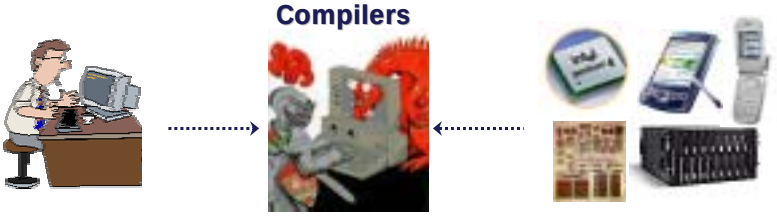
Code Space Optimization for Embedded Systems

August 13, 2004

Computer Science, KAIST
Han, Hwansoo

Roles of compilers

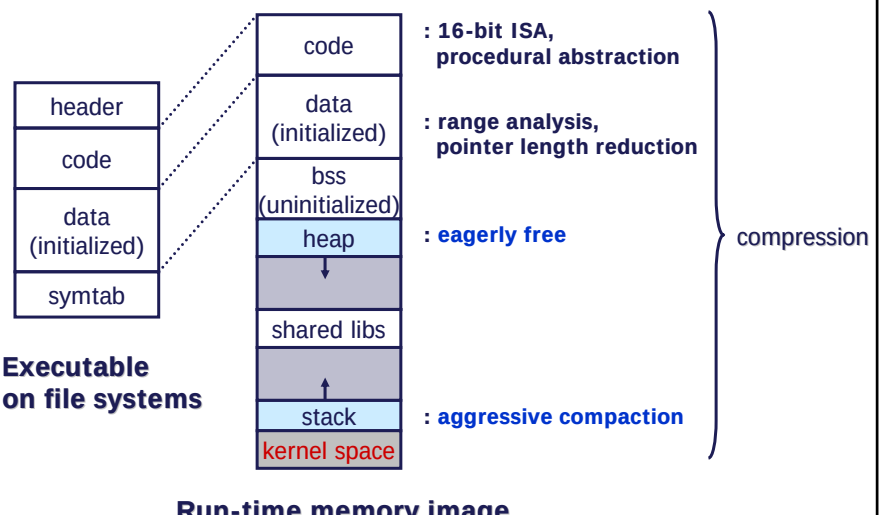
- Compilers help programmers use high-level constructs without performance loss.
- Compilers help applications fully utilize architectural features.



New requirement for embedded systems

- Red-dragon book still in good shape
- Anymore compiler techniques?
  - Backend optimizations are all NP problems
  - Need new heuristics for new systems
- Embedded systems require
  - Low power
  - Small space
  - High level language programming

Compiling for small space



Why small memory images?

- Embedded systems typically have small amount of memory
  - Mobile phones: 16MB SRAM, 16MB flash
  - Digital camcorder phones: 32MB SRAM, 64MB flash
- Multi-program requirement
  - Mobile devices require multi-program (*e.g.* WIPI)
  - Each program needs to reserve memory space to run
  - Data, heap, & stack need to be preserved between context switches (when no MMU support)
- Download over network
  - Bandwidth limitation on wireless network
  - High-latency for download and run scenarios

Static size vs. Dynamic size

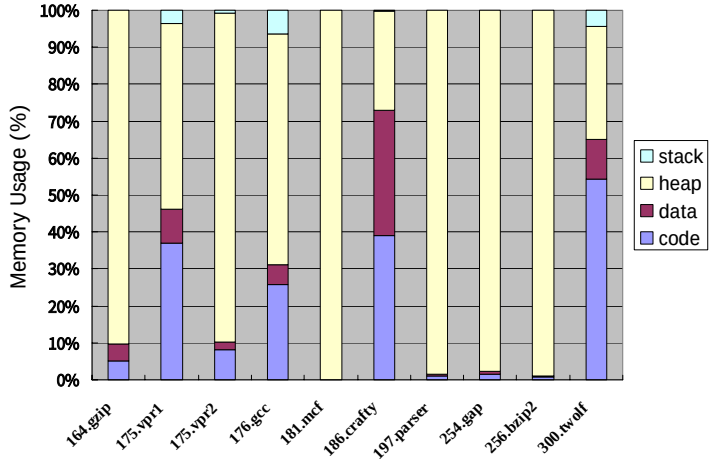
- Static size
  - Reduce downloading time (internet, wireless network)
  - May not impact on real performance
  - Not account for memory/cache footprint
- Dynamic size
  - Likely impact on real performance
  - Efficiently use cache, TLB, memory

Executable file vs. Memory image

- VM-less systems
  - Mobile phones, small controllers, etc.
  - Need to load the whole running image on memory
  - Reducing executable size (.text, .data) affects the size of memory image
- VM supported systems
  - PDA, smartphone, etc.
  - Demand paging (load only what's needed)
  - Need to reduce working set size not the whole image

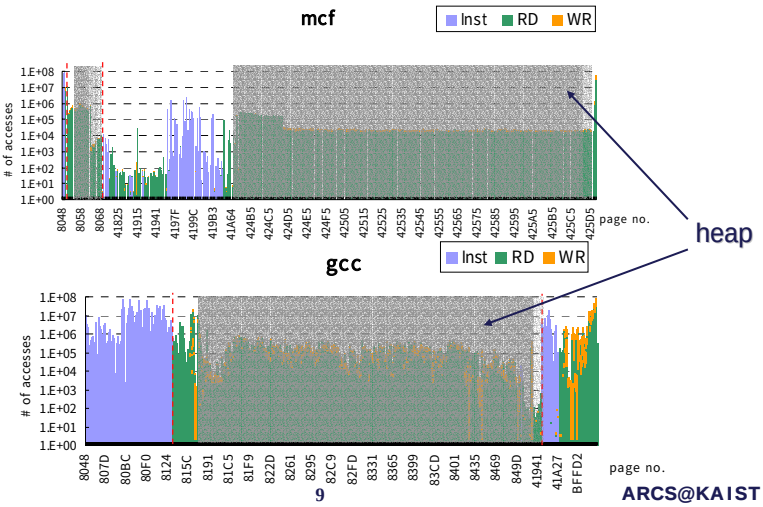
Memory space usage

- Heap touches big chunk of address space



Memory references

- Code section has high reference counts
- References to the same location of heap are small



Code Size Reduction

Recent researches

- [LCTES'02] Krishanswamy & Gupta
  - Thumb ISA (16-bit ISA) instead of ARM ISA (32-bit ISA)
    - 30% size reduction, 20% performance loss
  - Mixed ARM/Thumb ISA
    - 30% size reduction, 5% performance loss

- [PLDI'02] Debray & Evans
  - S/W only approach – 17% code compression of code
    - Compress cold spots, split stream compression
    - 17% static size reduction, small performance loss (4%)

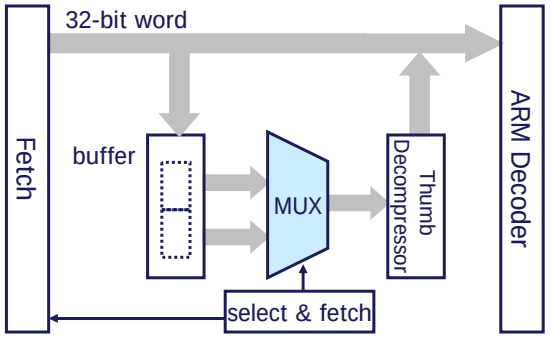
Recent researches

- [PLDI'99] Cooper & McIntosh
  - Procedural abstraction + register rename + relative branch
  - 5% decrease in static #instr, 6~7% increase in dynamic #inst

- [TOPLAS'00] Debray, *et. al.*
  - *squeeze* reduces code size by 28% using *several compiler techniques,* along with 10% speedup
    - redundant code elimination (gp *reg* related) 10%
    - abstracting basic block/region 8%
    - useless/dead code elimination 6%

ARM vs. Thumb instructions

- 32 bit ARM ISA vs. 16 bit Thumb ISA
  - thumb has small code size (30%)
  - thumb can achieve low instruction cache energy (up to 19%)
  - thumb increases instruction count (9% - 41%)



ARM vs. Thumb

- Most Thumb instructions cannot be predicated while ARM supports full predication
- Most Thumb instructions use a 2-address format while ARM supports 3-address format
- Visible registers
  - Thumb mode : *r0* through *r7*
  - ARM mode : *r0* through *r15 (all 16 registers)*
- Branch and Exchange instruction (BX)
  - Switch between ARM and Thumb modes
  - BX Rm Rm[0] = 1 : Thumb mode switc
    Rm[0] = 0 : ARM mode switch

Mixed mode compilation

- Thumb : small code size, low performance
- ARM : large code size, high performance

- Strategy in mixed mode
  - Thumb mode for non-critical parts of codes
  - ARM mode for critical parts of codes

- Granularity
  - Module
  - Function
  - Basic block
  - Sequence of instructions

Thumb vs. Mixed vs. ARM

Benchmark	Code Size		Cycle Count Comparison		Instruction Cache Energy	
	Thumb / ARM	Mixed / ARM	Thumb / ARM	Mixed / ARM	Thumb / ARM	Mixed / ARM
adpcm.rawcaudio	0.694	0.695	1.239	0.999	1.926	0.999
adpcm.rawaudio	0.694	0.695	1.304	0.999	1.212	0.999
g721.encode	0.701	0.719	1.057	1.033	0.920	0.992
g721.decode	0.701	0.719	1.057	1.039	0.907	0.983
jpeg.cjpeg	0.681	0.696	1.051	1.047	0.858	0.856
jpeg.djpeg	0.689	0.711	1.232	1.145	0.819	0.866
mesa.mipmap	0.723	0.772	1.196	1.017	0.941	0.986
mesa.osdemo	0.719	0.766	1.149	1.015	0.926	0.980
mesa.texgen	0.723	0.771	1.075	1.004	0.914	0.982
pegwit.gen	0.681	0.715	0.996	0.988	0.855	1.130
pegwit.encrypt	0.681	0.681	0.998	0.998	0.814	0.815
pegwit.decrypt	0.681	0.681	0.970	0.970	0.855	0.855

Mixed mode Strategies

Select *thumb* if

1. **#inst : no more than 3%**
2. **code size : at least 40% smaller**

Code Compaction

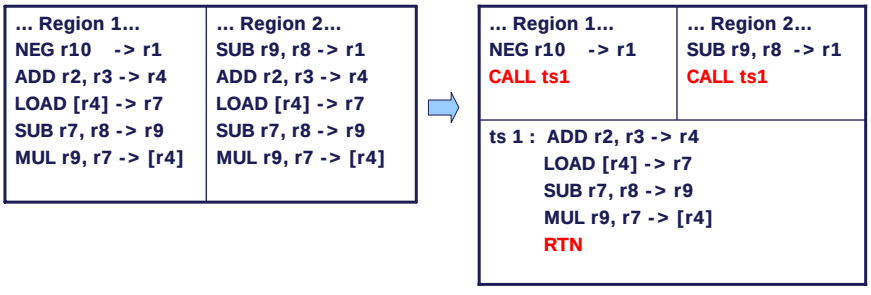
[1] Cooper & McIntosh, *Enhanced Code Compression for Embedded RISC Processors*, PLDI'99

[2] Debray, *et al.*, *Compiler Techniques for Code Compaction*, TOPLAS'00

- Procedural abstraction (or Code factoring)
  - Find repeated code patterns
  - Create a procedure and replace with calls
  - Use cross jumps, if branch back to the same target
- Other techniques combined
  - PC-relative branch targets – expand repeats across BB
  - Register renaming – enhance similarity
  - Predication – handle slightly different code patterns

Procedure abstraction

- Replace repeats with procedure



Cross jumping

- Function calls involve overhead
  - Branch is cheaper, if applicable
  - Jump to the same target at both ends

... Region 1...	... Region 2...
ADD r2, r3 -> r4	MOV r9 -> r3
LOAD [r3] -> r7	LOAD [r3] -> r7
SUB r7, r8 -> r9	SUB r7, r8 -> r9
STORE r9 ->[r4]	STORE r9 ->[r4]
JUMP L3	JUMP L3

... Region 1...	... Region 2...
ADD r2, r3 -> r4	MOV r9 -> r3
JUMP L5	L5: LOAD [r3] -> r7
	SUB r7, r8 -> r9
	STORE r9 ->[r4]
	JUMP L3

Register renaming

- Register renaming to enhance similarity
  - Register recoloring vs. renaming within basic block
  - Take care of live-in registers
    - May require extra moves

... fragment1 ...	... fragment2 ...
ADD r2, r3 -> r4	ADD r2, r3 -> r4
LOAD [r4] -> r7	LOAD [r4] -> r6
SUB r7, r8 -> r9	SUB r6, r8 -> r9
STORE r9 -> [r4]	STORE r9 -> [r4]
ADDI r4, 16 -> r5	ADDI r4, 16 -> r5
LOAD [r5] -> r6	LOAD [r5] -> r7
SUB r6, r4 -> r9	SUB r7, r4 -> r9
STORE r9 -> [r5]	STORE r9 -> [r5]

Register renaming within basic block

- Register Renaming within basic block
  - extra mov's

(R1,R2) live-in	
R0 = R1 + 1 R1 = R0 + R2 R5 = R0 * R1 R3 = R1 - R5 R4 = R5 * 2	R5 = R4 + 1 R3 = R5 + R2 R6 = R5 * R3 R0 = R3 - R6 R4 = R6 * 2
(R3, R4) live-out	

R4 = R1	
R5 = R4 + 1 R3 = R5 + R2 R6 = R5 * R3 R0 = R3 - R6 R4 = R6 * 2	R5 = R4 + 1 R3 = R5 + R2 R6 = R5 * R3 R0 = R3 - R6 R4 = R6 * 2
R3 = R0	

R0 → R5
R1 → R4, R3
R5 → R6
R3 → R0

Experiment results

- Cooper and McIntosh's result [1]
  - overheads increase dynamic instruction count
  - reduce size (static), but increase execution time (dynamic)

Program	% decrease in static instruction count			% increase in dynamic instruction count		
	lexical	rel. bran.	rel. reg.	lexical	rel. bran.	rel. reg.
adpcm	0.00%	0.00%	3.16%	0.00%	0.00%	7.01%
fftn	0.11%	0.11%	0.10%	0.09%	0.09%	0.01%
shorten	0.40%	0.40%	1.57%	0.00%	0.00%	1.81%
gzip	0.28%	0.43%	3.39%	0.00%	0.00%	0.80%
gsm	2.23%	2.23%	14.84%	9.31%	9.31%	13.00%
mpeg2dec	0.35%	0.36%	4.29%	0.02%	0.02%	5.81%
mpeg2enc	0.69%	1.02%	4.08%	0.02%	0.02%	5.81%
jpeg	1.09%	1.08%	6.23%	0.00%	12.99%	19.47%
gs	0.88%	0.89%	5.32%	0.10%	0.19%	4.85%
<mean>	0.67%	0.72%	4.88%	1.07%	2.53%	6.47%

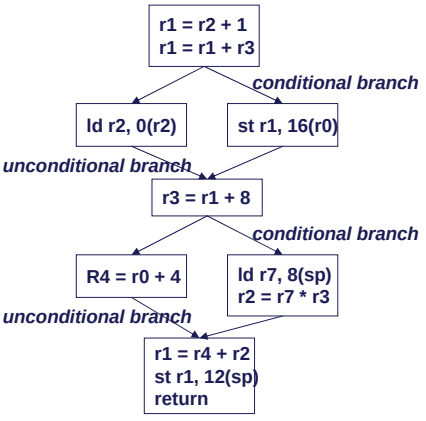
Partially matched block

- Some parts are same, the others are not
- Complex algorithm, computationally high cost
- Involve conditional execution or predication

r1 = r2 + 1 r1 = r1 + 3 ld r2, 0(r2) r3 = r1 + 8 r4 = r0 + 4 r1 = r4 + r2 st r1, 2(sp)	r1 = r2 + 1 r1 = r1 + 3 st r1, 16(r0) r3 = r1 + 8 ld r7, 8(sp) r2 = r7 * r3 r1 = r4 + r2 st r1, 2(sp)

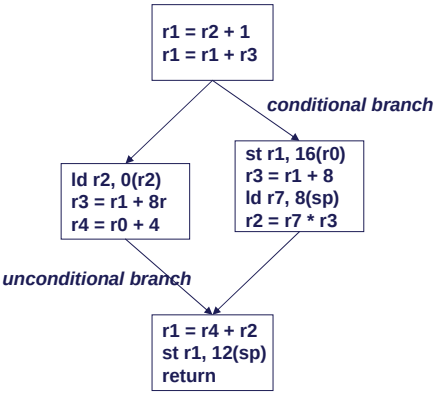
Conditional execution

Maximal matching



total 15 instructions

Unmatching non-profitable



total 14 instructions

Base vs. Squeeze

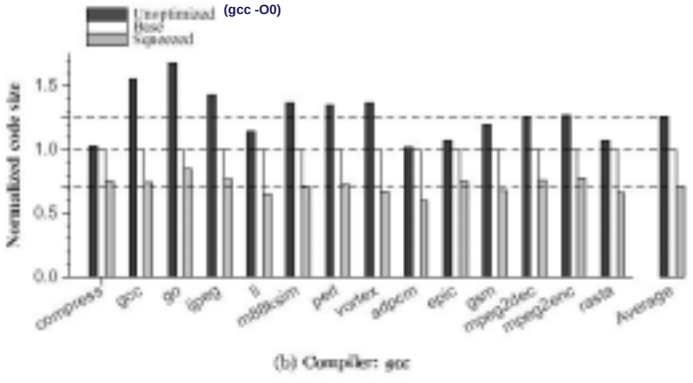
- Base (classic compiler optimizations)
  - Unreachable code elimination
  - No-op elimination

- Squeeze (binary code compression tool for Alpha)
  - Redundant code elimination (gp *reg* related) 10%
  - Basic block/region abstraction 8%
  - Dead (useless) code elimination 6%
  - Register save/restore abstraction 3%
  - Link-time interprocedural optimization 2%

  Total 29%

Experimental results (static code size)

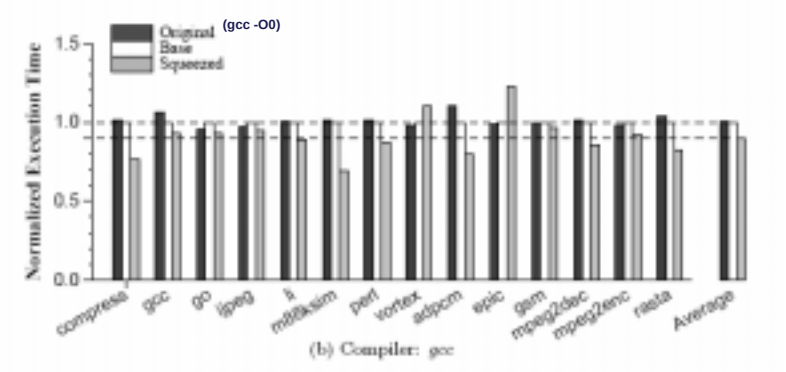
- *Squeeze* reduces 30% of static code size (Debray et al.'s result [2])



(b) Compiler: gcc

Experimental results (execution time)

- *Squeeze* reduces 10% of execution time (Debray et al.'s result [2])



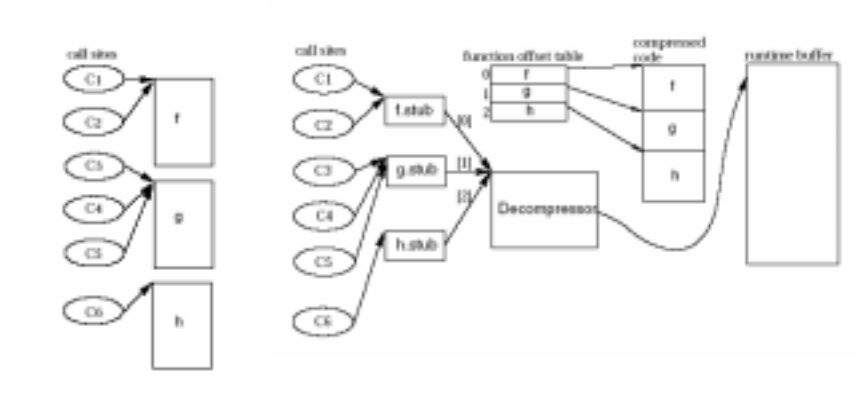
Profile-guided code compression

- [3] S. Debray, W. Evans, *Profile-guided code compression*, PLDI'02

- Design idea
  - 80-20 rule (hot-cold)
  - significant reduction in code size
  - less significant penalty in execution time
  - no H/W support

- Required elements
  - Runtime buffer : execute decompressed code
  - Stub : call compressed function thru decompressor
  - Function offset table :compressed function list

Code organization



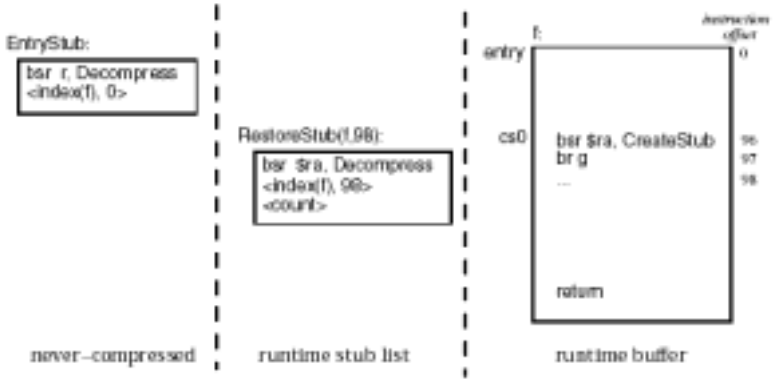
Original code *Compressed code*

Consideration in buffer management

- Never compressed vs. compressed
- Call from uncompressed f() to compressed g()
  - call decompress thru entry-stub
  - manipulate stack for return
  - decompress function in the runtime buffer
  - unconditionally jump to the entry of function
- Call from compressed f() to compressed g()
  - may overwrite runtime buffer
  - when overwrite required, add runtime restore-stub to decompress caller again and branch to the next address of call site
  - manipulate stack to return to restore-stub when g() returns

Restore-stub

- CreateStub inserted when decompress f()
- CreateStub creates RestoreStub

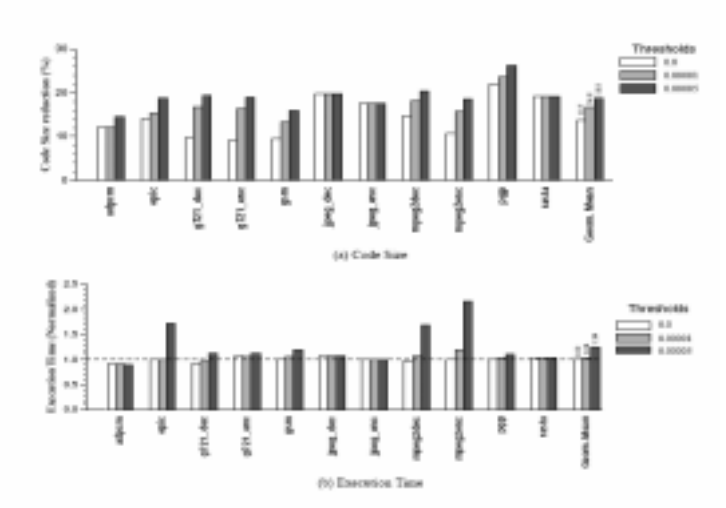


Compression & decompression

- Requirements
  - Good compression even on short instruction sequences
  - The size of decompressor itself
  - Fast decompressor
- Compression algorithm
  - Splitting stream
  - Huffman encoding
  - 66% compression rate

opcode	opd1	opd2	opd3
opcode	opd1	opd2	opd3

Experiment



Cold Region dynamically less than **threshold** instructions among total instructions

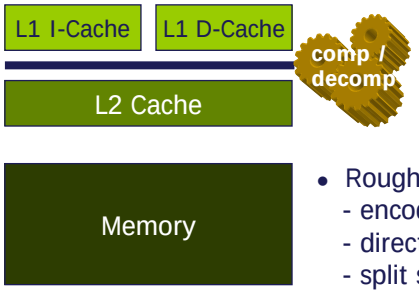
Whole Memory Image Reduction

Compression often reduces half

- Hardware compressor/decompressor
  - placed in-between memory hierarchy
  - compress/decompress all data traffic
  - [TOC'01] Bulent Abali, *et al.*,
    Hardware Compressed Main Memory: OS Support and Performance Evaluation
    IEEE Transaction on Computer Vol.50, No.11, November 2001
- Software compressor/decompressor
  - maintains compressed pool and normal pool
  - tracking/profiling hot data vs. cold data
  - [CA&HPC'03] R. S. Castro, *et al.*,
    Adaptive Compressed Caching : Design and Implementation
    http://linuxcompressed.sourceforge.net

H/W Techniques

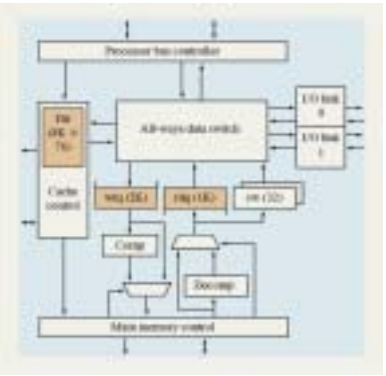
- Insert H/W comp/decomp in Memory hierarchy
  - Small latency than S/W comp/decomp
  - Virtually expand the given H/W space

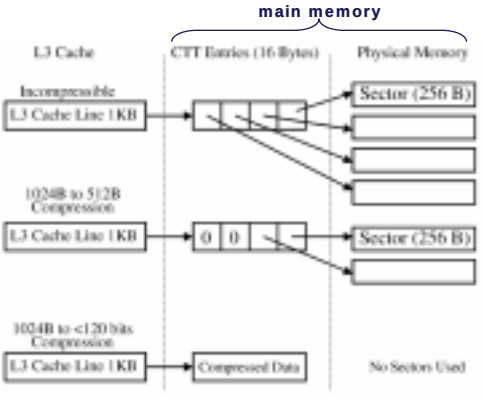


- Roughly 50% compression is achievable
  - encoding
  - directory based
  - split stream

MXT – memory xpansion technolgy

- Compressed main memory
- Hardware comp/decomp: L3 cache - main memory



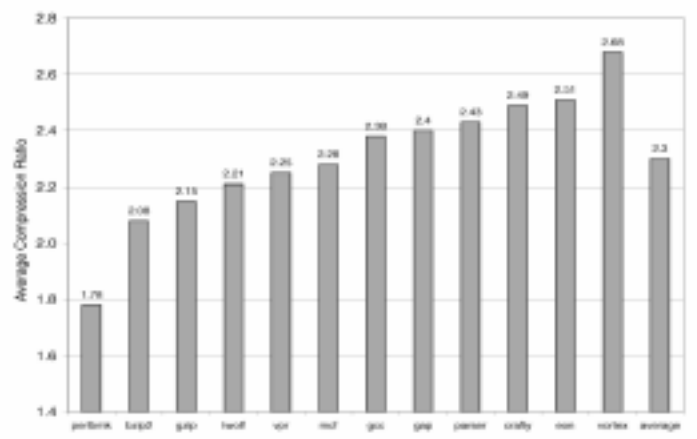


Kernel supports

- Minor changes to make OS VMM compression-aware
- BIOS reports twice larger size of main memory
  - Overcommitted memory requires 50% or better compression rate
  - Monitoring physical page usage
    - Try to keep at least 1MB reservation
    - Limit kernel pages (drivers, buffer cache)
    - Reclaim pages by swapper
    - Trigger CPU blocker to give a time for slow swapper

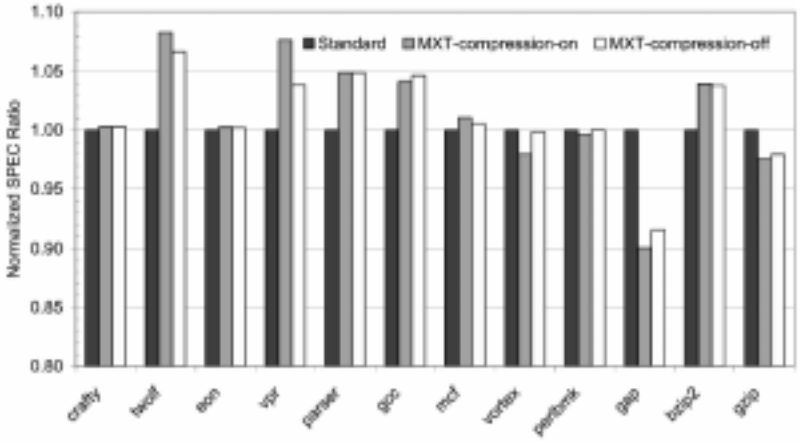
Compressibility – SPEC CINT2000

$$\frac{\textit{Time averaged}\ \text{Real memory utilization}}{\textit{Time averaged}\ \text{Physical memory utilization}}$$



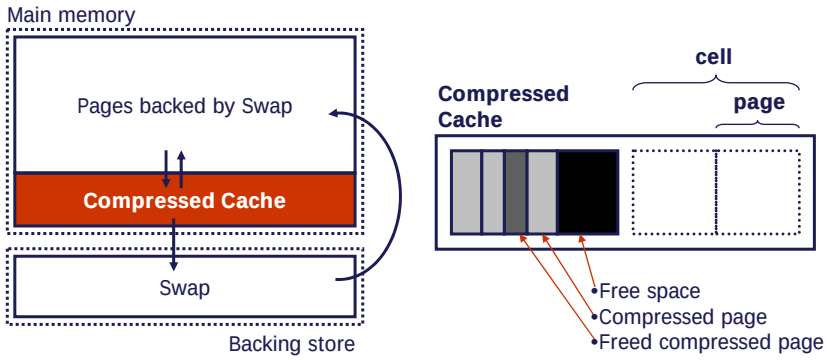
Performance – SPEC CINT2000

512MB DDR physical memory MXT system
Bus: 133MHz, CPU: 733MHz PIII



SPEC CINT2000

Adaptive compressed caching



- Cell (unit of compressed cache)
  - Consists of contiguous pages
  - Contains multiple compressed pages & meta-data

Merits from compressed cache

- Disk access is about 50,000 times slower than main memory (100 cycles vs. 5,000,000 cycles)
- Increase effective memory size (double the size)
- Reduce the number of accesses to backing store
- Considerations
  - Compressed cache size: static vs. adaptive
  - Cell size : 1 page vs. multiple contiguous pages
    - 1 page cell suffers fragmentation for pages with 50% above compression ratio
    - In general, larger cell may suffer more internal fragmentation
  - Page cache (file cache, buffer cache) competes for memory
  - Compressing clean page without later uses (hurt performance)

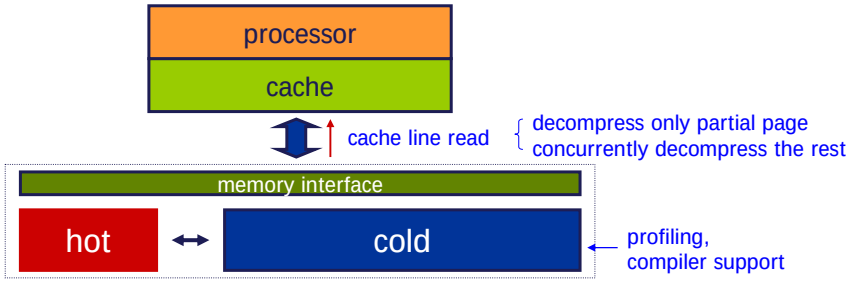
Performance

test	memory Mb	w/o CC seconds	reference gain (%)
kernel (-j1)	18	467.8	21.73
	21	326.68	8.89
	24	289.05	0.20
	27	280.45	0.11
	30	278.33	-0.23
	48	274	0.38
	768	271.17	-0.27
kernel (-j2)	18	1002.62	33.13
	21	608.98	35.84
	24	395.05	18.78
	27	313.8	5.37
	30	283.7	1.12
	48	272.3	0.19
	768	269.76	-0.01
kernel (-j4)	18	1826.14	14.98
	21	1067.47	15.62
	24	826.44	31.85
	27	654.83	34.72
	30	489.67	26.45
	48	274.95	-0.39
	768	271.23	0.28

test	memory Mb	w/o CC req/sec	reference gain (%)
httperf	24	38.5	171.38
	32	117.7	153.40
	36	1529.1	14.10
	40	1849	1.40
	48	1646.1	14.95
	64	1819	3.20
	768	1894.1	-0.25
MUMmer	330	143.5	16.09
	340	115.21	20.74
	360	82.86	26.25
	380	81.21	16.71
	400	80.55	23.02
	420	58.51	15.11
	500	45.35	-0.22
	768	44.7	-0.09
OSDB	24	1242.4	30.70
	48	758.97	-0.07
	768	735.5	0.00
Matlab (1Gb)	768	5880.36	6.12
Matlab (256Mb)	768	1977.83	-0.01
Matlab (80Mb)	768	579.30	-0.03

Working set size reduction

- Requirements
  - One interface for two pools
  - Stay compressed in cold pool
  - Fast read (small decompress latency)



Design considerations

- Exploit page fault mechanism with some H/W support
  - Flags in entries to accommodate partially decompressed pages
  - Any idea on VM-less systems?
  - Background decompression for spatial locality in accesses
- Time to compress
  - Evicted from TLB entries?
  - Fixed of adaptive number of compressed pages?
  - Working set tracking with profile, signature
- Working set size
  - What if it's bigger than physical pages?
  - What if it's bigger than TLB entries?
  - What if it's smaller than TLB entries?

Concluding thoughts

- Space optimization is not only for compilers
  - Need support from H/W, OS
- VM-less systems are hard to deal with
  - Some cell phone makers are adopting MMU
  - Still many embedded systems are VM-less
- We still need to clearly understand what to compress
  - Pages with fewer references?
  - Pages with sparse references?
  - Any other techniques with less impact on performance