	188.62	184.34	186.42	131.05	88.99	85.01
	11	2.89	2.93	2.97	0.54	0.52	0.44	193.10	184.69	191.94	100.01	86.38	97.79
	12	2.86	2.79	2.79	0.49	0.43	0.41	184.45	185.74	196.33	93.65	86.94	100.81
	20	2.75	2.72	2.82	0.48	0.43	0.49	185.72	186.56	197.26	98.27	94.07	98.39
Matrix Mul.: 512 by 512 matrix multiplication Data size: 2948 KB Code size: 4 KB	8	10.34	3.60	3.04	1.51	0.56	0.49	190.99	193.42	195.71	n.a	88.74	88.62
	9	7.75	2.84	2.68	0.94	0.42	0.43	185.38	184.55	185.10	n.a	84.70	87.64
	10	2.94	2.79	2.85	0.47	0.42	0.42	188.62	184.34	186.42	131.05	88.99	85.01
	11	2.89	2.93	2.97	0.54	0.52	0.44	193.10	184.69	191.94	100.01	86.38	97.79
	12	2.86	2.79	2.79	0.49	0.43	0.41	184.45	185.74	196.33	93.65	86.94	100.81
	20	2.75	2.72	2.82	0.48	0.43	0.49	185.72	186.56	197.26	98.27	94.07	98.39

Figure 4: Overall performance of CRAMES

6.2. Effectiveness of adaptive memory management

In order to determine the effectiveness of adaptive memory management in providing more memory to applications under significant memory pressure, we designed the following experiments. We wrote a ‘memeater’ program that continuously requests 1 MB of memory at a time and fills the allocated memory with random numbers (including zero runs with similar frequency to that observed in real swap traces), until an allocation request fails. Memeater was then executed on Zaurus under three different system settings: without using CRAMES (none), using CRAMES without adaptive memory management (CRAMES), and using CRAMES with adaptive memory management (A-CRAMES). Figure 3 presents the total memory allocated and average execution times under the three system settings (memeater was executed multiple times under each setting). Without CRAMES, the system was only able to provide 16 MB of memory to memeater. With CRAMES, 33 MB of memory were provided and the execution time was proportional to the amount of memory allocated, i.e., no delay was observed. Furthermore, when adaptive memory management is enabled (A-CRAMES), 38 MB of memory were allocated with no additional cost. These results support our claim in Section 5 that A-CRAMES helps to prevent on-line data compression deadlock.

6.3. Overall performance of the improved CRAMES

Embedded system with high-performance on-line memory compression can be designed with less RAM and still support desired applications. In order to evaluate the impact of using CRAMES to reduce physical RAM, we artificially constrained the memory size of a Zaurus with a kernel module that permanently reserves a certain amount of physical memory. The memory allocated to a kernel module cannot be swapped out and therefore is not compressed by CRAMES. This guarantees the fairness of our comparison. With reduced physical RAM, we measured and compared the run times, power consumptions, and energy consumptions of four batch benchmarks, i.e., three applications from MediaBench [11] (Adpcm, Jpeg, Mpeg2) and a 512 by 512 matrix multiplication application. Figure 4 shows execution times, power consumptions, and energy consumptions of benchmarks running without compression, with LZO compression, and with PBPM compression under different memory constraints. Note that adaptive memory management was enabled in both LZO and PBPM compression to ensure fair comparison. In our experiments, each benchmark was executed multiple times; the average results are reported.

As shown in Figure 4, when system RAM was reduced to 8 MB, without CRAMES, all benchmarks suffered from significant performance degradation; the 512 by 512 matrix multiplication couldn’t even execute due to memory constraints. However, with the help of CRAMES, all benchmarks were able to execute with only slight performance and energy consumption penalties. Compared with the base case in which system RAM is 20 MB and CRAMES is not used, PBPM compression results in an average performance penalty of 2.1% and a worst-case performance penalty of 9.2%. This rep-

resents a substantial improvement over LZO, for which the average performance penalty is 9.5% and the worst-case performance penalty can be as high as 29%.

7. Conclusions and acknowledgments

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

We would like to acknowledge Dr. Srimat Chakradhar of NEC Laboratories America for his support and technical advice. We would also like to acknowledge Hui Ding of Northwestern University for his suggestions on efficiently implementing PBPM.

8. References

- [1] L. Yang, *et al.*, “CRAMES: Compressed RAM for embedded systems,” in *Proc. Int. Conf. Hardware/Software Codesign and System Synthesis*, Sept. 2005.
- [2] J. Ziv and A. Lempel, “A universal algorithm for sequential data compression,” *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.
- [3] B. Tremaine, *et al.*, “IBM memory expansion technology,” *IBM Journal of Research and Development*, vol. 45, no. 2, Mar. 2001.
- [4] 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.
- [5] F. Douglass, “The compression cache: Using on-line compression to extend physical memory,” in *Proc. USENIX Conf.*, Jan. 1993.
- [6] 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.
- [7] L. Rizzo, “A very fast algorithm for RAM compression,” *Operating Systems Review*, vol. 31, no. 2, pp. 36–45, Apr. 1997.
- [8] I. C. Tudeuce and T. Gross, “Adaptive main memory compression,” in *Proc. USENIX Conf.*, Apr. 2005.
- [9] 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.
- [10] “LZO real-time data compression library,” <http://www.oberhumer.com/opensource/lzo>.
- [11] 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>.