Compiler-Guided Data Compression for Reducing Memory Consumption of Embedded Applications

O. Ozturk, G. Chen, and M. Kandemir
Pennsylvania State University

I. Kolcu
University of Manchester

Outline

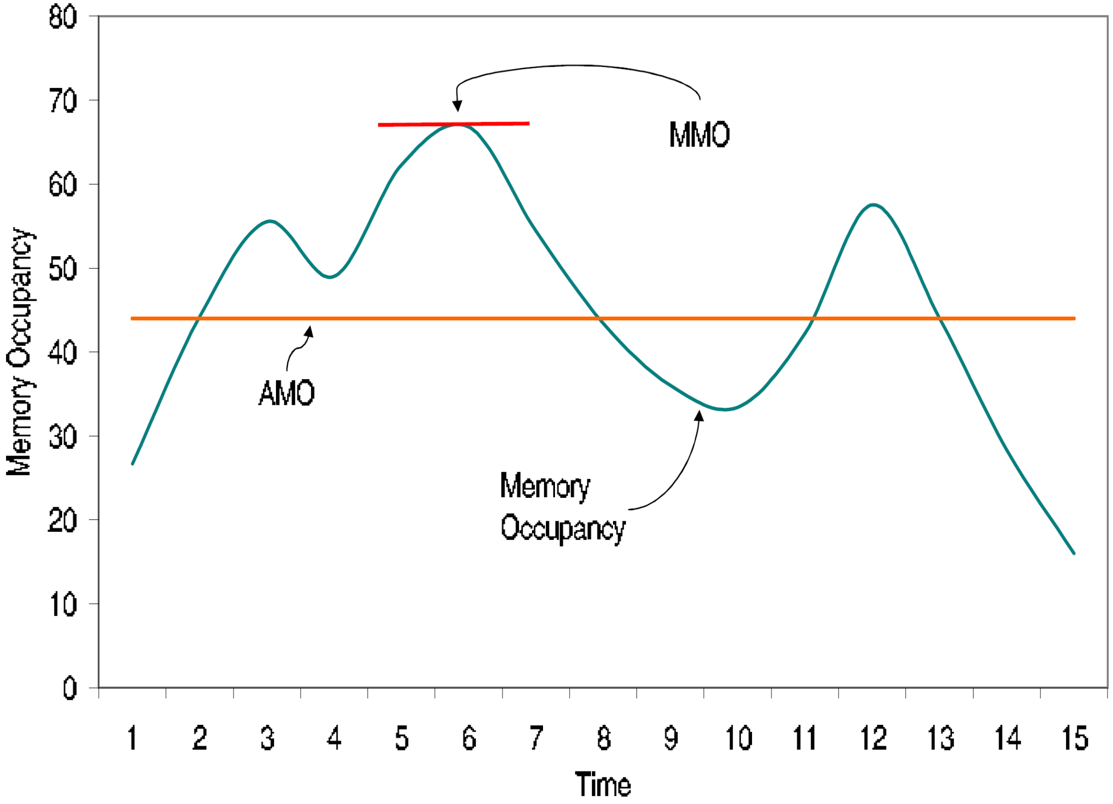
- Introduction and Motivation
- Data Compression
- Data Tiling
- Compiler-Guided Data Compression
- Example
- Experimental Evaluation
- Conclusion

Introduction and Motivation

- A critical component of an embedded computing system is its memory architecture
  - Embedded applications are data intensive and make frequent memory references
    - Execution cycles spent in memory accesses
    - Contribute to a large fraction of overall power consumption
  - Constitute a significant portion of overall chip area
    - Vulnerable to transient errors → reliability optimizations
  - Security leaks are exploited through manipulation of memory space

Memory Occupancy

- MMO: Maximum memory occupancy
  - captures the amount of memory that needs to be allocated for the application
- AMO: Average memory occupancy
  - important in a multi-programming based embedded environment where multiple applications compete for the same memory space
- The drops in the curve
  - Application-level dead memory block recycling
  - System-level garbage collection



Data Compression

- Goal
  - Keep the data block in the on-chip memory even if the reuse distance is large
  - Compress data blocks with large inter-access times
  - When next request comes, decompress the data block and forward it to the requester

Data Compression

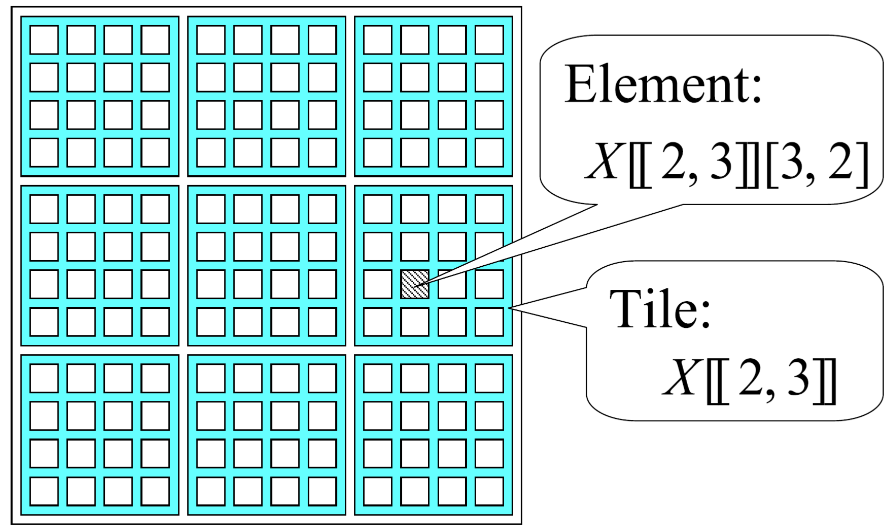
- Advantages
  - Data is kept on-chip
  - Less memory occupation
- Drawback
  - Decompression
  - We should not compress the data block if its reuse distance is short
  - Need a global on-chip memory space optimization scheme

Data Compression

- Challenges :
  - Which data blocks should be compressed and decompressed
  - The order of compressions and decompressions
  - Data sharing across the processors must be accounted for
    - Decisions should be made by global data access patterns
  - Original execution cycle count should not increase excessively
    - Need to be careful about the critical path of execution
    - Complex compression/decompression algorithms should be avoided

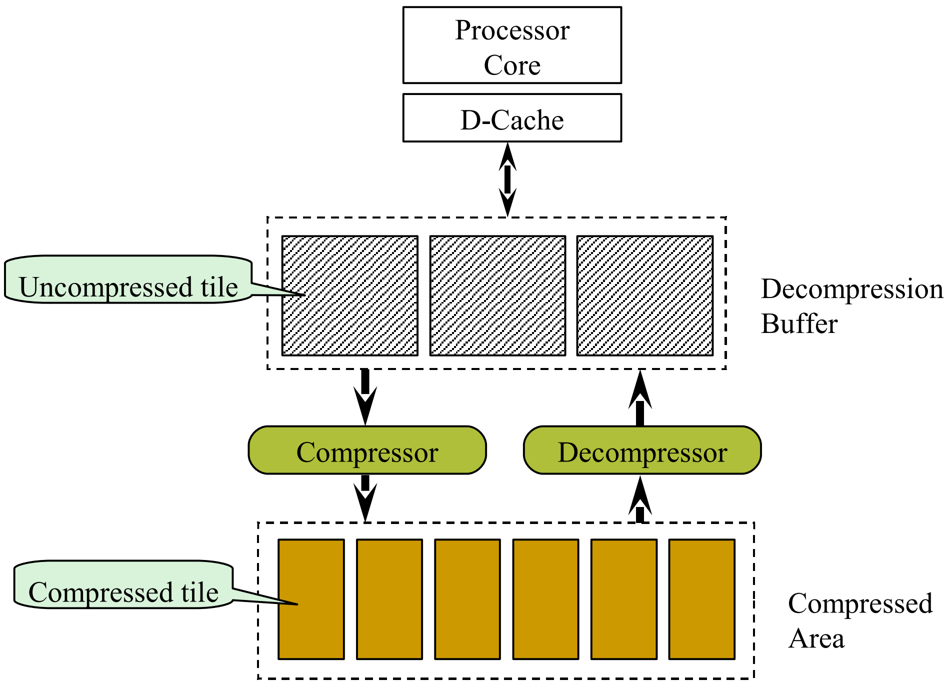
Data Tiles/Blocks

- Arrays are divided into equal-sized tiles/blocks
- In X[[I]][J]
  - I is the tile subscript vector
  - J is the intra-tile subscript vector, which indexes an element within a given tile



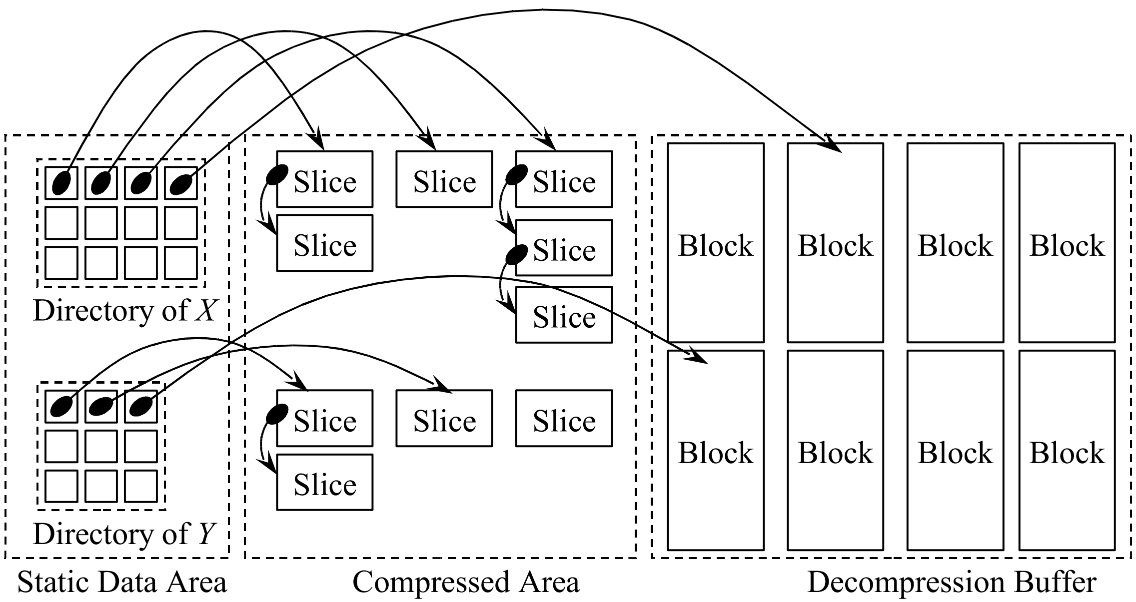
Our Approach

- Compressed area
  - Dynamically managed
  - Uses the compiler-determined schedule
- Decompression buffer
  - Dynamically managed
  - Uses the compiler-determined schedule
- Static data area
  - Scalar variables
  - Statically allocated at compilation time



Our Approach

- Each array has a directory entry
  - Contains a pointer to the memory location
  - In the static data area
- Free table is used to keep track of the free blocks in the decompression buffer
- The compressed area is divided into equal-sized slices
  - A slice is smaller than a block
  - Compression ratio depends on the specific tile
    - Number of slices may vary
  - Slices of the same tile form a link table
- Compiler automatically tiles
  - A user-transparent process



Data Tiling

- Data tiling transforms memory layout
- Example:
  - 600 X 600 array
  - A tile is 100 X 100
- Data tiling:
  - Decompressor invoked 100 times / tile
  - Total decompressions 100 X 36 = 3600

$$\begin{array}{l} \text{for } i = 0 \text{ to } 599 \\ \quad \text{for } j = 0 \text{ to } 599 \; \{ \\ \quad\quad \ldots X[i, j] \ldots \\ \quad \} \end{array}$$

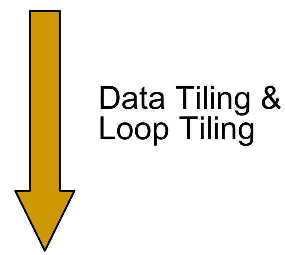
Data Tiling

$$\begin{array}{l} \text{for } i = 0 \text{ to } 599 \\ \quad \text{for } j = 0 \text{ to } 599 \; \{ \\ \quad\quad \ldots X[\![\lfloor i/100 \rfloor, \lfloor j/100 \rfloor]\!][i \mod 100, j \mod 100] \ldots \\ \quad \} \end{array}$$

Data and Loop Tiling

- Loop tiling (iteration space blocking)
  - transforms the order in which the array elements are accessed
  - can significantly reduce the number of decompressions
- Data and loop tiling:
  - Decompressor invoked 1 time / tile
  - Total decompressions 1 X 36 = 36

$$
\begin{array}{l}
\text{for } i = 0 \text{ to } 599 \\
\quad \text{for } j = 0 \text{ to } 599 \ \{ \\
\qquad \ldots X[i, j] \ldots \\
\quad \}
\end{array}
$$



$$
\begin{array}{l}
\text{for } i = 0 \text{ to } 5 \\
\quad \text{for } j = 0 \text{ to } 5 \\
\qquad \text{for } ii = 0 \text{ to } 100 \\
\qquad\quad \text{for } jj = 0 \text{ to } 10 \ \{ \\
\qquad\qquad \ldots X[\![i, j]\!][ii, jj] \ldots \\
\qquad\quad \}
\end{array}
$$

Compiler-Guided Data Compression

- Reduce the number of off-chip memory accesses
  - Even if this increases the number of on-chip communication
  - Optimize on-chip data reuse as much as possible
- If reuse distance is large:
  - Send data to off-chip memory
    - Subsequent access is costly
  - Keep it on-chip
    - Reduces effective memory capacity

Compiler-Guided Data Compression

- Compiler augments the loop nests with the decompression buffer management code
- $d_i(t)$ → The reuse distance of tile
  - Number of intra-tile loop nests executed between the current and the next accesses to the tile
- A compiler-based approach to compute
  - $d_i(t)$ the reuse distance of tile *t* at intra-tile loop nest T_i

Compiler-Guided Data Compression

- If ($d_i(t)$ > n*N) reuse distance is treated as ∞
  - *N* → threshold
  - *n* → number of intra-tile loop nests
- An inaccuracy in computing the reuse distance
  - May lead to performance penalties
  - Not a correctness issue

Compiler-Guided Data Compression

- Decompressions can still incur performance penalties
- Overlap compression and decompression procedures with that of the computing loop nest
- Two threads run in parallel
  - Computing thread
  - Buffer management thread

Example

i	j	c	Nest	Tiles of X in buffer		
1	1	1	L[illegible]	*[1,1]:4	*[1,2]:2	[illegible]
1	1	2	L_2	[1,1]:4	[1,2]:3	*[1,0]:∞
1	2	3	L_1	[1,1]:4	[1,2]:6	*[1,3]:4
1	2	4	L_2	[1,1]:∞	[1,2]:6	[1,3]:5
1	3	5	L_1	*[1,4]:6	[1,2]:6	[1,3]:8
1	3	6	L_2	[1,4]:7	[1,2]:∞	[1,3]:8
2	1	7	L_1	[1,4]:7	*[2,1]:10	*[2,2]:8
2	1	8	L_2	*[2,0]:∞	[2,1]:10	[2,2]:9
2	2	9	L_1	*[2,3]:10	[2,1]:10	[2,2]:12
2	2	10	L_2	[2,3]:11	[2,1]:∞	[2,2]:12
2	3	11	L_1	[2,3]:14	*[2,4]:12	[2,2]:12
2	3	12	L_2	[2,3]:14	[2,4]:13	[2,2]:∞

- Intra-tile loop nests: L1 and L2
- 9 tiles, and buffer can accommodate 3 tiles
- Intra-tile loop nest L1:
  - Tiles → X[[i, j]] and X[[i, j+1]]
  - Reuse distances → d1 = 3 and d2 = 1
- Intra-tile loop nest L2
  - Tiles → X[[i, j+1]] and X[[i, j−1]]
  - Reuse distances → d3 = 1 and d4 = ∞
- [[x, y]] : r → tile [[x, y]] will be reused at c = r
- "*" indicates that the tile is decompressed
- Each tile is decompressed only once

```
for i = 1 to 2
  for j = 1 to 3 {
        c = c + 1;
        load(X[[i, j]], X[[i, j + 1]])
    L1: for ii = 1 to 10
          for jj = 1 to 10 {
            S1: ...X[[i, j]][...]...
            S2: ...X[[i, j + 1]][...]...
          }
        c = c + 1;
        load(X[[i, j + 1]], X[[i, j − 1]]);
    L2: for ii = 1 to 10
          for jj = 1 to 10 {
            S3: ...X[[i, j + 1]][...]...
            S4: ...X[[i, j − 1]][...]...
          }
  }
```

Experimental Evaluation

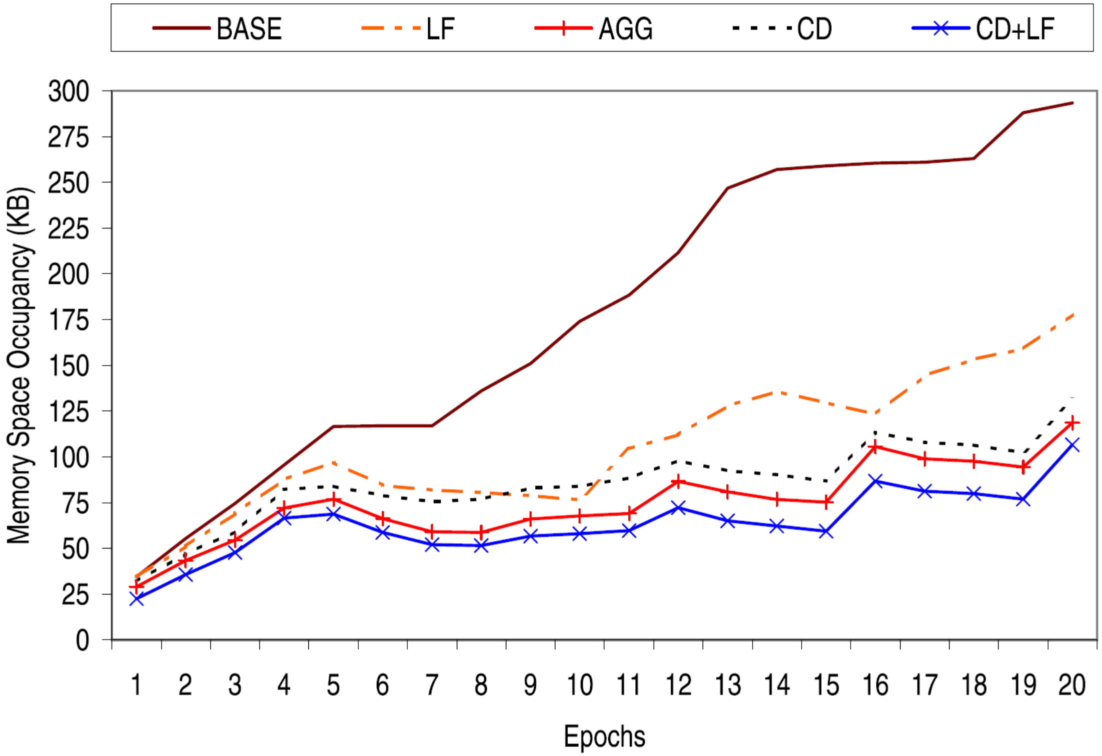
- Implemented using the SUIF
  - Defines a small kernel
  - Implemented as a separate pass
- Used the LZO algorithm as a data compression library
  - Very fast in compression
  - Extremely fast in decompression
  - Thread-safe and lossless
  - Supports overlapping compression and in-place decompression
- Our approach can work with any compression algorithm

Experimental Evaluation

- BASE:
  - Does not employ any data compression or decompression
  - Uses iteration space tiling
- LF:
  - Uses a lifetime analysis at a data block level
  - Reclaims the memory space occupied by dead data blocks
- AGG:
  - Aggressive data compression and decompressions
  - As soon as an access to a data block is completed, it is compressed
  - Reduces memory space consumption significantly
  - Incurs higher performance penalties
- CD:
  - This is the compiler-directed scheme proposed
  - Uses compression and decompression based on the data reuse information
- CD+LF:
  - Combines our compression based approach with dead block recycling
  - Should generate the best memory occupancy savings

Experimental Evaluation

- Memory space occupancy for Jacobi
- 20 epochs with the same number of execution cycles
- Memory occupancy of BASE continuously increases
- The best space savings are achieved with the CD+LF version
  - Combines data compression and dead block recycling



Experimental Evaluation

- Average performance degradation is 3.3\%
- The largest performance loss occur with: mpeg-2 and wave
  - Lowest data reuse

Benchmark	LF		AGG		CD		CD + LF	
	MMO	AMO	MMO	AMO	MMO	AMO	MMO	AMO
Facerec	143.3	118.3	114.7	88.2	126	97.4	98.8	77.1
Jacobi	177.7	105.4	118.7	74.8	131.3	86.0	106.7	63.4
LU	328.0	299.8	253.2	218.8	281.1	236.9	217.8	194.8
Mpeg-2	367.4	281.4	307.6	229.9	340.5	254.1	281.5	205.4
Simi	427.2	352.0	376.8	298.6	399.0	328.3	352.1	277.1
Spec	151.3	114.5	126.3	83.2	143.6	98.1	109.4	71.6
Wave	311.5	245.1	258.1	197.7	303.0	257.8	234.0	180.3
Wibi	585.4	399.2	498.7	320.3	518.7	338.3	466.7	301.9

Conclusion

- A compiler-directed approach
  - Compiler analyzes a given application code
  - Extracts data reuse information at the data block level.
  - Decides the set of data blocks to be compressed/decompressed
  - Decides compression/decompression points
  - Inserts compression and decompression calls in the application code
  - Reduces maximum and average memory space consumption

Thanks