High-Performance Operating System Controlled Online Memory Compression

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

Yi Wang

INTRODUCTION

Overview

- Design of modern embedded systems:
    - minimize size, cost and power consumption
    - maximize functionality
        - increase flexibility and features
            - increase memory requirements
- Add physical RAM
    - increase size, cost and power consumption
- Make better use of physical memory via memory compression

Memory Compression Motivation and Introduction

- Memory compression techniques comparison:
    - hardware-based:
        - require the design special-purpose compression-decompression hardware
    - software-based:
        - not require dedicated hardware
            - simplify design process
            - reduce time-to-market and design cost
                - more easily apply to existing embedded systems

Memory Compression Motivation and Introduction

- Disadvantage of software-based techniques:
  - high performance and power consumption penalties
  - practical issues like managing migration between compressed and uncompressed portions of memory
- Two techniques presented to further improve performance of online software-based memory compression

Memory Compression Motivation and Introduction

- Pattern-based partial match compression algorithm (PBPM)
  - efficiently compress data pages in RAM
  - twice as fast as best compression algorithms
  - competitive compression ratio
- Adaptive memory management scheme
  - predictively allocate memory for compressed data
  - further increase available memory to applications by up to 13%

Memory Compression Motivation and Introduction

- Demonstration/evaluation:
  - possible to increase available application memory by 2.5x
  - without hardware or application change
  - negligible performance and power consumption penalties

RELATED WORK AND CONTRIBUTIONS

Hardware-Based Memory Compression

- Code compression techniques
  - store instructions in compressed format in ROM
    - offline and slow compression
  - decompress during execution
    - fast, done by special hardware
- Main memory compression
  - insert hardware compression and decompression unit between cache and RAM
  - data stored uncompressed in cache
  - data compressed when transferred to memory

Software-Based Memory Compression

- Compressed caching
  - introduce software-based cache to virtual memory system
  - use part of memory to store data in compressed format
- Swap compression
  - compress swapped pages and store in memory region that act as cache between memory and disk
- Neither designed or evaluated for use in embedded systems
  - use compression algorithms that impose high overheads
  - require hard disk as a backing store

Compression Algorithms for In-RAM Data

- Compression techniques can be lossy or lossless
- Online memory compression requires lossless algorithms
  - many existing not suitable for applications in embedded systems
- Example
  - LZO [Oberhumer]:
    - very fast general-purpose lossless compression algorithm
    - work well on memory data

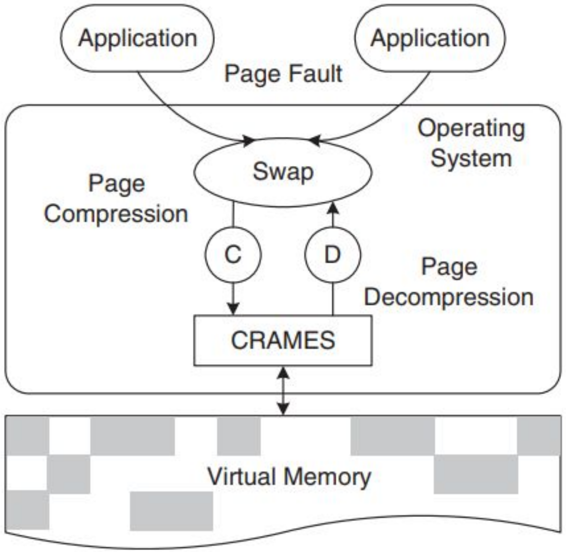
OVERVIEW OF CRAMES

CRAMES design goal

- Memory requirements overran the initial estimate.
- Increase available memory by compression.

CRAMES implementation

- Loadable Linux kernel module.
- Store swapped-out pages in compressed format.
- Compressed memory maintained in a linked list



CRAMES interface

Read operation:

- Locate block using index mapping table, decompress it, and copy to requested buffer.

Write operation:

- Locate block using index mapping table, determines whether to discard old block, and compress the new block

Free:

- Accessed by the only owner of the block.
- Kernel notify the CRAMES to eliminate from compressed memory.

Design Challenges of CRAMES

Which part of memory to compress?

- Selection and scheduling of pages.

How to compress?

- Compression algorithm.

Where to put the compressed memory?

- How much space to allocate for the compressed memory.
- How to manage the allocated space for compressed memory with different sizes.

CRAMES: Which part to compress?

LRU policy for choosing pages to compress.

Frequently used pages in uncompressed area.

Least recently used memory in compressed area.

PATTERN-BASED PARTIAL MATCH COMPRESSION

Background (LZO)

First consideration: LZO algorithm

Advantage:

- significantly faster than many other general-purpose compression algorithms

Background (LZO)

First consideration: LZO algorithm

Disadvantage:

- not designed for memory compression
  - not fully exploit the regularities of in-RAM data
- requires 64KB of working memory for compression
  - significant overhead on embedded systems

Background (PBPM)

- Better result possible for online memory compression
- Extremely fast and well-suited for memory compression
- Observation: frequently encountered data patterns can be encoded with fewer bits to save space.
- Mechanism:
  - Scan through input data by word
  - Exploit frequent patterns within each word
  - Search for complete and partial matches with dictionary entries

Background (PBPM)

- Mechanism:
  - Very frequent patterns:
    - encoded using special bit sequences
    - much shorter than the original data
  - Less frequent patterns:
    - stored in a dictionary
    - encoded using the index of their location in dictionary
  - Least frequent patterns:
    - just stored in dictionary

In-RAM Data Patterns

- Regularities of in-RAM data:
  - pages usually zero-filled after being allocated
  - zeroes commonly encountered during memory compression
- Evaluate relative frequencies of patterns

In-RAM Data Patterns

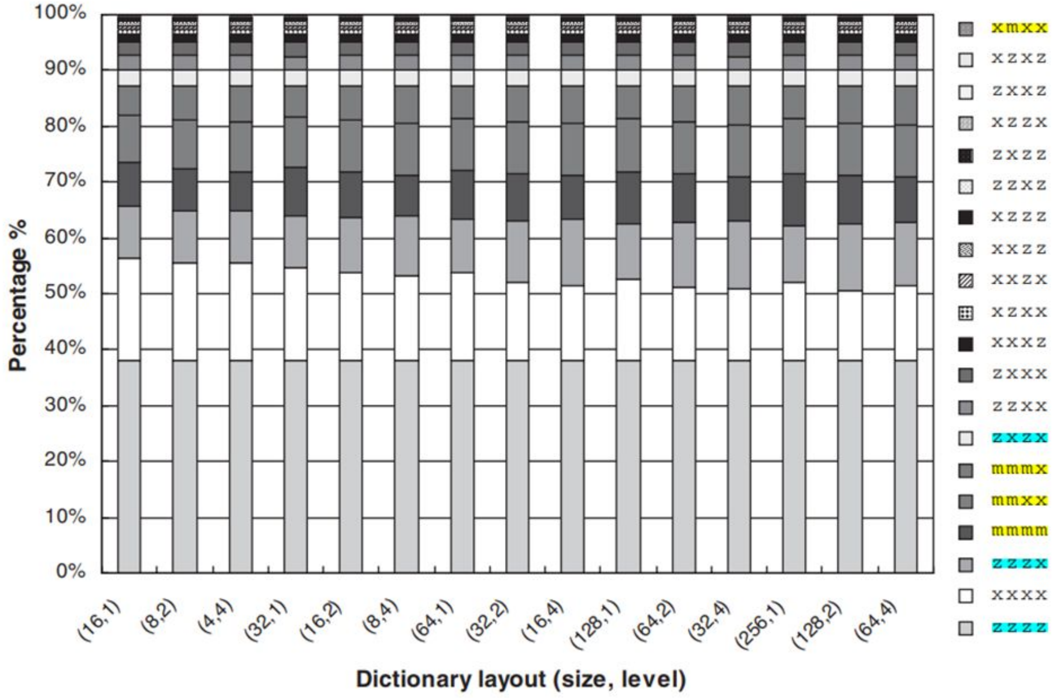


Fig. 2. Frequent pattern histogram.

In-RAM Data Patterns

- Different dictionary sizes and layouts tested
- Most frequent patterns and most effective dictionary layouts for PBPM selected
- A small two-way set-associative dictionary of 16 recently seen words

In-RAM Data Patterns

- Represent each word with four symbols, each representing a byte
- Representation:
  - ‘z’: a zero byte
  - ‘x’: an arbitrary byte
  - ‘m’: a byte that matches a dictionary entry
- Example:
  - “zzzz”: an all-zero word
  - “mmmx”: a partial match with a dictionary entry

In-RAM Data Patterns

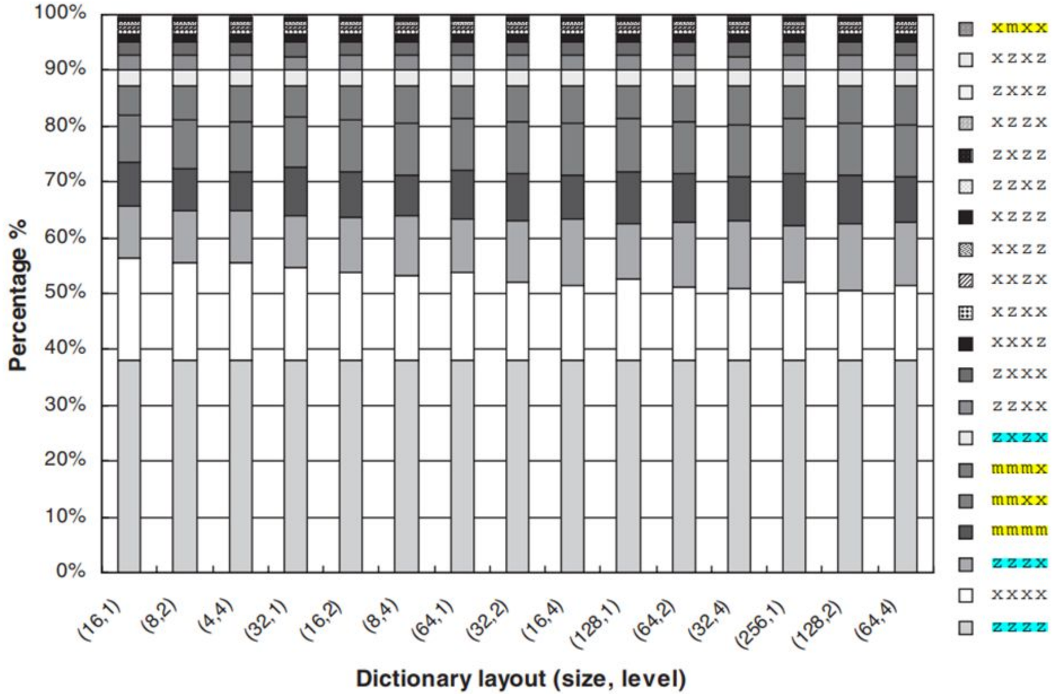


Fig. 2. Frequent pattern histogram.

In-RAM Data Patterns

- Hash-mapped dictionary:
  - allow fast search and update operations
  - the third byte of a word is hash-mapped to a 256 entry table
    - achieve decent hashing quality with low computational overhead
  - contain random indices within the range of the dictionary

In-RAM Data Patterns

Table I. Pattern Encoding in PBPM

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

Only four matched patterns need to be considered:

- “mmmm”
- “mmmx”
- “mmxx”
- “xmxx”

In-RAM Data Patterns

- Possible to consider non-byte-aligned partial matches
- Sufficient to exploit the partial similarities among in-RAM data while permitting efficient implementation (experimental result)

PBPM Compression Algorithm

- Scan through a page (usually 4KB), for each word:
  - first condition met
    - encode with special bit sequence
  - second condition met
    - check whether fully or partially matches a dictionary entry
    - inserted into the dictionary location indicated by hashing on its third byte.

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

PBPM Compression Algorithm

- Scan through a page (usually 4KB), for each word:
  - third condition met:
    - no match at all
    - just inserted into the dictionary
- Set-associative dictionary provides the benefits of both LRU replacement and speed
- Oldest of the dictionary entries sharing one hash target index is replaced

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

PBPM Compression Algorithm

- Neither the hash table nor the dictionary need be stored with the compressed data
- Hash table is static and the dynamic dictionary is regenerated automatically during decompression
- Decompressor
    - read through the compressed output
    - decode the format based on the patterns given
    - add entries to the dictionary upon a partial match or dictionary miss

Software Acceleration

- Store output of all compressed words in one flat array (tape)
- Example:
    - for pattern “mmmm”:
        - two-bit code 01 sent to the output tape
        - four-bit index bbbb sent to the output tape
        - separate tapes used for code, index, and data

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

Software Acceleration

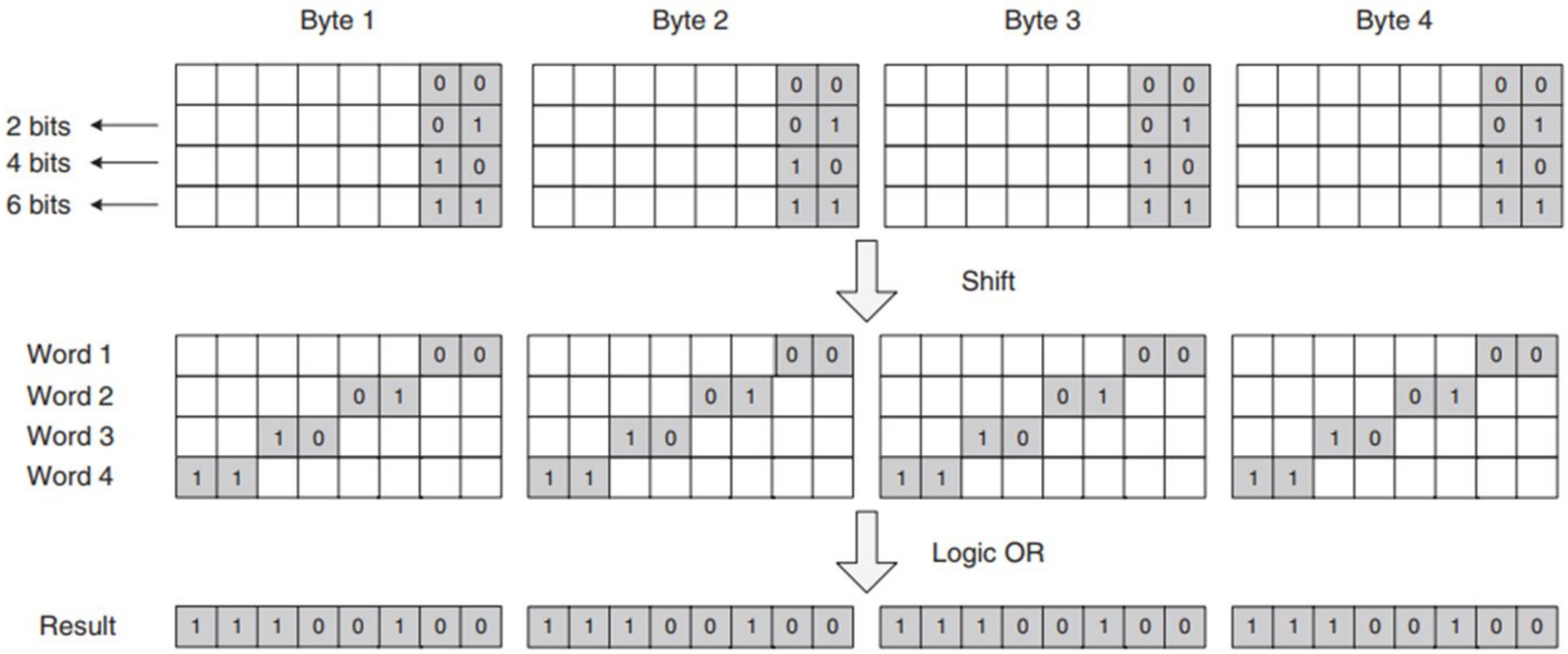
- Code length may be either two-bit or four-bit
- Two tapes for codes, each of which consists of two-bit sequences
- Example:
  - for pattern “mmmx”:
    - first two-bit code 10 sent to tape
    - second two-bit code 11 sent to tape
    - index and data send to tape meanwhile

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

Software Acceleration

- Acceleration technique
  - for code tapes and index tapes
  - allow fast bit operations
- Mechanism:
  - store each two-bit code in the lowest two bits of a byte
  - pack the codes four words at a time after all collected
  - shifting the second word by two bits, the third word by four bits, and the fourth word by six bits
  - perform a logical “or” of these four words

Software Acceleration



Software Acceleration

- Minimizes the total number of shifts required to pack all two-bit sequences
- Four-byte shifts may be carried out in parallel on 32-bit architectures
- Similar technique applied to the index tape, which contains four-bit sequences

Word Alignment

- High performance loads or stores of nonaligned data objects not supported by some processors
- Example:
  - access to a four-byte word with an address that is not evenly divisible by four
    - may be illegal
    - or impose substantial performance penalties

Word Alignment

- Data length may vary for different patterns
- Data tape consists of onebyte, two-byte, and four-byte sequences
- May impose substantial overhead for those processors
- Two alignment schemes implemented to solve the problem

Word Alignment

- Scheme:
  - separate aligned tapes:
    - separate data tapes for one-byte, two-byte and four-byte data
    - copy two to the end of one
    - short tapes -> little copying -> low performance overhead

Word Alignment

- Scheme:
  - single word-aligned tape:
    - maintain only one data tape
    - write data to it in word-aligned manner
    - three pointers maintained to record positions of the next available one-byte, two-byte, and four-byte locations
    - check and update pointer reduce performance
    - no need for coping tapes

Word Alignment

- Compare two alignment techniques and original PBPM (no consideration for the alignment problem)
- Separate vs. single:
    - smaller performance overhead

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

Word Alignment

- Seperate vs. original:
  - 15% compression time increase
  - no effect on decompression time
- Single vs. original:
  - 30% compression time increase
  - 67% decompression time increase

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

Word Alignment

- Conclusion:
  - For architectures that suffer high performance overheads on misaligned accesses:
    - use separate aligned tapes
  - Otherwise:
    - use non-aligned PBPM

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

ADAPTIVE COMPRESSED MEMORY MANAGEMENT

Memory management: How much to allocate for the compressed memory?

- Uncompressed/compressed deadlock
  - No space to compress the uncompressed data
- Predictively request additional memory
  - So that CRAMES can always make space for memory request
- CRAMES request space when compressed area exceed fill rate

Memory management: How to manage the compressed memory space?

- Unlike memory pages, compressed memory are fragmented due to compression.
- Kernel Memory Allocation problem: trade-off between allocation speed and memory usage.
- Resource Map allocator was used for best tradeoff.

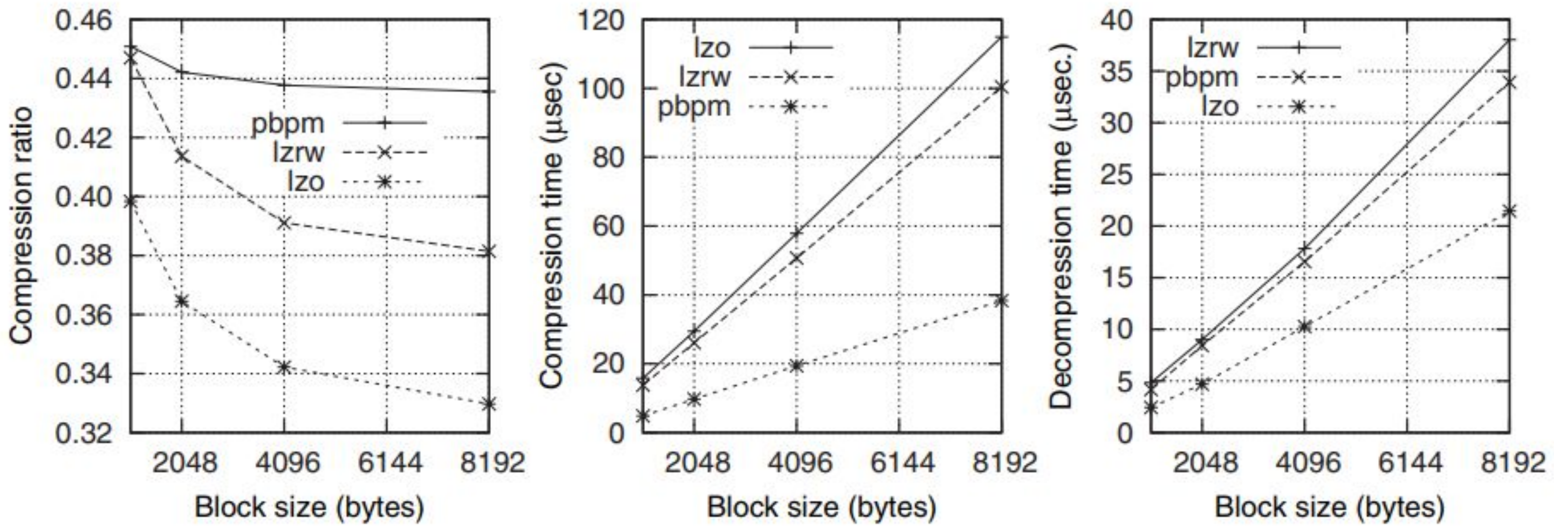
EVALUATION

Overview

- Evaluation methodology and results of the techniques proposed for high-performance online memory compression
- Experimental setup
  - Sharp Zaurus SL-5600 PDA
  - battery-powered embedded system running an embedded version of Linux
  - 400 MHz Intel XScale PXA250 processor, 32 MB of flash memory, and 32 MB of RAM

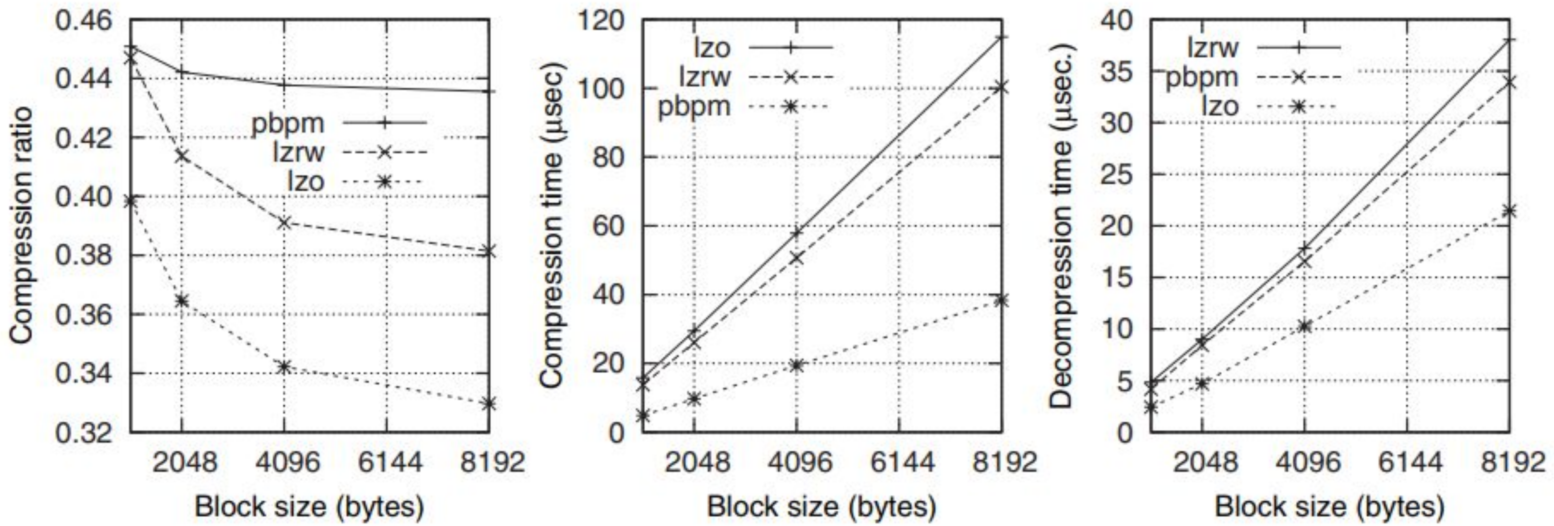
Quality and Speed of the PBPM Algorithm

- Comparison of the compression ratio and speed
- PBPM vs. fastest mode of LZO and LZRW1-A (among the fastest of the available LZRW algorithms)



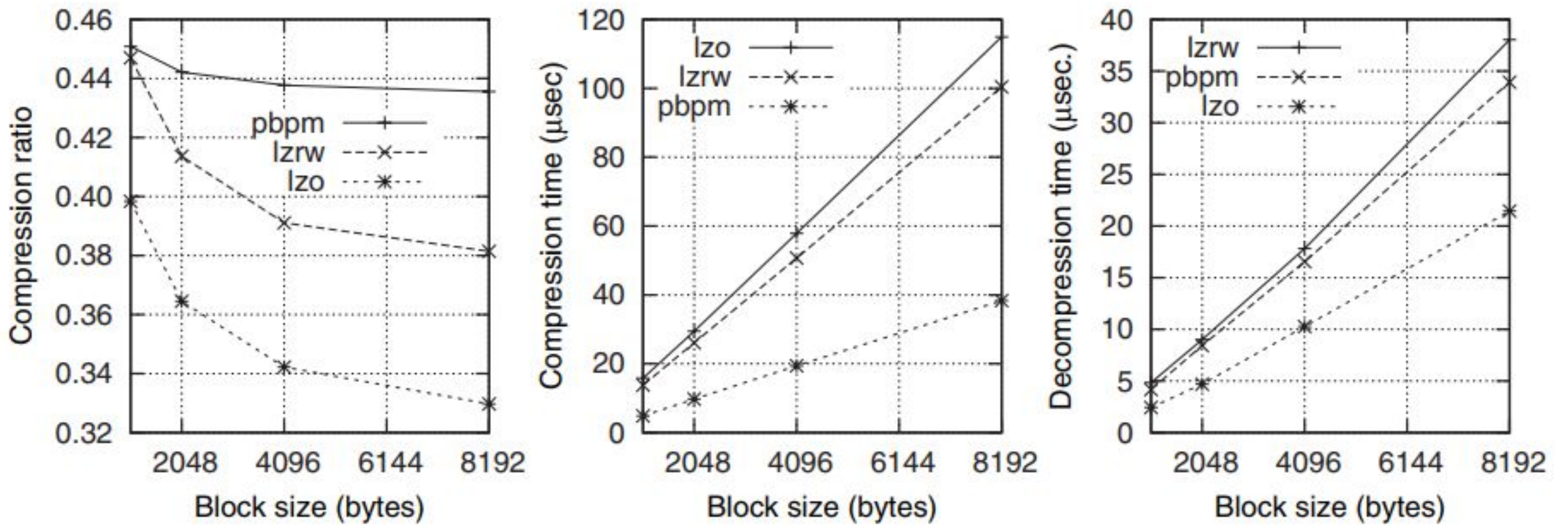
Quality and Speed of the PBPM Algorithm

- Block size = page size (4096 KB) for online memory compression



Quality and Speed of the PBPM Algorithm

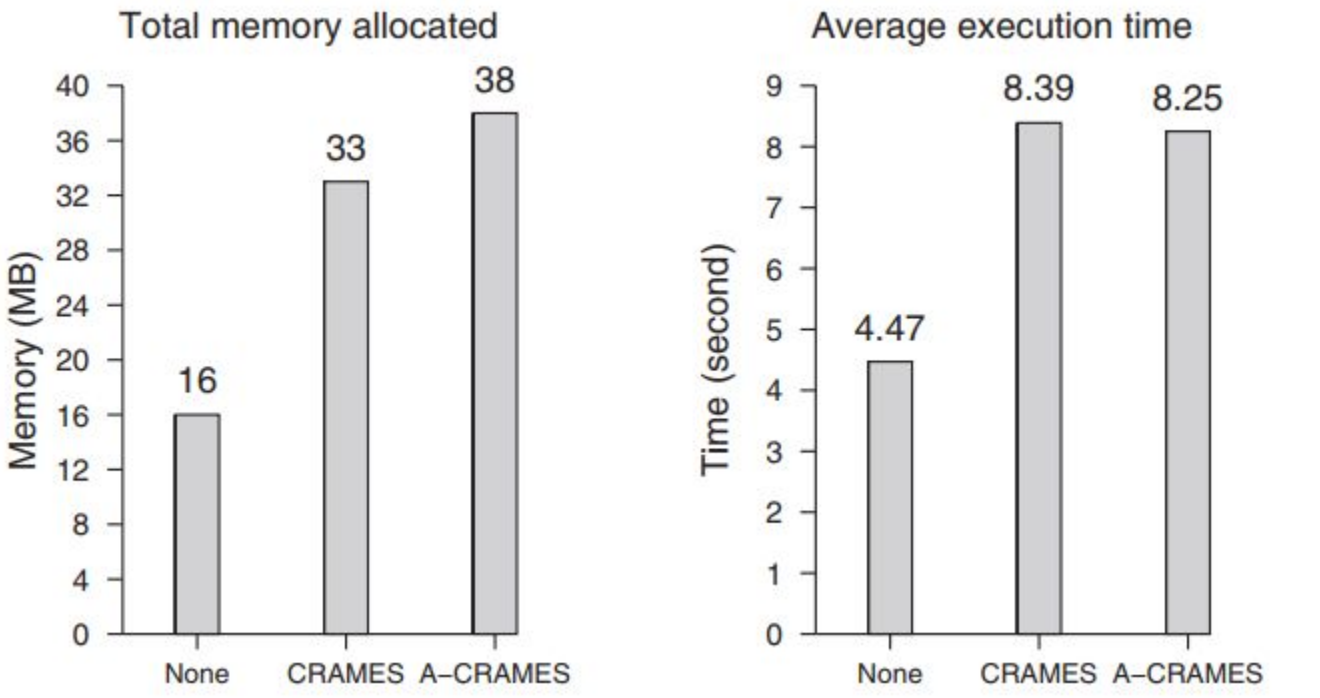
- Result for PBPM:
  - 200% speedup over both
  - competitive compression ratio (44% to 34%/39%)



Effectiveness of Adaptive Memory Management

- Continuously requests memory in 1 MB blocks until a request fails
- Comparison of total memory allocated and execution time
- A-CRAMES (with adaptive memory management enabled) vs. CRAMES vs. without CRAMES

Effectiveness of Adaptive Memory Management



Effectiveness of Adaptive Memory Management

- Result:
  - without CRAMES:
    - 16 MB of memory provided
  - CRAMES:
    - 33 MB of memory provided
    - no delay observed
  - A-CRAMES:
    - 38 MB of memory provided (13% more)
    - without performance penalty
- help to prevent online memory compression deadlock

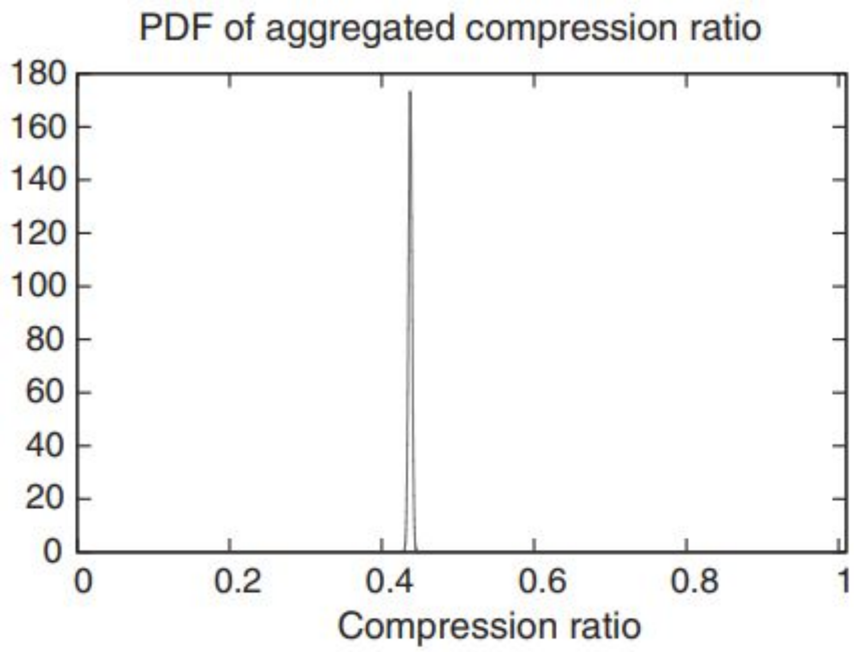
Probability of Running out of Memory

- Overall memory compression ratio influenced by the running applications
- Possible that under some workloads, some applications will write relatively incompressible data to memory
  - prevent CRAMES and PBPM from achieving the predicted aggregate system-wide compression ratio
    - prevent running applications from allocating additional memory

Probability of Running out of Memory

- Probability equivalent to the one of the aggregate compression ratio of in-RAM data exceeding the target compression ratio when deciding the amount of physical RAM in the embedded system
- Approximate the probability of exceeding the target compression ratio (assuming 50%) by using statistical techniques

Probability of Running out of Memory



- The probability of exceeding our target compression ratio of 50% can be estimated as the area under the aggregate PDF
- 3.40 × 10^(−158)

Probability of Running out of Memory

- Conclusion:
  - although no guarantee that a particular set of applications produce pages with an aggregate compression ratio below a particular target compression ratio
  - is unlikely to pose a problem for CRAMES and PBPM

Overall Performance of CRAMES

- With CRAMES, embedded system could:
  - be designed with less RAM
  - still support desired applications
  - with some potential performance and energy consumption overheads
- When under substantial memory pressure
  - PBPM and adaptive memory management minimize these overheads

Overall Performance of CRAMES

- To evaluate the impact of using CRAMES to reduce physical RAM
  - artificially constrained the memory size of with a kernel module
  - permanently reserves a certain amount of physical memory
- Measure and compare the runtimes, power consumptions, and energy consumptions of four batch benchmarks

Overall Performance of CRAMES

- Comparison of performance numbers of benchmarks
- Without compression vs. LZO compression vs. PBPM compression
- Under different memory constraints
- Adaptive memory management enabled for both LZO and PBPM for fair comparison

Overall Performance of CRAMES

Table III. Performance of CRAMES with PBPM and Adaptive Allocation

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

Overall Performance of CRAMES

- Result:
  - both LZO and PBPM impose only small power consumption overheads on the applications
  - performance overheads of both compression algorithms insignificant when system memory not reduced dramatically
  - performance difference between LZO and PBPM becomes obvious when system under tight memory constraints

Overall Performance of CRAMES

- Result (compared to the base case):
  - PBPM:
    - average performance penalty of 0.2%
    - worst-case performance penalty of 9.2%
  - LZO:
    - average performance penalty of 9.5%
    - worst-case performance penalty of 29%

CONCLUSION

Conclusion

- High-performance OS-controlled memory compression can assist embedded system designers to optimize hardware design
- PBPM (efficient compression algorithm for use in OS controlled memory compression)
  - compression ratios competitive with existing algorithms
  - significantly better performance when system memory under tight constraints

Conclusion

- Adaptive compressed memory management scheme
  - prevent online memory compression deadlock
  - further increase the amount of usable memory
- Experimental results:
  - using these two techniques allows applications to execute with only slight penalties
  - even when available RAM reduced to 40% of original size
- No changes to applications or hardware required