Code Compression

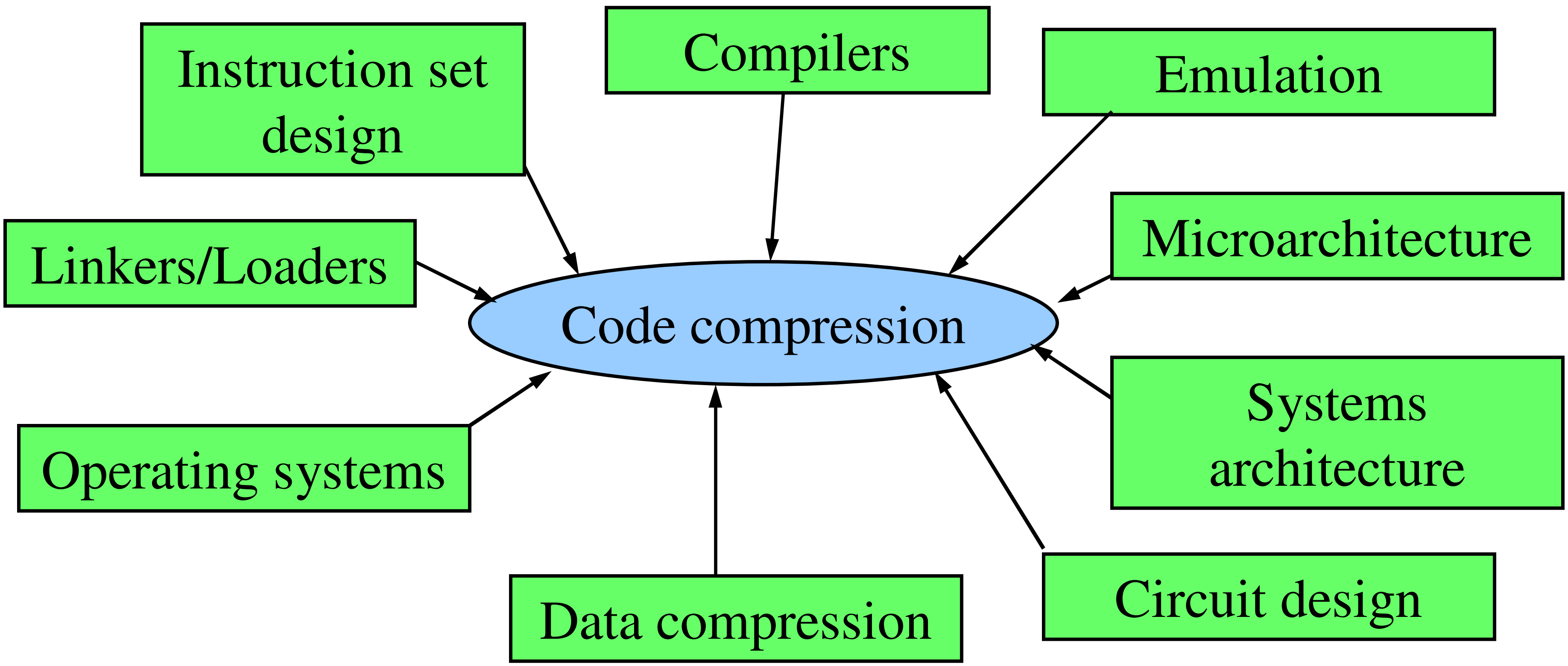
Charles Lefurgy

`http://www.research.ibm.com/people/l/lefurgy`

Austin Research Lab
IBM

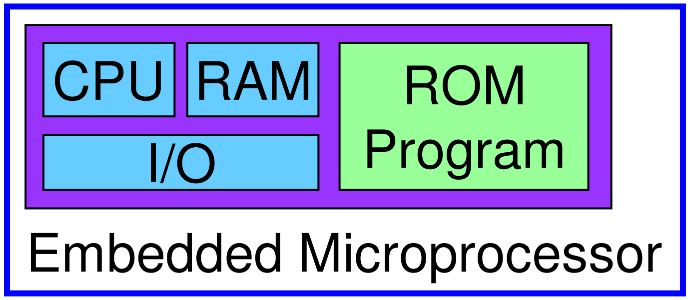
Code Compression

- **Compressing ordinary computer programs <u>and executing the compressed form</u>.**
- **Usually refers only to instruction (not data) memory**

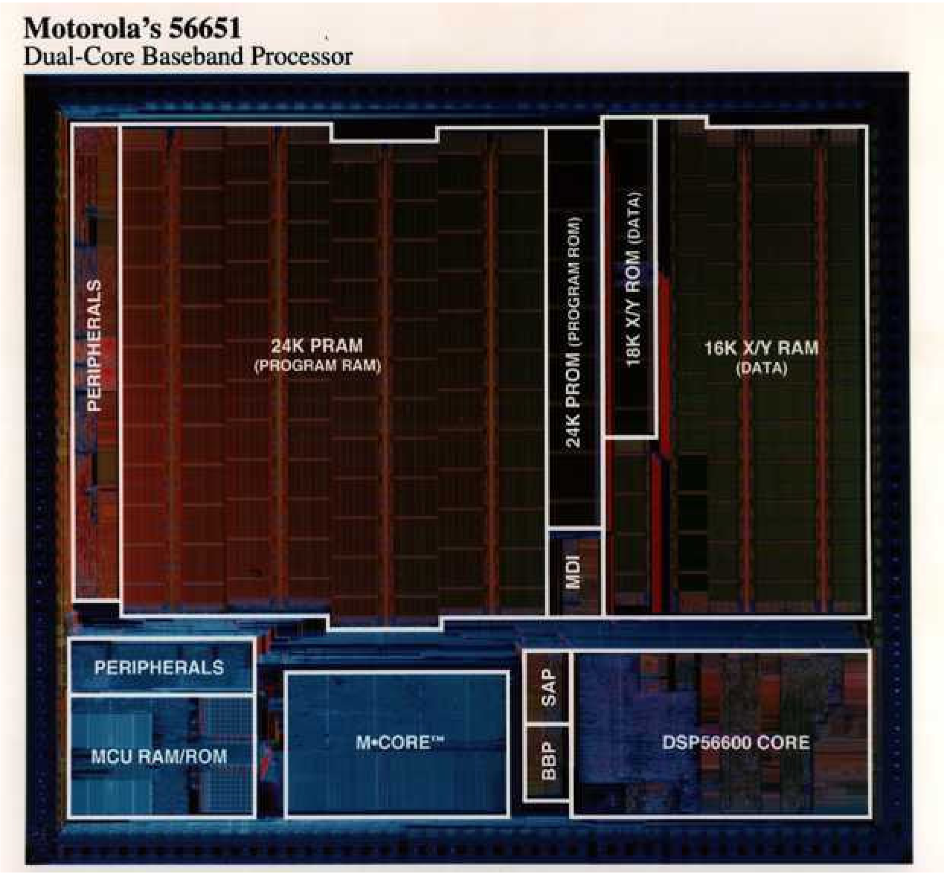


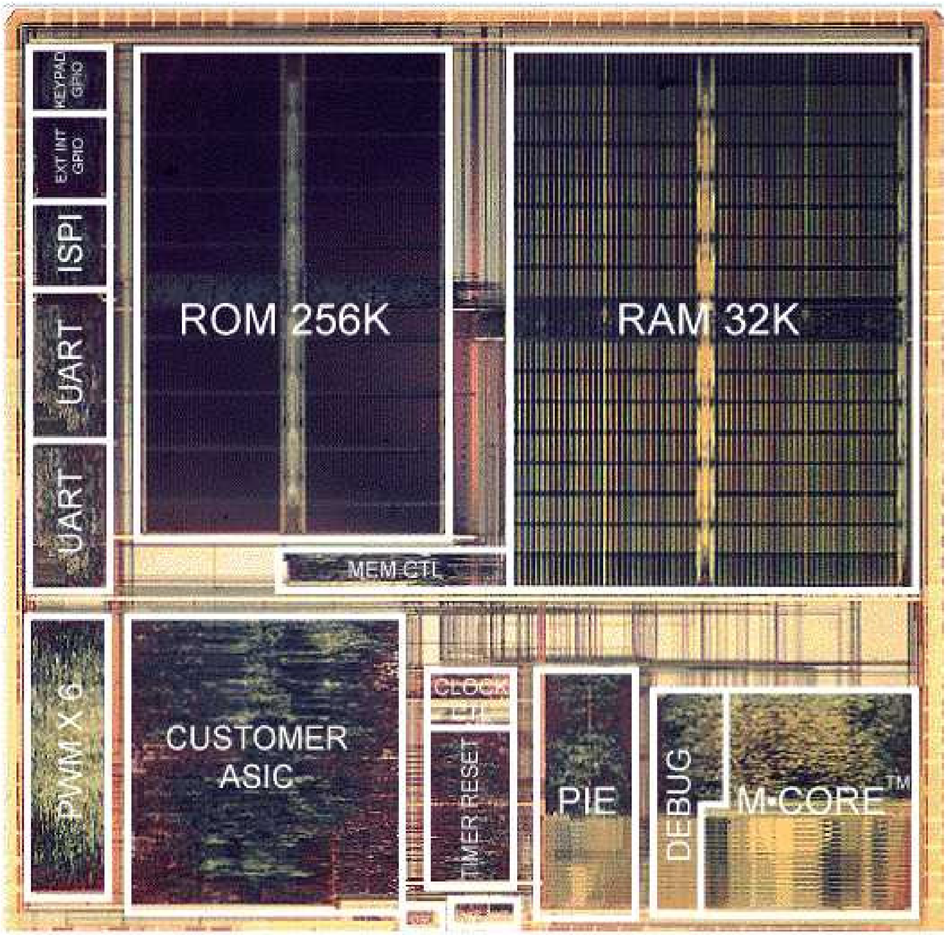
The problem

- **Microprocessor die cost**
  - Low cost is critical for high-volume, low-margin embedded systems
  - Control cost by reducing area and increasing yield
- **Increasing amount of on-chip memory**
  - Memory is 40-80% of die area [ARM, MCore]
  - In control-oriented embedded systems, much of this is program memory
- **How can program memory be reduced?**



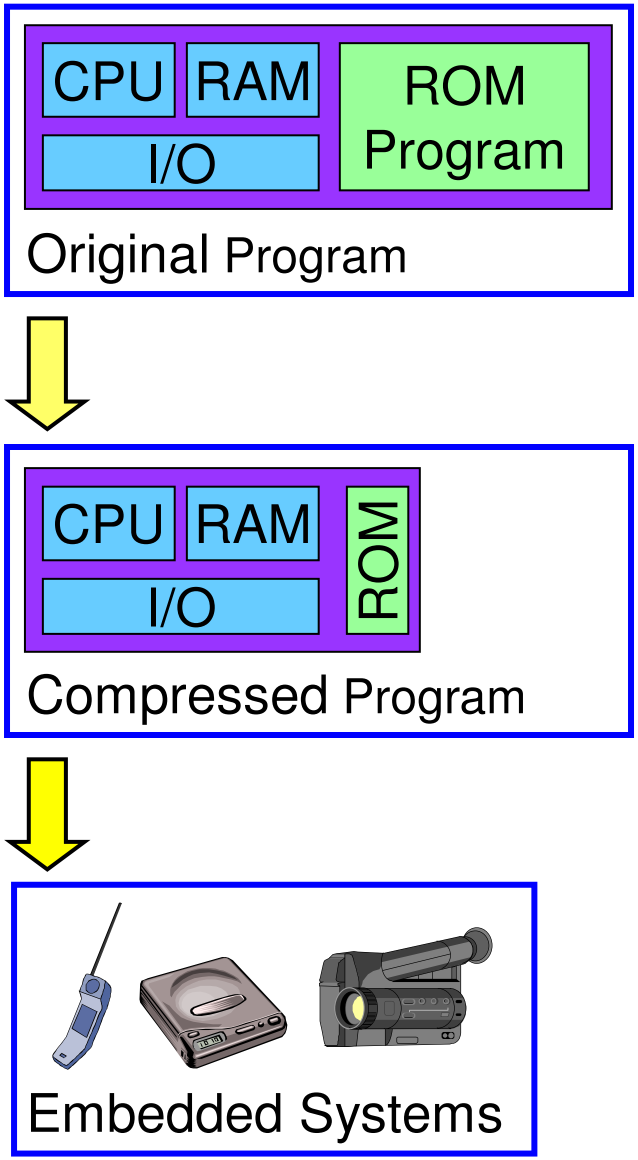
System-on-chip





Solution

- **Code compression**
  - Reduce compiled code size
  - Compress at compile-time
  - Decompress at run-time



Outline

- **Compression methods**
  - Metrics
  - Object code
  - Gzip
  - Static dictionary
  - Adaptive dictionary
  - Stream division
- **Implementations**
  - CCRP
  - CodePack
- **Impact and issues**
  - Performance
  - Energy
  - Compiler optimizations
- **Alternatives to code compression**
  - Instruction set design
  - Compiler optimizations
- **Conclusion**

Compression methods

Metrics
Object code
Gzip

Static dictionary
Adaptive dictionary (LZ)
Stream division

Metrics

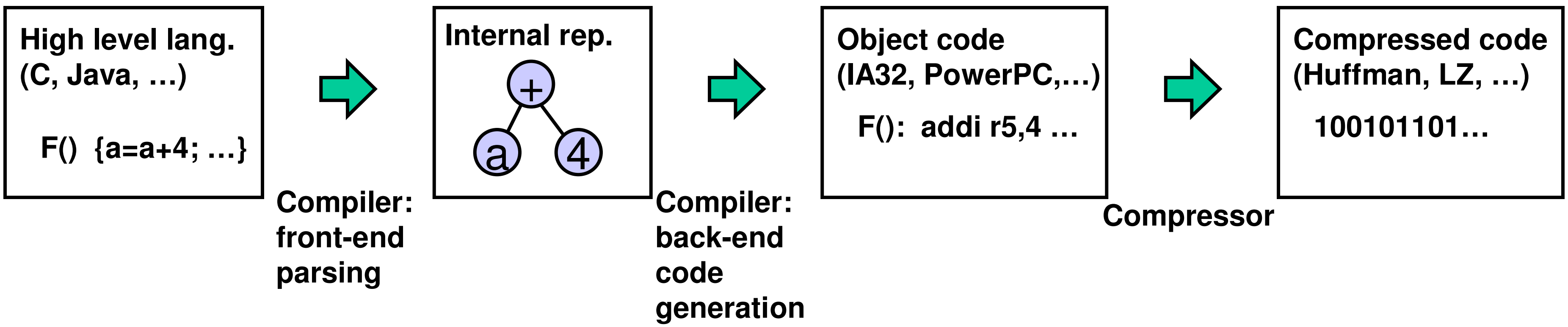
- **Compression ratio**
  - Ranges from 0 to 1
  - 1 is original code size

$$compression\ ratio = \frac{compressed\ size}{original\ size}$$

- **Execution time**
  - Decoding efficiency
- **Energy**
  - Important for battery-operated system
  - Compare to system without code compression
- **Power**
  - Especially for hardware implementations
  - Chip cooling solution is constrained by maximum power dissipated

Code generation

- **Code representations:**
  - High level language
  - Compiler internal format
  - Object code



- **What to compress?**
  - This talk focuses on compressing object code.
  - Compressing the high-level language and compiler formats has been proposed.

Object code

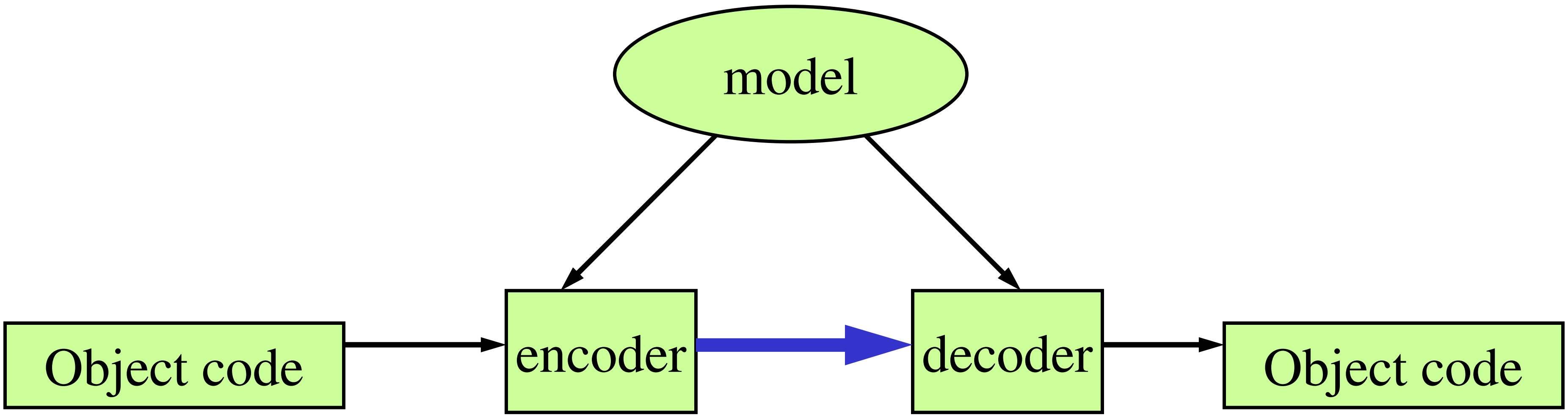
- **Example: PowerPC code from ijpeg benchmark in SPEC95**

```
Offset  Bytes         Assembly code
51ec0   34 e7 ff ff   addic.     r7,r7,-1
51ec4   81 83 00 18   lwz        r12,24(r3)
51ec8   80 63 00 20   lwz        r3,32(r3)
51ecc   4d 80 00 20   bltlr
51ed0   81 04 00 00   lwz        r8,0(r4)
51ed4   38 84 00 04   addi       r4,r4,4
51ed8   39 40 00 00   li r10,0
51edc   54 c9 10 3a   rlwinm     r9,r6,2,0,29
51ee0   7c 8a 60 40   cmplw      cr1,r10,r12
51ee4   81 65 00 00   lwz        r11,0(r5)
51ee8   38 c6 00 01   addi       r6,r6,1
51eec   7d 29 58 2e   lwzx       r9,r9,r11
51ef0   40 84 00 1c   bge        cr1,00051f0c <grayscale_convert+4c>
51ef4   88 08 00 00   lbz        r0,0(r8)
51ef8   7c 09 51 ae   stbx       r0,r9,r10
51efc   39 4a 00 01   addi       r10,r10,1
51f00   7c 8a 60 40   cmplw      cr1,r10,r12
51f04   7d 08 1a 14   add        r8,r8,r3
51f08   41 84 ff ec   blt        cr1,00051ef4 <grayscale_convert+34>
51f0c   34 e7 ff ff   addic.     r7,r7,-1
51f10   40 80 ff c0   bge        00051ed0 <grayscale_convert+10>
51f14   4e 80 00 20   blr
```

Data to be compressed.

Data compression

- **Model**
  - What are the symbols in the input? (instructions, fields, bytes, etc.)
  - What are their frequencies? (Fixed or varying?)
- **Encoder/Decoder**
  - How to encode a single symbol?
    - Most common symbols have the shortest codes
  - Example: Huffman



Transmit compressed
object code

Why not just use gzip?

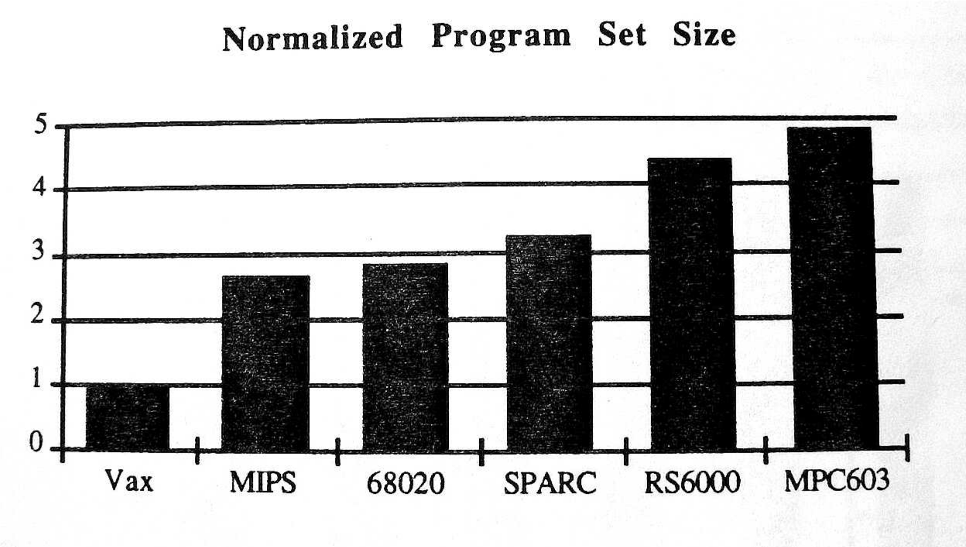


Figure 4. Sum of Program Sizes for Each Machine (Normalized to the VAX 11/750)

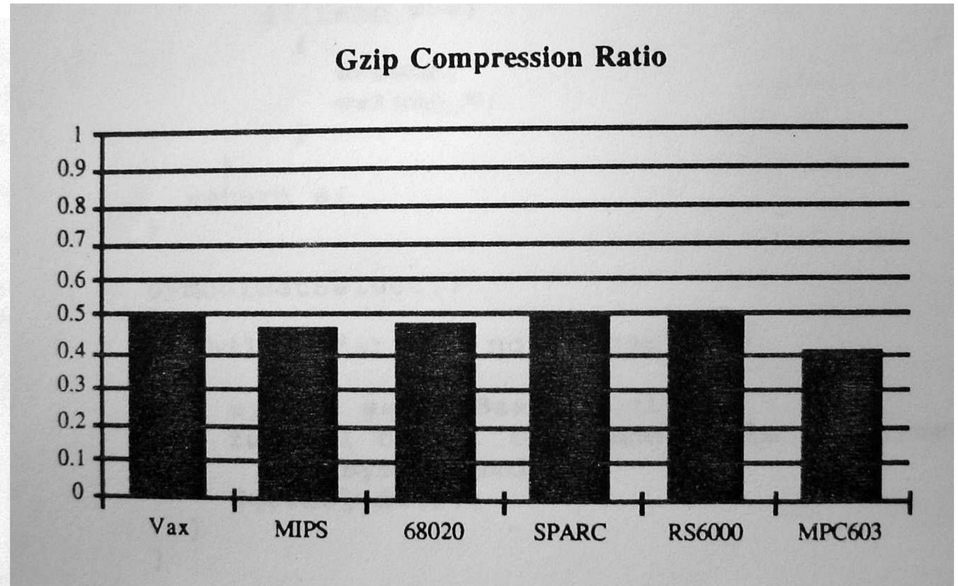


Figure 8. Gzip Compression Ratio

[Kozuch & Wolfe, Int. Conf. on Computer Design, 1994]

Data compression assumptions

	For generic data	For computer programs
Type	Lossless or lossy.	Lossless.
Data length	Possibly infinite.	Finite.
Number of passes	Single.	No restrictions.
Input context	Long.	Short (< 1000 bytes).
Decompression entry point	From beginning only.	From any instruction or function boundary.
Code alignment	Bit-aligned.	Probably word-aligned for fast decoding.
Compression speed	Important for real-time applications.	Not important. Done at compile time.
Data content	Use original data.	May apply code optimizations that result in better compression. (e.g. register allocation)
Example:	Gzip	CodePack

Example of dictionary compression

ADI SHARC DSP code. (from go:g2.c in SPEC 95)

Original	Indices		Dictionary
`cjump readfile (db)`	1	1	`cjump readfile (db)`
`dm(i7,m7)=r2`	2	2	`dm(i7,m7)=r2`
`dm(i7,m7)=pc`	3	3	`dm(i7,m7)=pc`
`cjump life (db)`	4	4	`cjump life (db)`
`dm(i7,m7)=r2`	2	5	`cjump getscore (db)`
`dm(i7,m7)=pc`	3		
`cjump getscore (db)`	5		
`dm(i7,m7)=r2`	2		
`dm(i7,m7)=pc`	3		

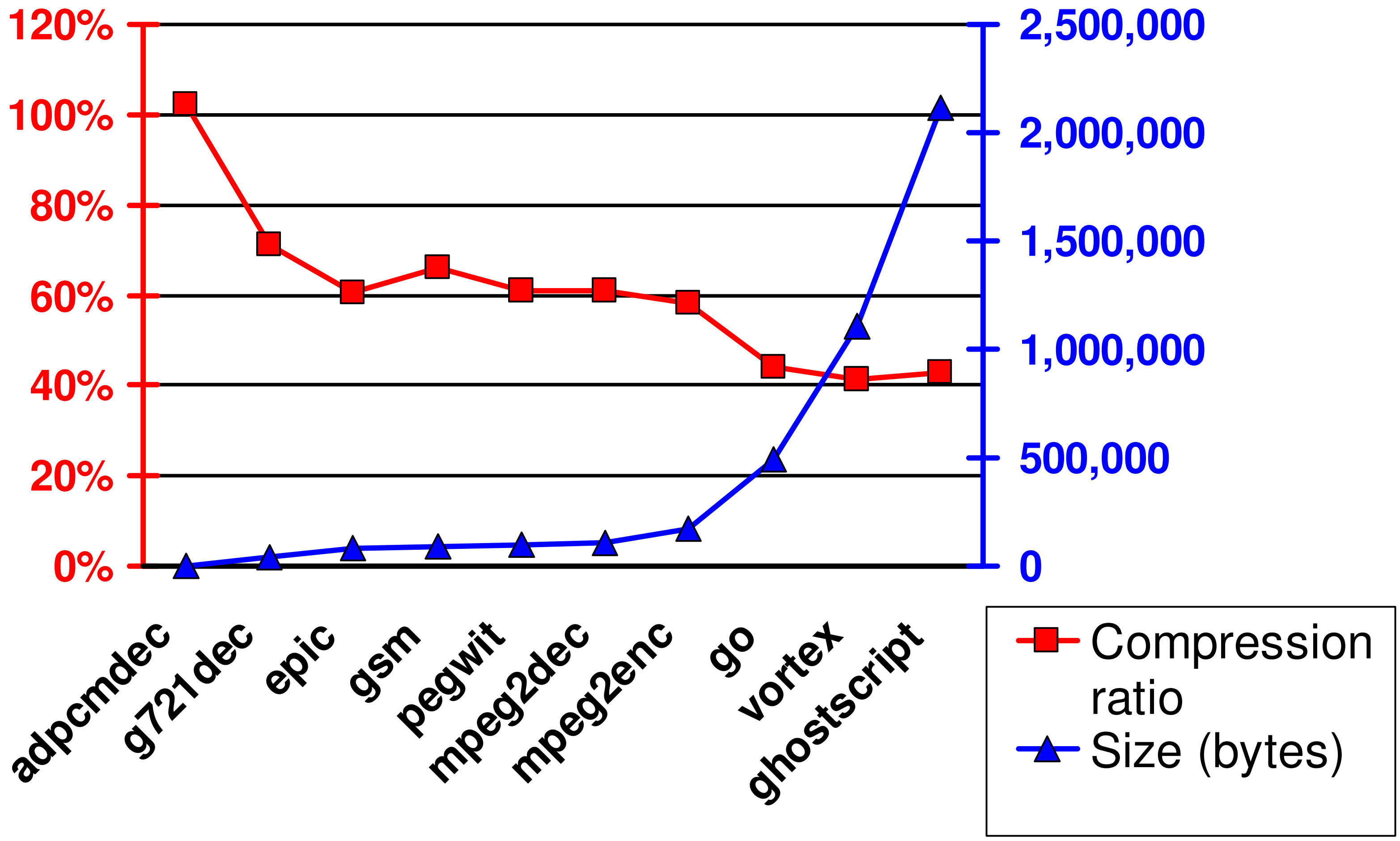
48-bits wide — **16-bits wide** — **48-bits wide**

54 bytes — **18+30 = 48 bytes**

Compression ratio = 48/54 = .89

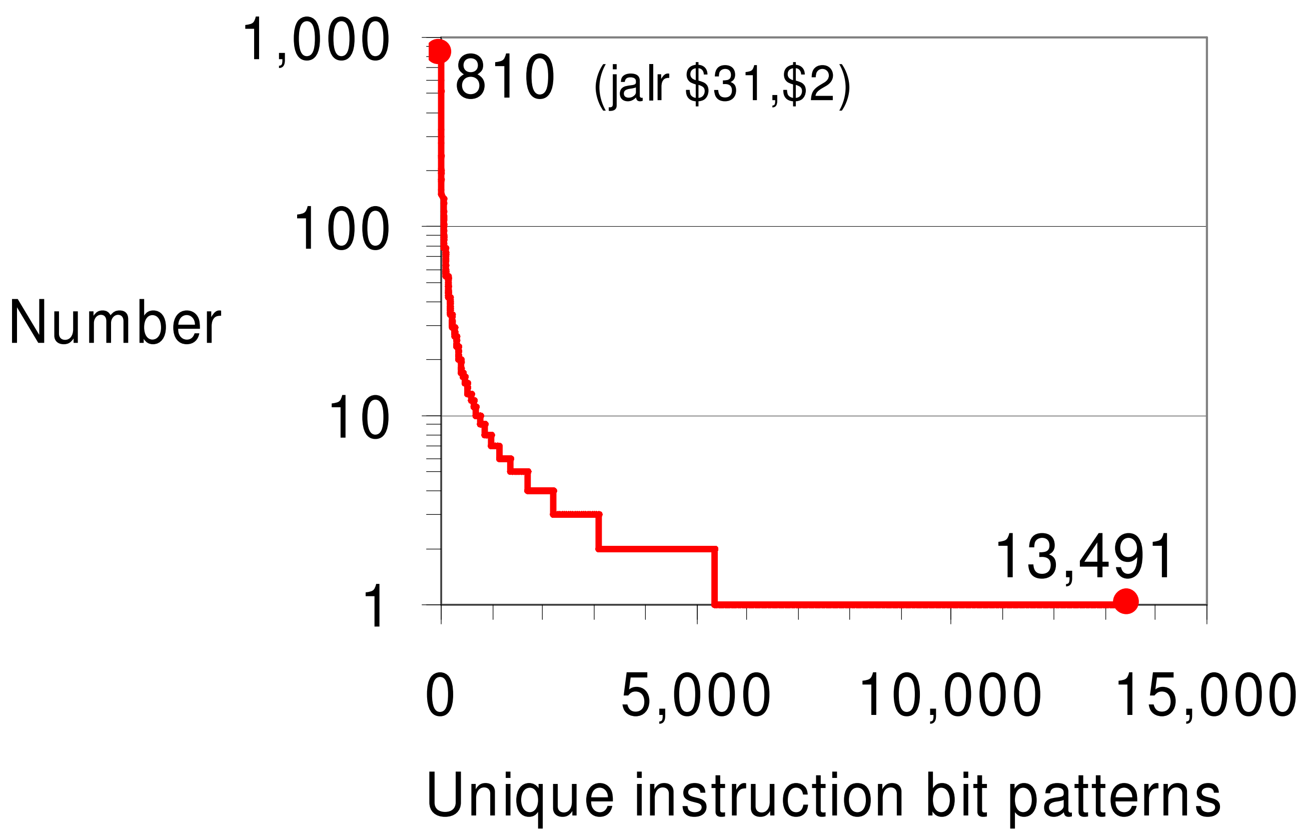
SHARC Experiments

- **Dictionary compression applied to SHARC DSP programs**
- **Instructions are 6 bytes long. Contain up to 3 operations.**



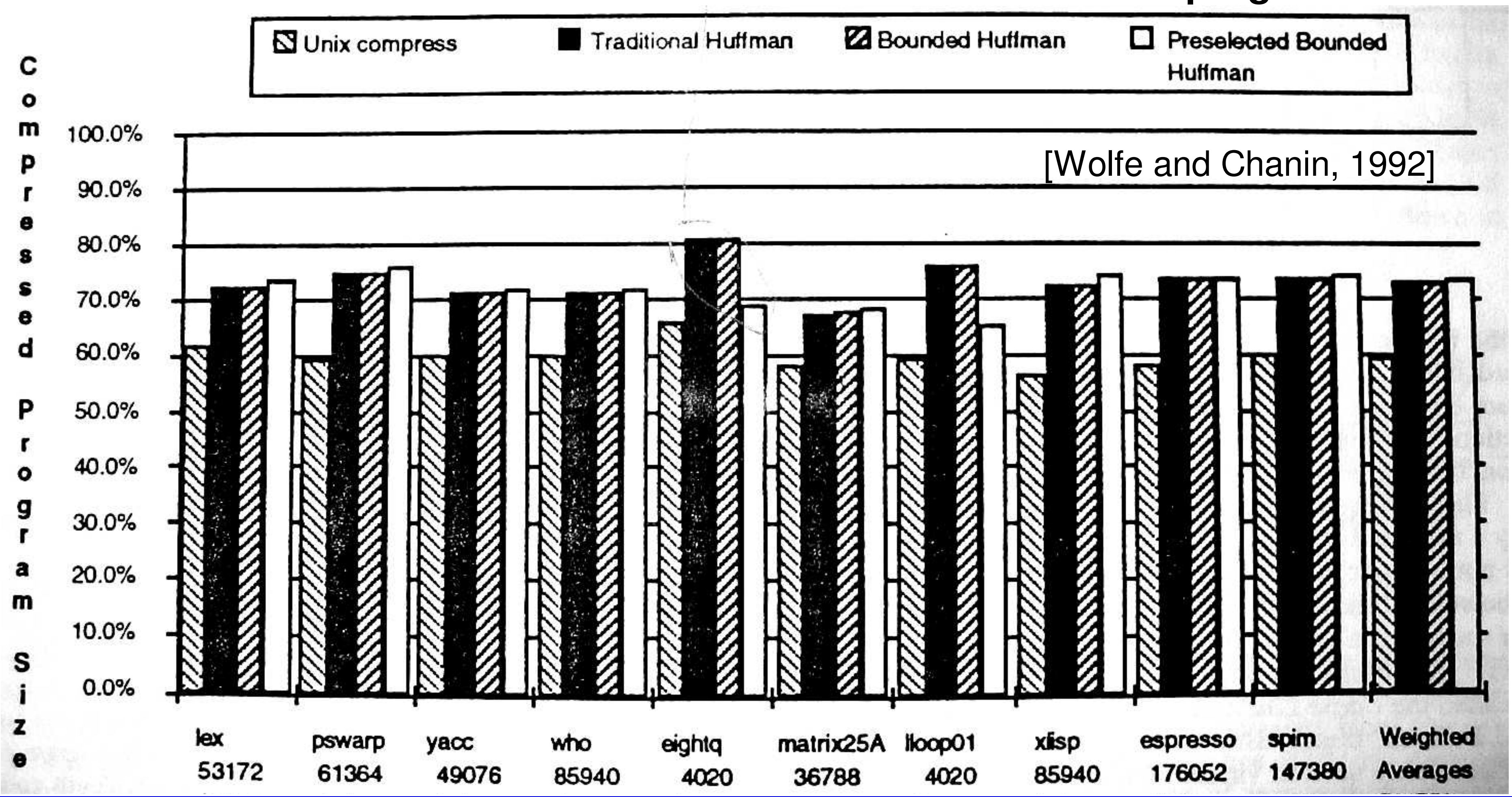
Instruction-based dictionary compression

- **ljpeg benchmark** (MIPS gcc 2.7.2.3 -O2)
  - 49,566 static instructions
  - 13,491 unique instructions
  - 1% of unique instructions cover 29% of static instructions



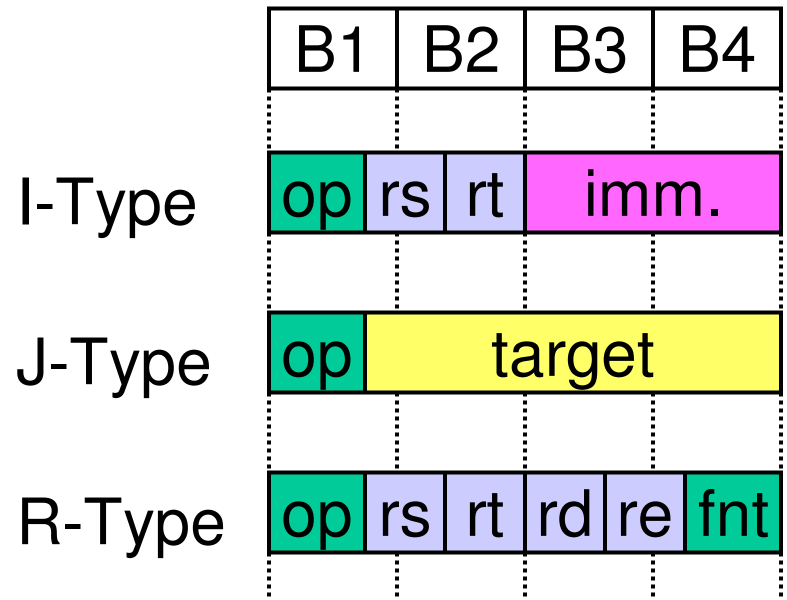
Byte-oriented Huffman compression

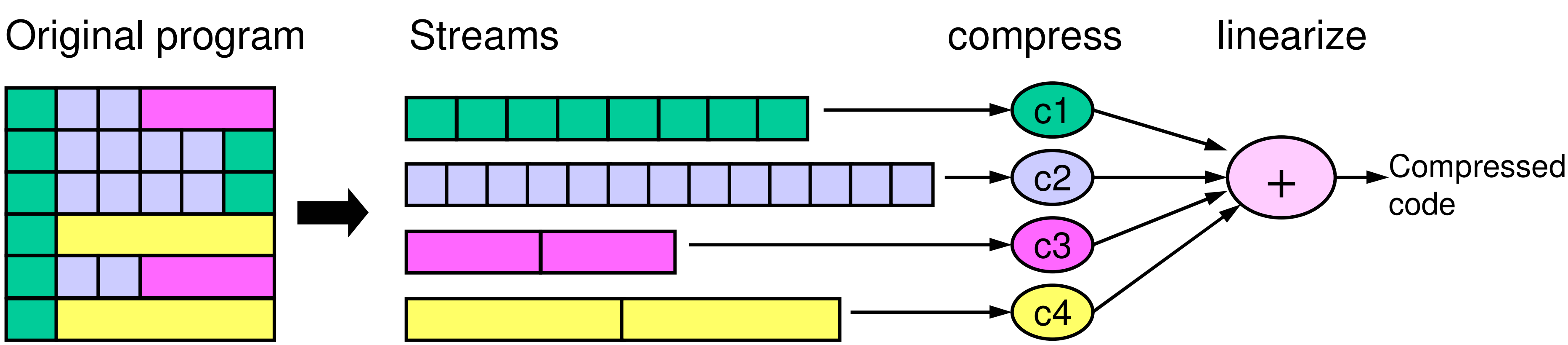
- **Symbols are 8-bit bytes**
- **Bounded Huffman: limit codes to 16-bits max**
  - Use escape code to encode original byte if code is longer than 16 bits.
- **Preselected Bounded Huffman: use the same codes for each program**



From bytes to fields

- **MIPS instruction format**
  - 32-bit fixed-length instructions
  - 3 types of instructions
  - Fields do not align to byte boundaries
    - Poor for 1-byte Huffman encoding
- **Stream compression**
  - Compress each field type separately
  - Improve similarity between symbols





Semiadaptive Dictionary Compression

- **Example of a higher-order model for code compression**
- **SADC achieves 50% compression ratios**
  - Divide MIPS instruction into streams for each instruction field.
    - Opcode
    - Register
    - Immediate
    - Long immediate
  - Markov model for next-bit probabilities.
  - Use arithmetic coding on each stream.
  - Opcode dictionary to encode frequently used sequences of opcodes.
  - Semi-adaptive: probabilities and dictionary are different for each program.

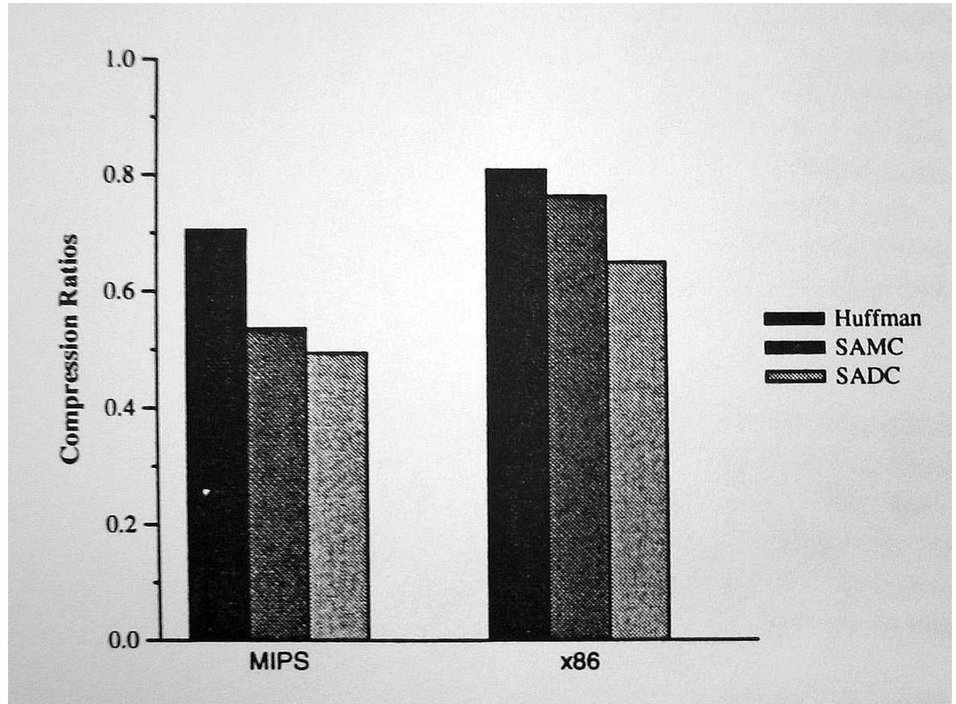
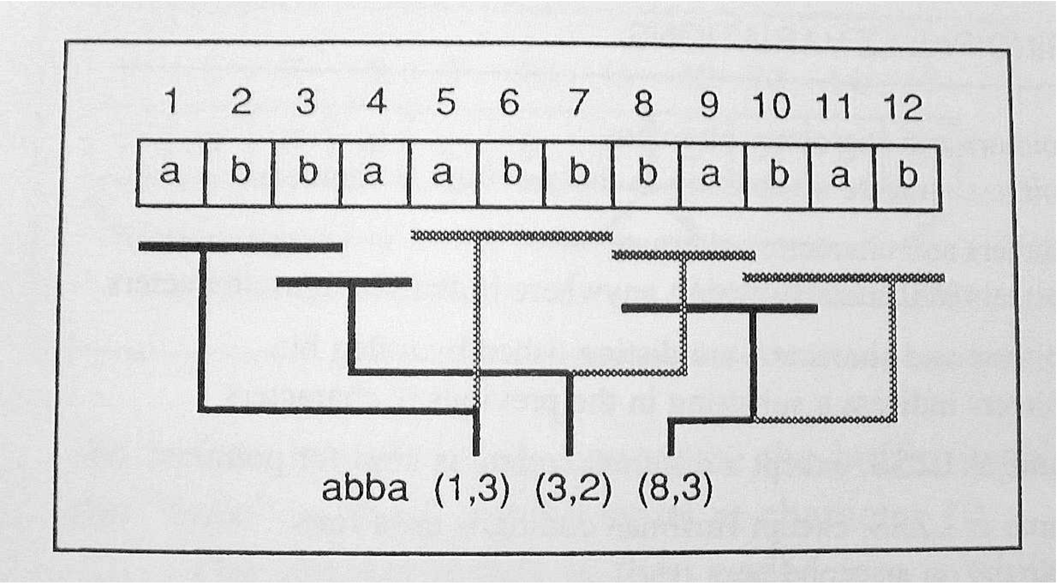


Figure 9: Instruction Compression Algorithms

[Lekatsas & Wolf, 1998]

Lempel-Ziv: adaptive dictionaries

- **Encode several symbols at a time**
  - Create dictionary of recently seen strings of symbols
- **Use sliding window of recent input to find matching strings**
  - Assumes that next symbols will look similar to ones recently seen.
  - Automatically adapts as symbol frequencies change
  - Larger window (context) yields better compression
- **Basis for popular compression programs: pkzip, gzip, etc.**



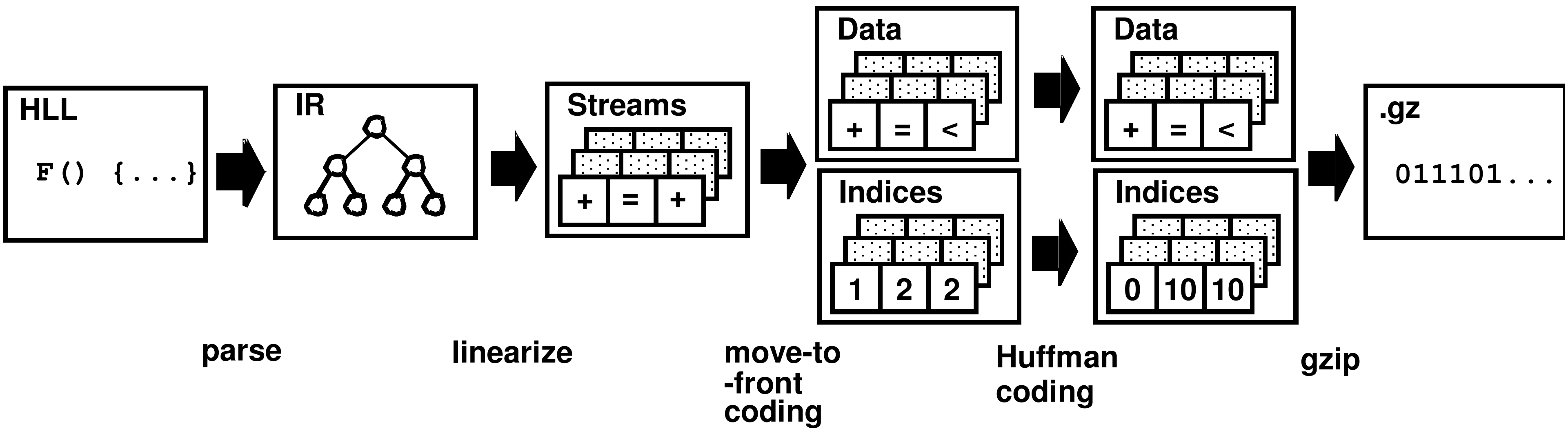
Dictionary

abb	(1,3)
ba	(3,2)
bab	(8,3)
…	

[Bell, Cleary, and Witten, *Text Compression*]

“Wire code”

- **Overview**
  - [Ernst et al., 1997]
  - Wire codes
  - Compress compiler representation
- **Results**
  - 1/5 size of SPARC program
  - Good for sending code over a network
  - Must decompress and compile using just-in-time compiler



Implementations

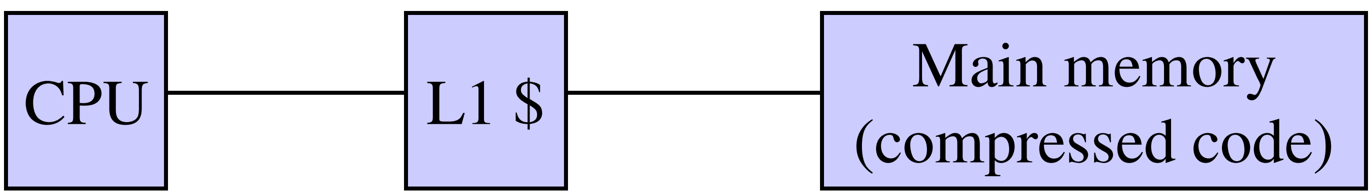
General issues

CCRP: widely studied method

CodePack: a commercial solution

Issues

- **Where to decompress?**
  - Between memory and L1 cache. (Focus of this talk)
    - Part of the memory system. Invisible to core processor.
    - Code in the cache can execute without more decompression.
      - Improves CPU performance.
  - Between L1 cache and instruction execution.
    - Part of core processor. Decoder must be modified.
    - Instructions must be decompessed each time they are executed.
    - Fewer off-chip accesses.
      - Improves memory performance.



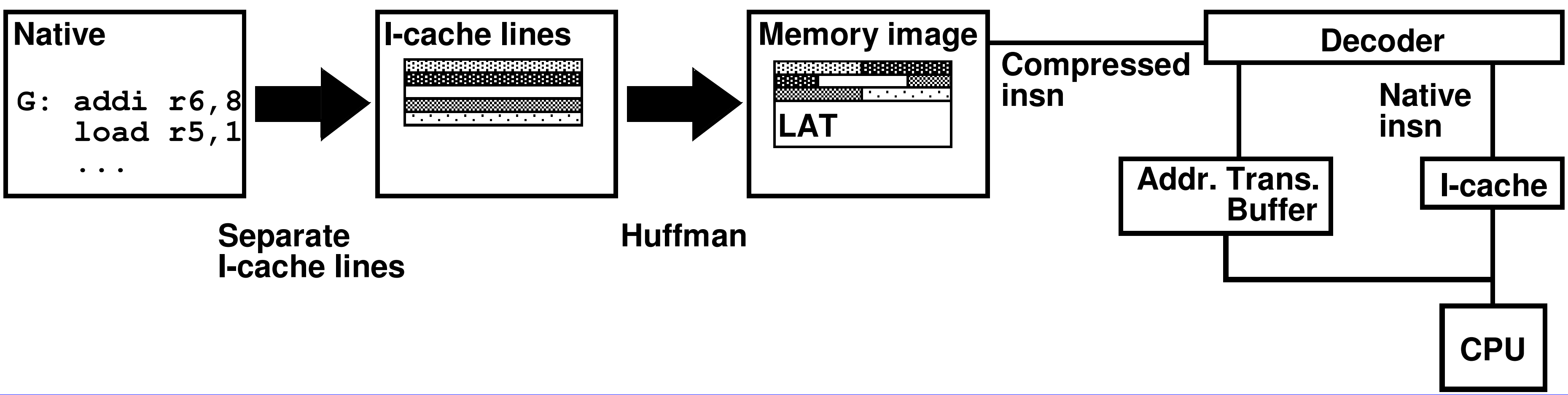
More issues

- **Blocking: unit of decompression**
  - Large blocks allow for more context and better compression
  - Large blocks slow execution
    - Jumps into middle of block: must decompress first instructions
    - May jump out of block before reaching the end. Decompress unused instructions.

- **Should dictionary be different for each program?**
  - Smaller compressed size
  - Adapts better to each compiler and program.
  - Could be a barrier to wide adoption. Must re-load decoder to decompress the next program.

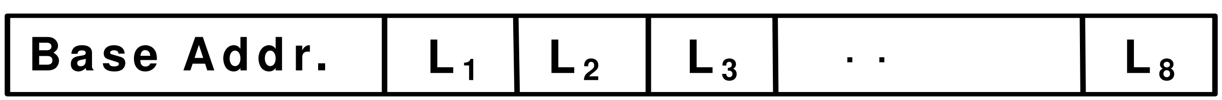
CCRP: compressed code RISC processor

- **Overview**
  - [Kozuch and Wolfe, 1994]; [Benes et al. 1998]
  - Compressed Code RISC Processor (CCPR)
  - Huffman encode cache lines
  - Address translation for random access to cache lines.
    - LAT: line address table
  - Programs run from -10% to +30% faster than conventional system.
    - Faster when memory is slow or instruction cache miss ratio is high.
- **Results**
  - 73% compression ratio for MIPS
  - 0.8µ CMOS, 0.75 mm^2, decompression output 163 MB/s

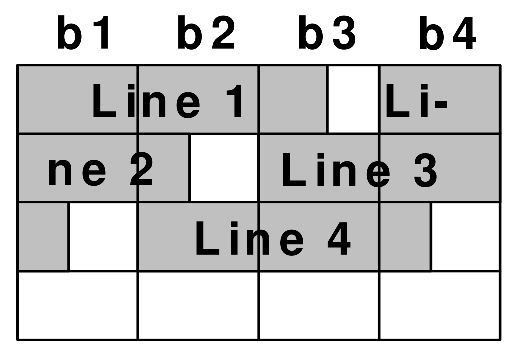


CCRP address translation

- **LAT: line address table**
  - Input: a program address
  - Output: the corresponding compressed code address
- **LAT entry (8 bytes)**
  - Encodes 8 cache lines. 24-bit base address, 5 bit offsets.
  - Base address: first address of the compressed block
  - L1…Lk: offset to compressed cache line
  - Address of nth cache line = base + L1 + ... + Ln
  - LAT overhead is 3% of compressed code.



LAT Entry

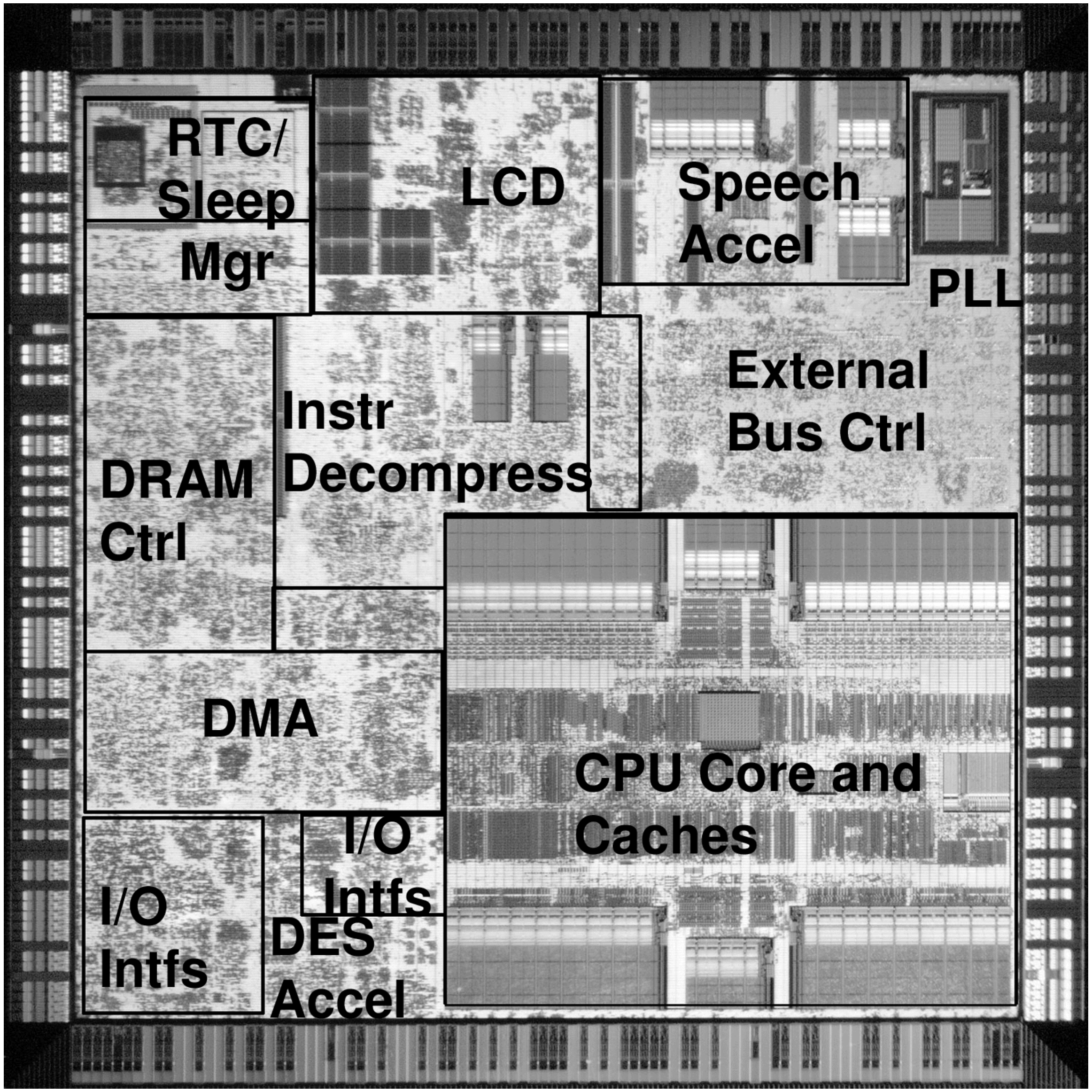


Byte-aligned compressed cache lines

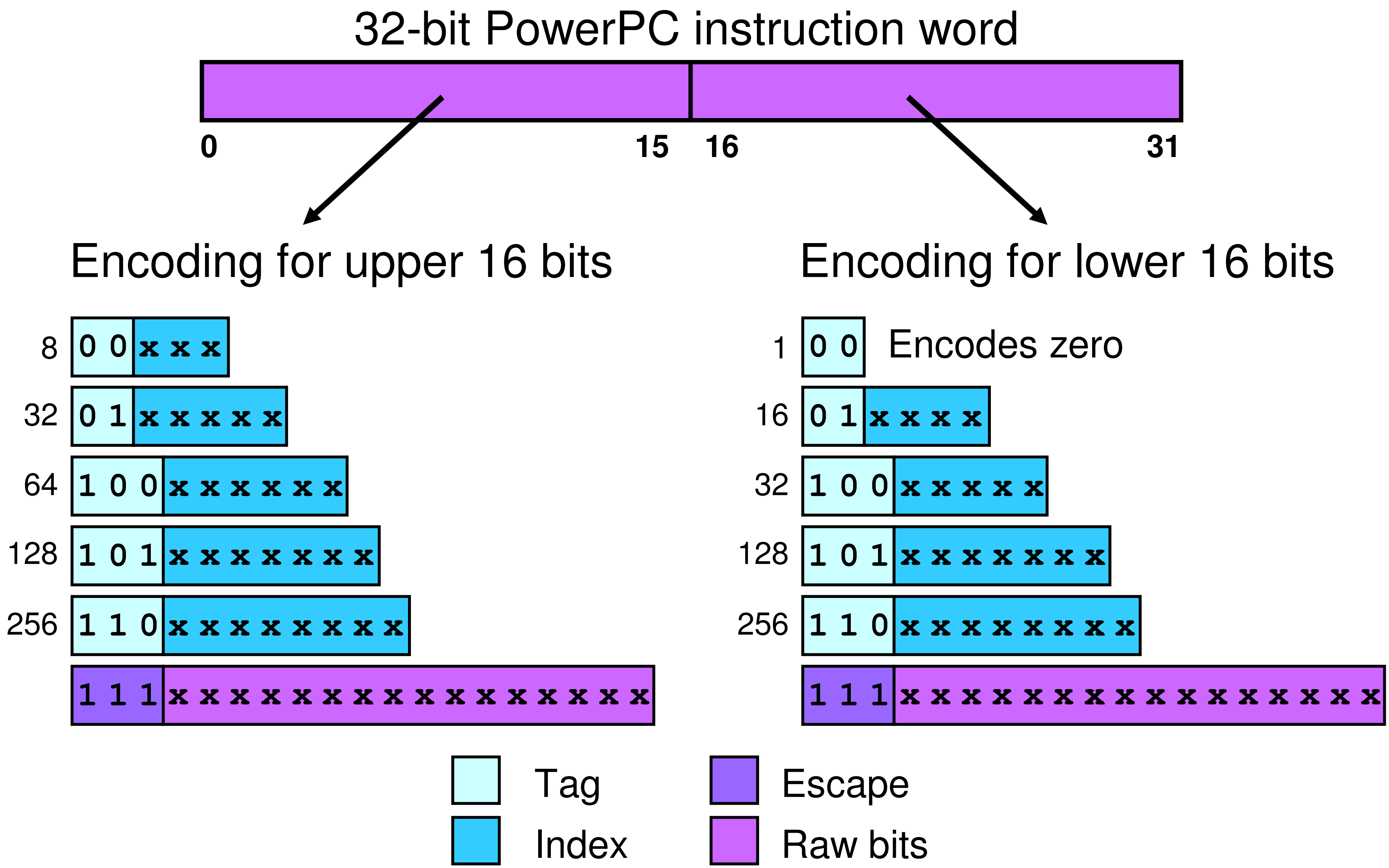
CodePack

- **Overview**
  - The only widely-deployed code compression method
  - IBM
  - PowerPC instruction set
  - 60% compression ratio, ±10% performance [IBM]
    - performance gain due to prefetching
- **Implementation**
  - Binary executables are compressed after compilation
  - Compression dictionaries tuned to application
  - Decompression occurs on L1 cache miss
    - L1 caches hold decompressed data
    - Decompress 2 cache lines at a time (16 insns)
  - PowerPC core is unaware of compression

PowerPC 405 LP

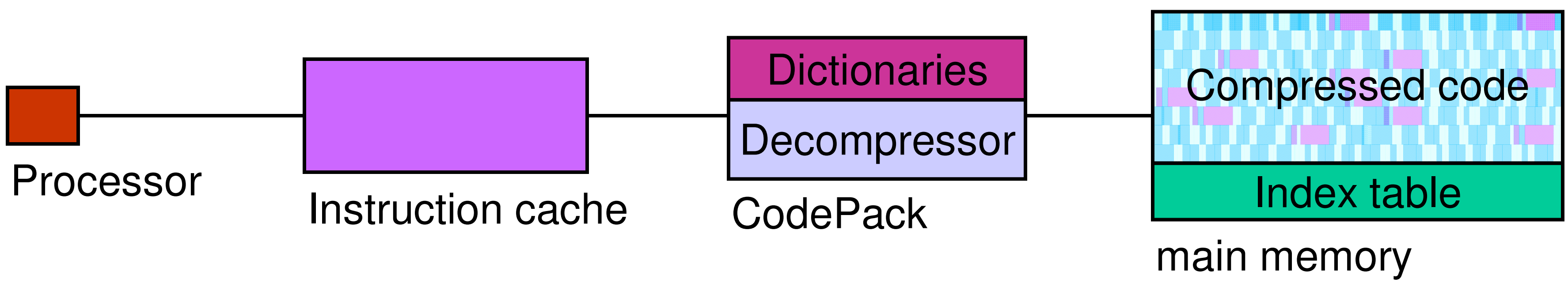


CodePack encoding



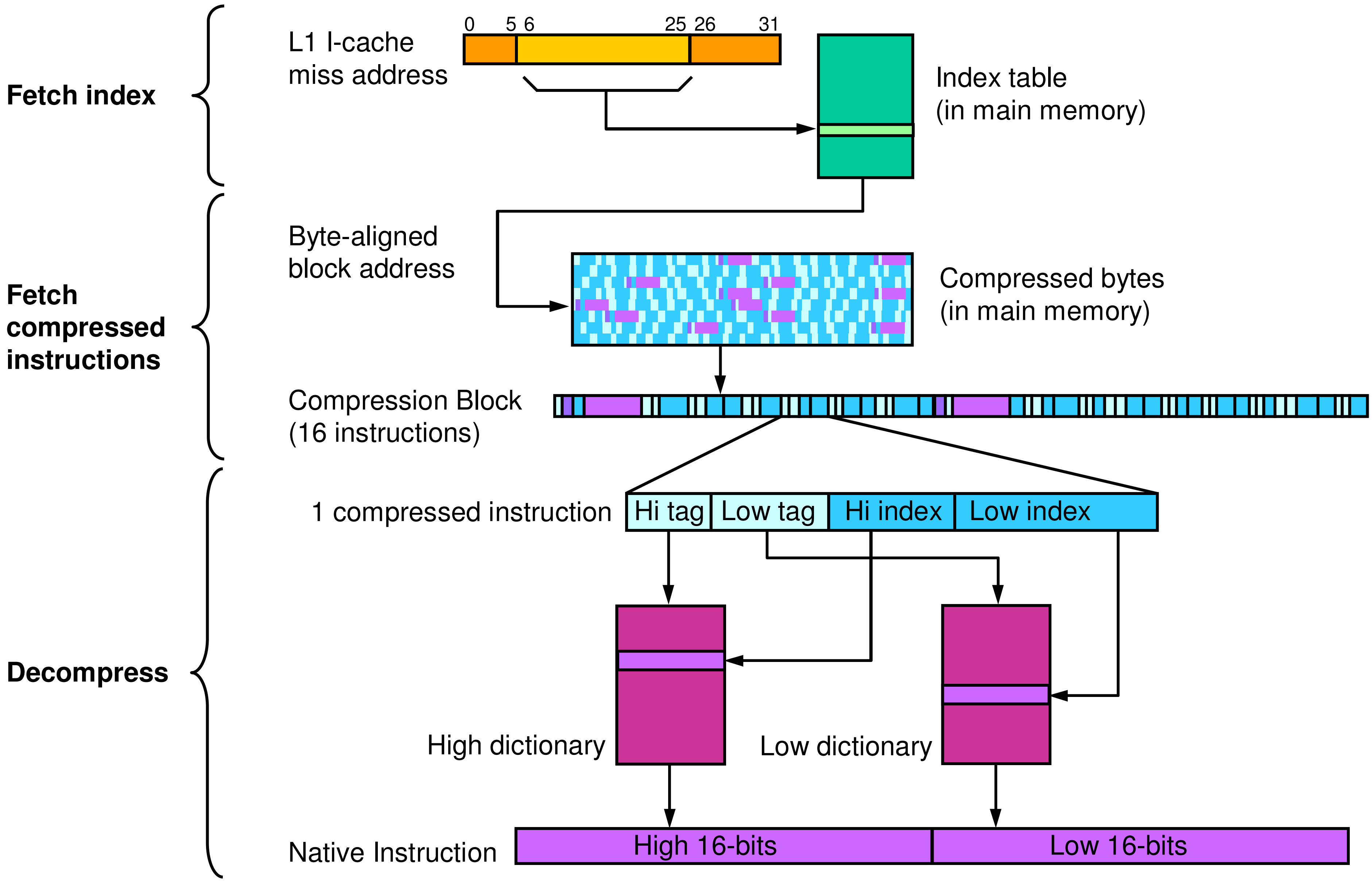
CodePack system

- **CodePack is part of the memory system**
  - After L1 instruction cache
- **Dictionaries**
  - Contain 16-bit upper and lower halves of instructions
- **Index table**
  - Maps instruction address to compressed code address



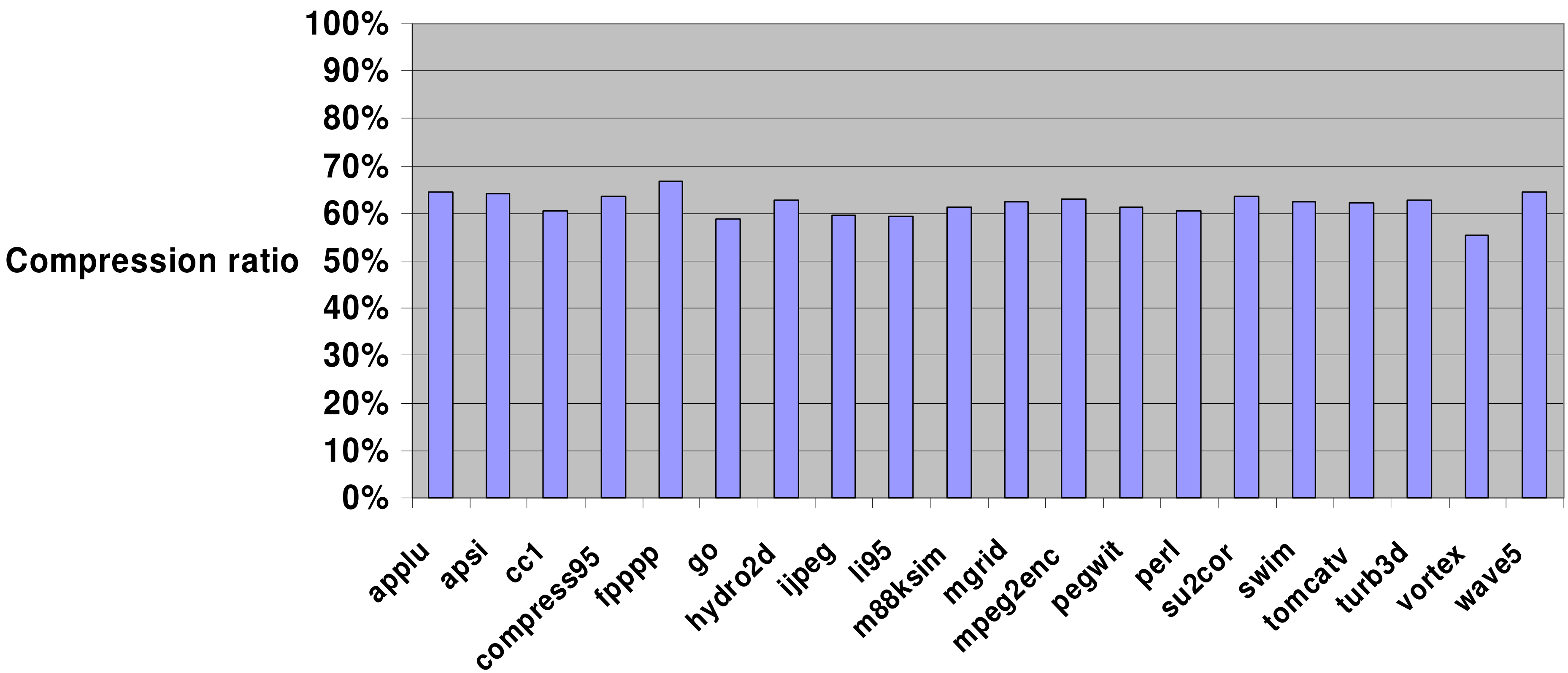
Instruction memory hierarchy

CodePack decompression



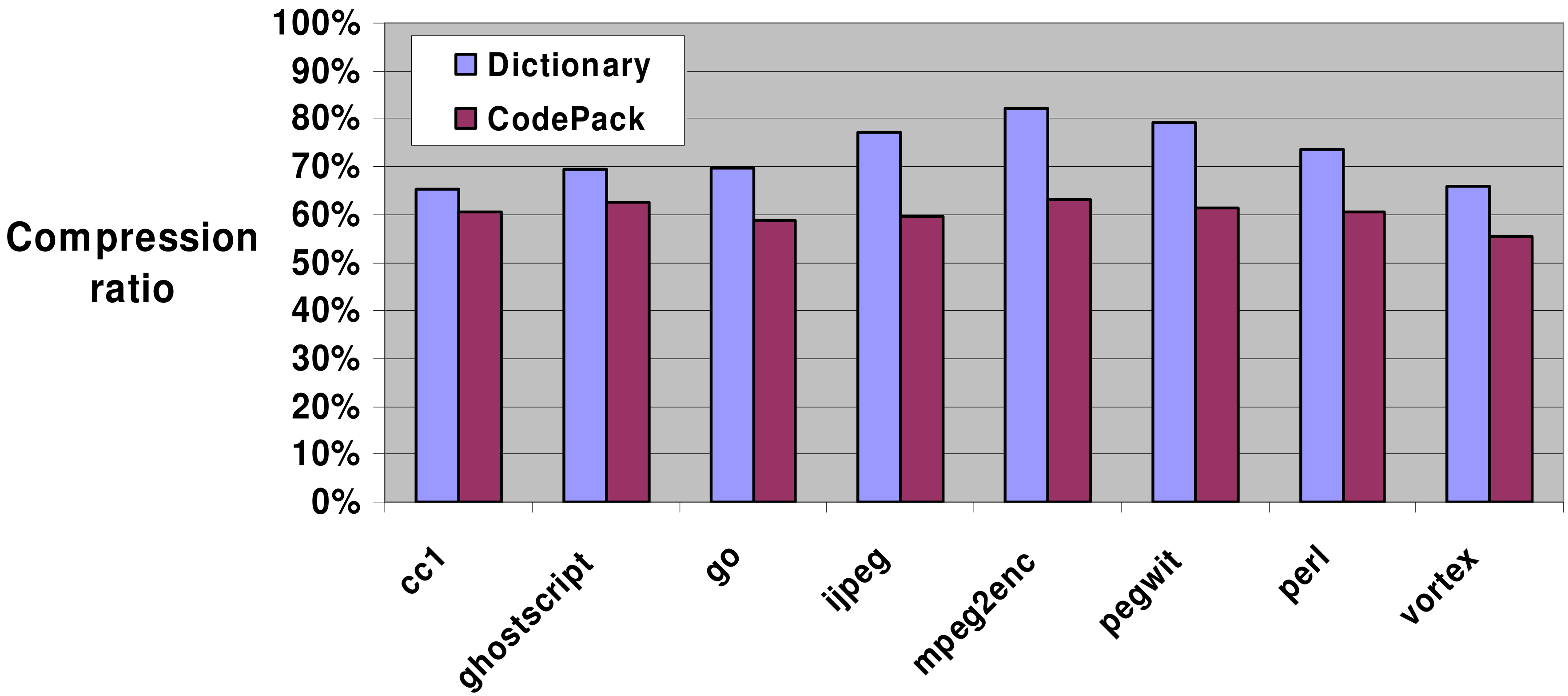
Compression ratio

- $$compression\ ratio = \frac{compressed\ size}{original\ size}$$
- **Average: 62%**



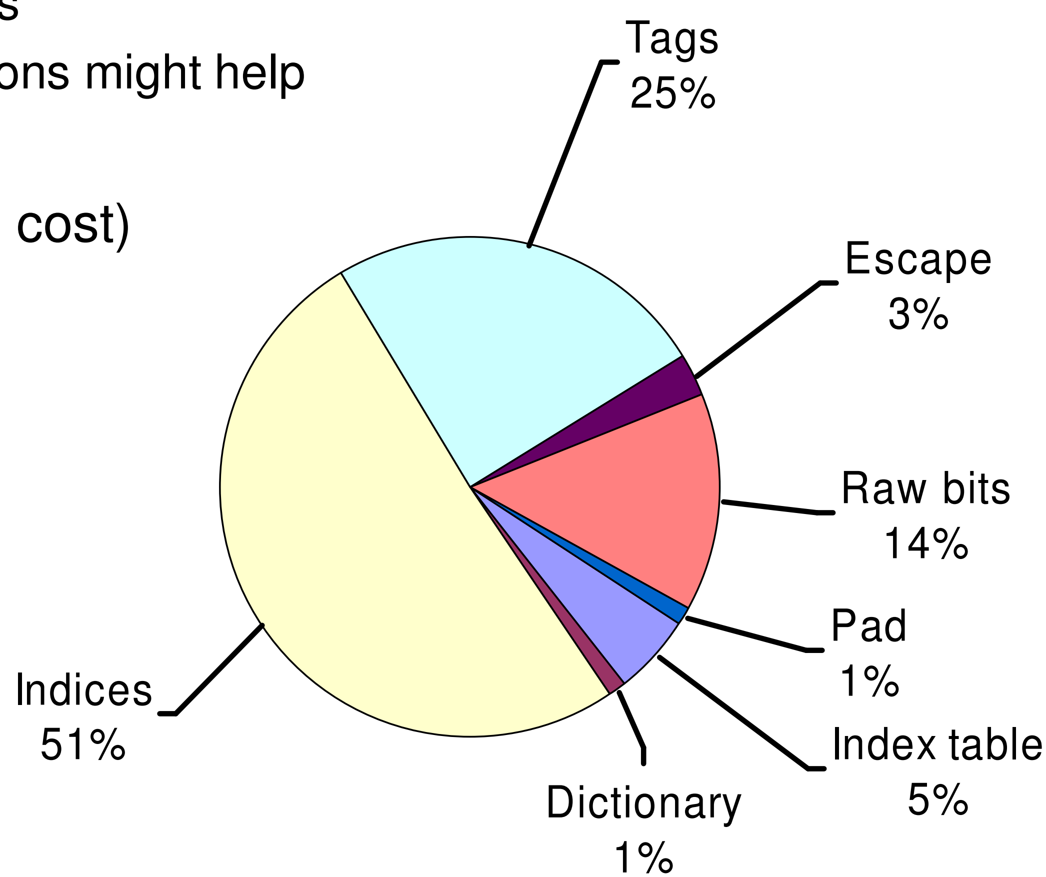
Compression ratio

- $$compression\ ratio = \frac{compressed\ size}{original\ size}$$
  - CodePack: 55% - 63%
  - Dictionary: 65% - 82%



CodePack programs

- **Compressed executable**
  - 17%-25% raw bits: not compressed!
    - Includes escape bits
    - Compiler optimizations might help
  - 5% index table
  - 2KB dictionary (fixed cost)
  - 1% pad bits



go

Impact and Issues

Performance

Energy

Compiler optimization

Can code compression improve performance?

- **Evidence both ways.**

- **Yes:**
  - Fewer main memory accesses required.
  - Less swapping, less use of overlays, etc.
  - Loading compressed code from disk and compiling it can be faster than loading native code.
  - If compressed instructions can be stored in cache, then caches are effectively bigger.

- **No:**
  - Decode time can increase latency of executing instruction
    - Compressed instruction in L1 cache must be decoded each time they are executed.
  - Increased cache miss latency (CodePack and CCRP)

Can code compression save power?

- **Many studies, but no definitive answers.**
  - Results are simulated, not measured on real hardware.
- **Yes:**
  - Less data is transmitted over memory bus: less bit flips.
  - Less memory is required.
  - Less memory accesses.
  - Narrower memory bus can be used.
  - If code runs faster, power-down modes can be used more often.
- **No:**
  - Slowdown causes CPU and peripherals to stay in power-up mode longer.
  - Time for program to complete has a first-order impact on energy used.
    - CPU energy cost overwhelms any gain in memory/bus energy.

Compiler optimizations for code compression

- **Example: Instruction selection**
  - Repetition improves compression
  - Choose PC-relative or absolute branches for similarity
  - Improves compression ratio by over 10% for Spec95
  - Reduces dictionary size by 50% for some benchmarks
  - Removes many “singleton” instructions

PC-relative branches to same target cause different instruction words

```
80d4: e59f0010  ldr r0,&”hello”
80d8: eb000237  bl  89bc <printf>
80dc: e59f000c  ldr r0,&”goodbye”
80e0: eb000235  bl  89bc <printf>
```

Using absolute addressing makes instruction words the same and compressible

```
80d4: e59f0010  ldr r0,&”hello”
80d8: eb0089bc  bla 89bc <printf>
80dc: e59f000c  ldr r0,&”goodbye”
80e0: eb0089bc  bla 89bc <printf>
```

Alternatives to code compression

New instruction sets

Compiler optimizations

Alternatives to code compression

- **New instruction set**
  - ARM → Thumb
  - Smaller, but could still compressed more.
- **Compiler optimization for small code-size**
  - Limited effect on code size. 10% is typical.
  - Procedure abstraction

Thumb

- **Thumb instruction set is based on ARM.**
  - Processor can switch between ARM and Thumb instruction sets.
- **16-bit instructions (ARM is 32-bit)**
- **8 32-bit general registers (ARM has 16)**
- **Destructive (2 register) instructions**
- **Load/store architecture**
- **Removed instructions**
  - Multiply-accumulate
  - Atomic memory operations
  - Reverse subtract
  - Co-processor operations
  - Conditional Execution
  - In-line shifts

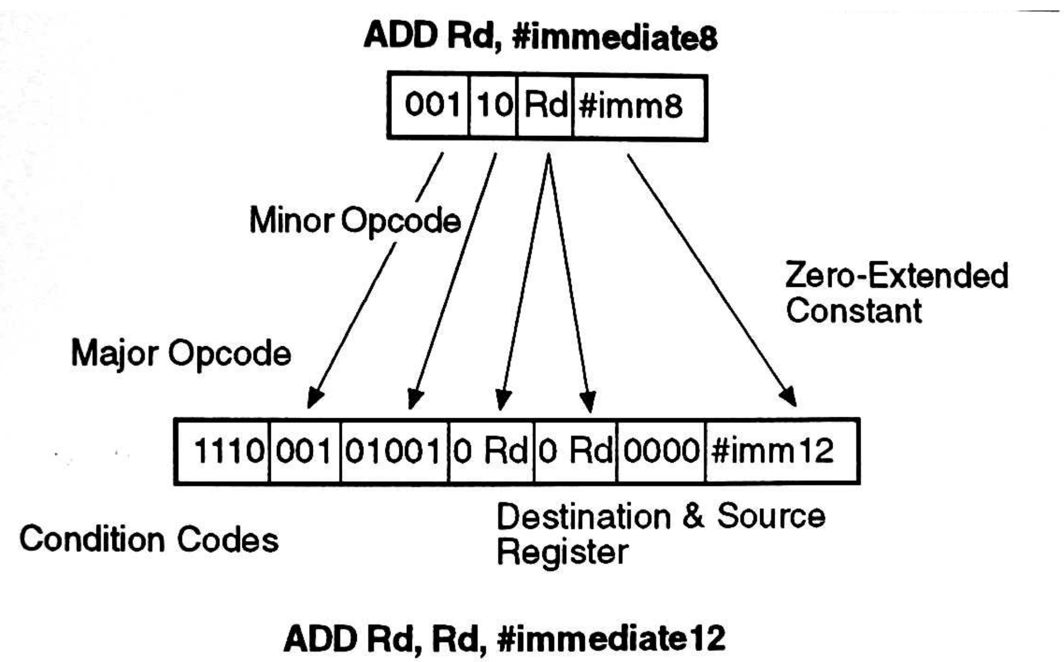


Figure 1. The Thumb instruction decompression logic expands op-codes and register references into their 32-bit ARM equivalents.

[Microprocessor Report, 1995]

Thumb performance

- **Compression ratio = 0.7**
- **Runs faster narrow busses.**
  - Instructions can be read with fewer memory accesses
- **Runs slower on wide busses.**
  - 15-20% more dynamic instructions are executed.
- **Hybrid programs**
  - Use Thumb for infrequently used functions. (Most of the program.)
  - Use ARM for the few performance-critical functions.
  - Best compilers help you decide how to trade-off code size and performance.

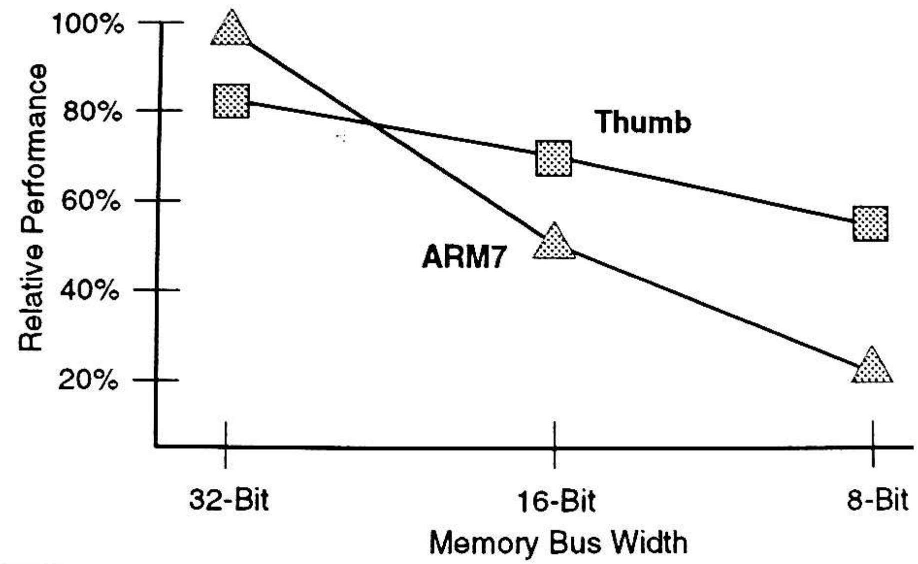


Figure 2. Comparing ARM7 cores with and without Thumb shows the effects of memory bus width on performance. The 16-bit Thumb instruction set surpasses ARM in low-cost systems. (Source: ARM).

[Microprocessor Report, 1995]

Compiler optimizations for small code

- **Procedure abstraction [Standish, 1976]**
  - Use function call mechanism to abstract common code
  - Apply to source code, compiler IR, or object code

```
// Count 2 lists

F() {
  total = 0;
  while (a_ptr) {
    total++;
    a_ptr = a_ptr->next;
  }
  a = total;

  total = 0;
  while (b_ptr) {
    total++;
    b_ptr = b_ptr->next;
  }
  b = total;
}
```

```
// Count 2 lists

int G(p) {
  total = 0;
  while (p) {
    total++;
    p = p->next;
  }
  return(total);
}

F() {
  a = G(a_ptr);
  b = G(b_ptr);

}
```

Conclusions

- **Does code compression help size?**
  - Yes. 30-50% reduction for object code. 80% reduction for compiler IR + JIT.
- **Does code compression help performance?**
  - Possibly, in the right situations. (slow memory, narrow bus)
  - Often decompression step causes systems to run slower.
  - Hybrid programs (compressed and native code) can reduce performance impact.
- **Does code compression help energy consumption?**
  - Helps memory and bus power (fewer accesses)
  - May not help full system power. Remains to be demonstrated.
- **Future?**
  - No new industrial solutions in last few years.
    - But still new ISAs. Thumb-2 can mix 16-bit and 32-bit instructions freely.
    - Larger register sets (IA-64, MMX, Vectors)
  - Sensors networks. An ideal application?
  - Cell phones. Ever smaller with more features.
  - DRAM scaling is slowing: 4x/3years → 2x/3years (driven by PC market)
  - Main memory compression (compress/decompress in memory controller)

References

- **A. Beszédes et al., "Survey of Code-Size Reduction Methods", *ACM Computing Surveys*, Vol. 35, No. 3, September, 2003, pp. 223-267.**
  - Code compaction and compression.
- **"Software Techniques for Program Compaction", Guest editors B. De Sutter and K. De Bosschere, *Communications of the ACM*, Vol. 46, No. 8, August, 2003, pp. 33-34.**
- **The Code Compression Bibliography http://www.iro.umontreal.ca/~latendre/codeCompression/**
  - 150+ citations.
- **T. C. Bell, I. H. Witten, and J. G. Cleary, *Text Compression*, Prentice Hall, 1990.**
  - Good reference on non-lossy data compression.

End