Heads and Tails

A Variable-Length Instruction Format Supporting Parallel Fetch and Decode

Heidi Pan and Krste Asanović

MIT Laboratory for Computer Science

CASES Conference, Nov. 2001

Motivation

- Tight space constraints
  - Cost, power consumption, space constraints
  - Program code size
  - *Variable-length* instructions: more compact but less efficient to fetch and decode
- High performance
  - Deep pipelines or superscalar issue
  - *Fixed-length* instructions: easy to fetch and decode but less compact
- Heads and Tails (HAT) instruction format
  - *Easy to fetch and decode AND compact*

Related Work

- 16-bit version of existing RISC ISAs
- Compressed instructions in main memory
- Dictionary compression
- CISC

16-Bit Versions

- Examples
  - MIPS16 (MIPS), Thumb (Arm)
- Feature(s)
  - Dynamic switching between full-width & half-width

16-Bit Versions, cont'd.

- Advantages
  - Simple decompression of just mapping 16-bit to 32-bit instructions
  - Static code size reduced by ~30-40%
- Disadvantages
  - Can only encode limited subset of operations and operands; more dynamic instructions needed
  - Shorter instructions can sometimes compensate for the increased number of instructions, but performance of systems with instruction cache reduced by ~20%

Compression in Memory

- Examples
  - CCRP, Kemp, Lekatsas, etc.
- Feature(s)
  - Hold compressed instructions in memory then decompress when refilling cache

Compression in Memory, cont'd.

- Advantages
  - Processor unchanged (see regular instructions)
  - Avoids latency & energy consumption of decompression on cache hits
- Disadvantages
  - Decrease effective capacity of cache & increase energy used to fetch cached instructions
  - Cache miss latencies increase
    - Translate pc ; block decompressed sequentially

Dictionary Compression

- Examples
  - Araujo, Benini, Lefurgy, Liao, etc.
- Features
  - Fixed-length code words in instruction stream point to a dictionary holding common instruction sequences
  - Branch address modified to point in compressed instruction stream

Dictionary Compression, cont'd.

- Advantage(s)
  - Decompression is just fast table lookup
- Disadvantages
  - Table fetch adds latency to pipeline, increasing branch mispredict penalties
  - Variable-length codewords interleaved with uncompressed instructions
  - More energy to fetch codeword on top of full-length instruction

CISC

- Examples
  - x86, VAX
- Feature(s)
  - More compact base instruction set
- Advantage(s)
  - Don’t need to dynamically compress and decompressing instructions

CISC cont'd.

- Disadvantages
  - Not designed for parallel fetch and decode
- Solutions
  - P6: brute-force strategy of speculative decodes at every byte position; wastes energy
  - AMD Athlon: predecodes instruction during cache refill to mark boundaries between instructions; still need several cycles after instruction fetch to scan & align
  - Pentium-4: caches decoded micro-ops in trace cache; but cache misses longer latency and still full-size micro-ops

Heads and Tails Design Goals

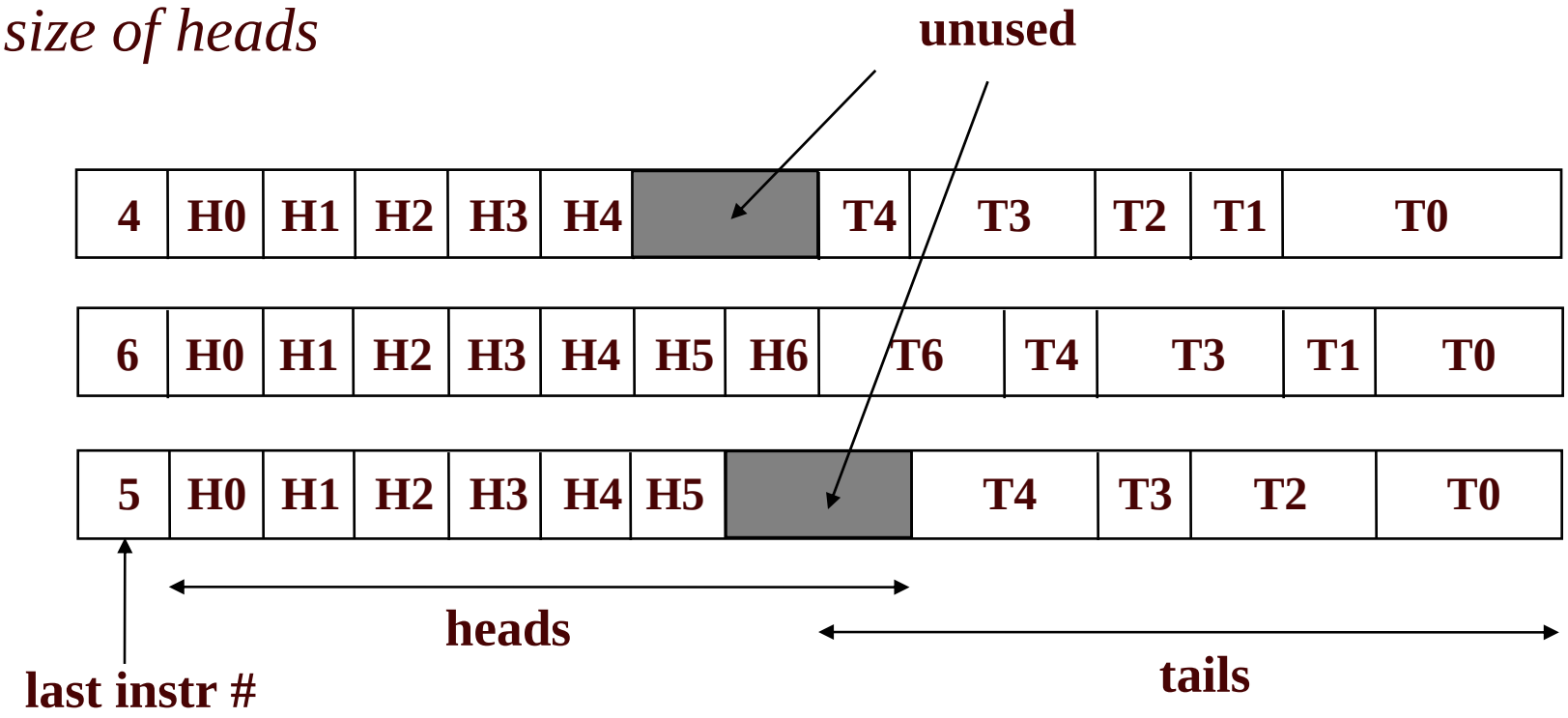
- Variable-length instructions that are easily fetched and decoded
- Compact instructions in memory and cache
- Format applicable for both compressing existing fixed-length ISA or creating new variable-length ISA

Heads and Tails Format

- Each instruction split into two portions: fixed-length *head* & variable-length *tail*
- Multiple instructions packed into a fixed-length *bundle*
- A cache line can have multiple bundles

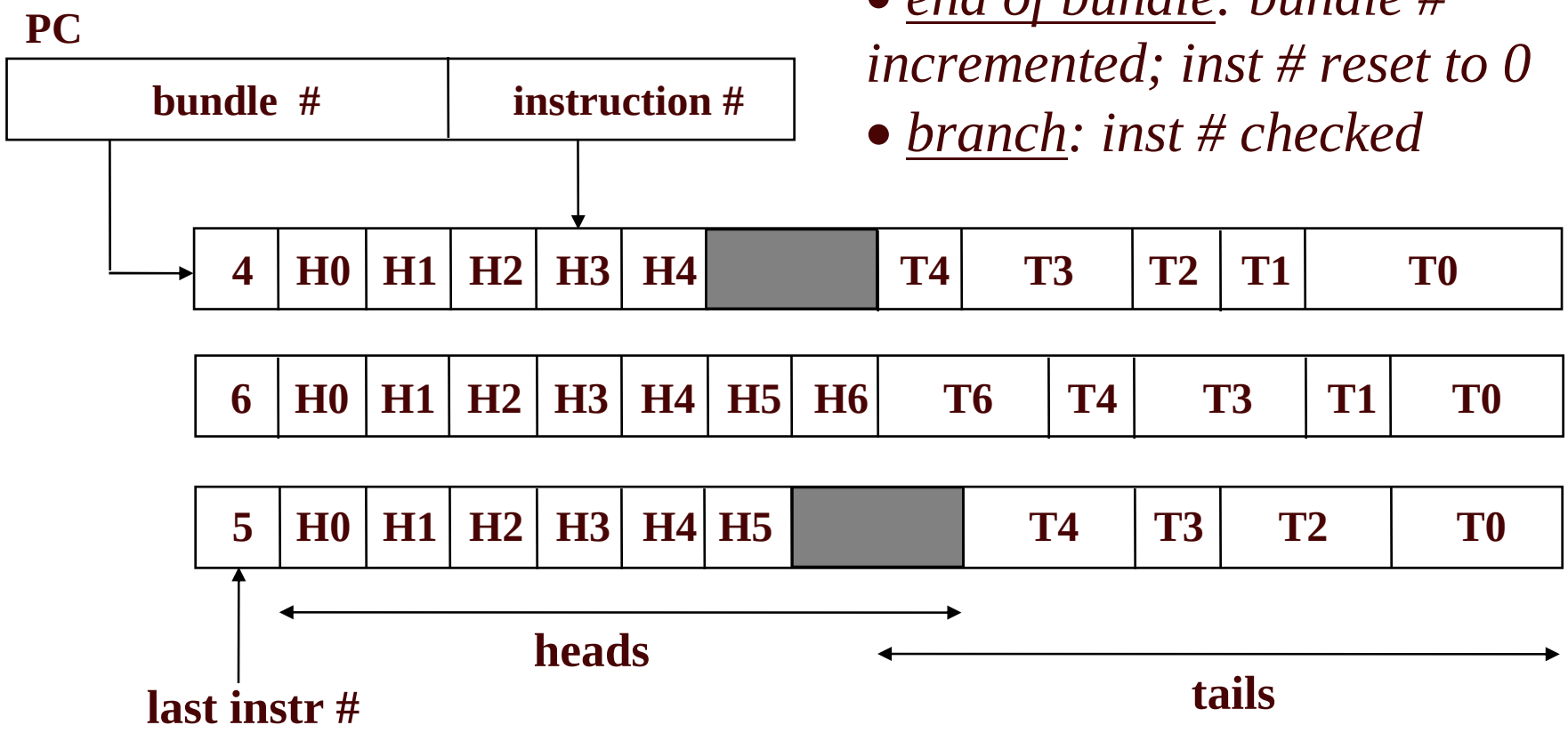
Heads and Tails Format

- *not all heads must have tails*
- *tails at fixed granularity*
- *granularity of tails independent of size of heads*



Heads and Tails Format

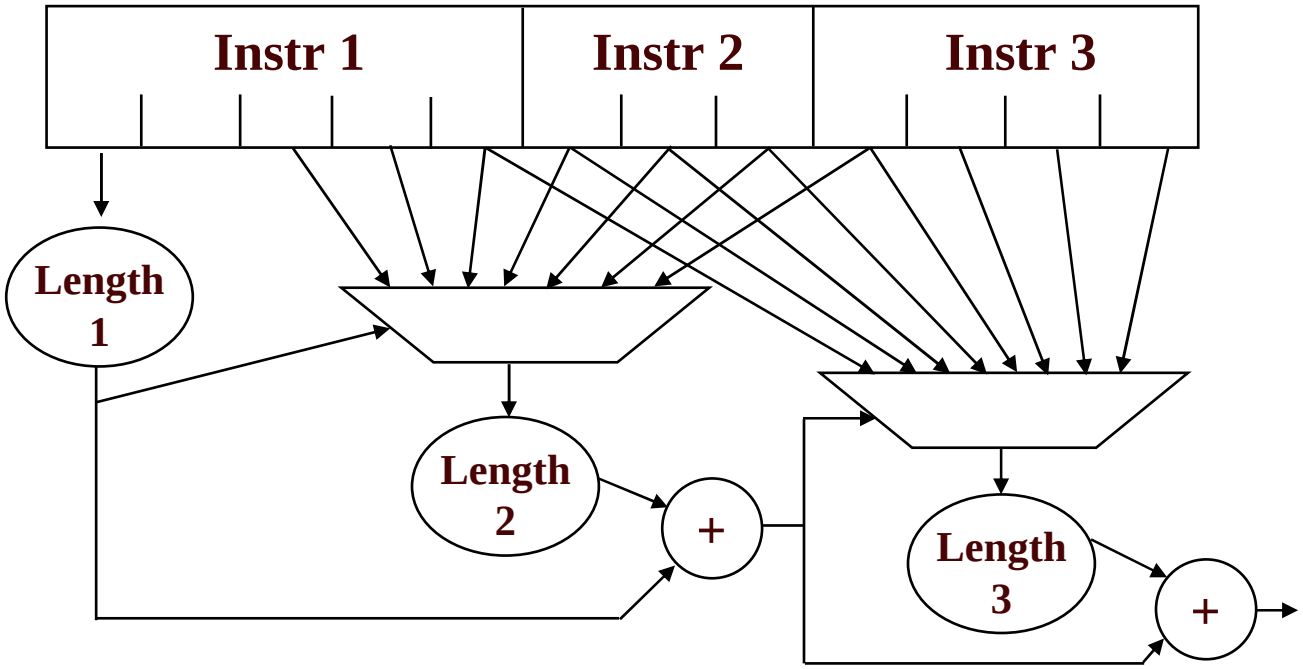
- *sequential: pc incremented*
- *end of bundle: bundle # incremented; inst # reset to 0*
- *branch: inst # checked*



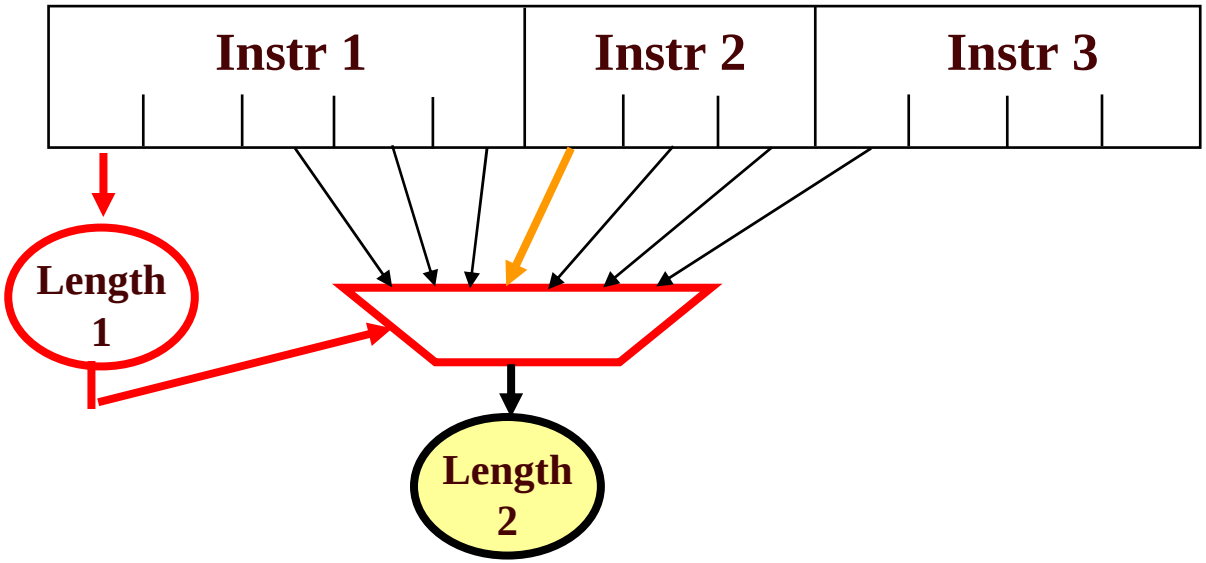
Length Decoding

- Fixed-length heads enable parallel fetch and decode
- Heads contain information to locate corresponding tail
- Even though head must be decoded before finding tail, still faster than conventional variable-length schemes
- Also, tails generally contain less critical information needed later in the pipeline

Conventional VL Length-Decoding

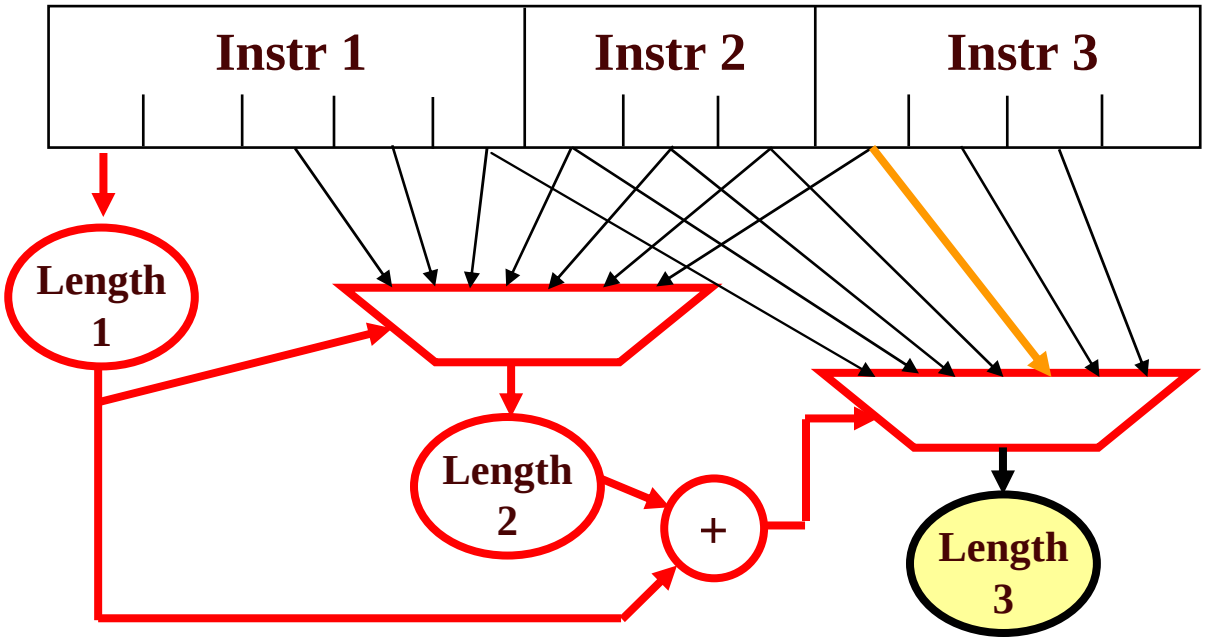


Conventional VL Length-Decoding



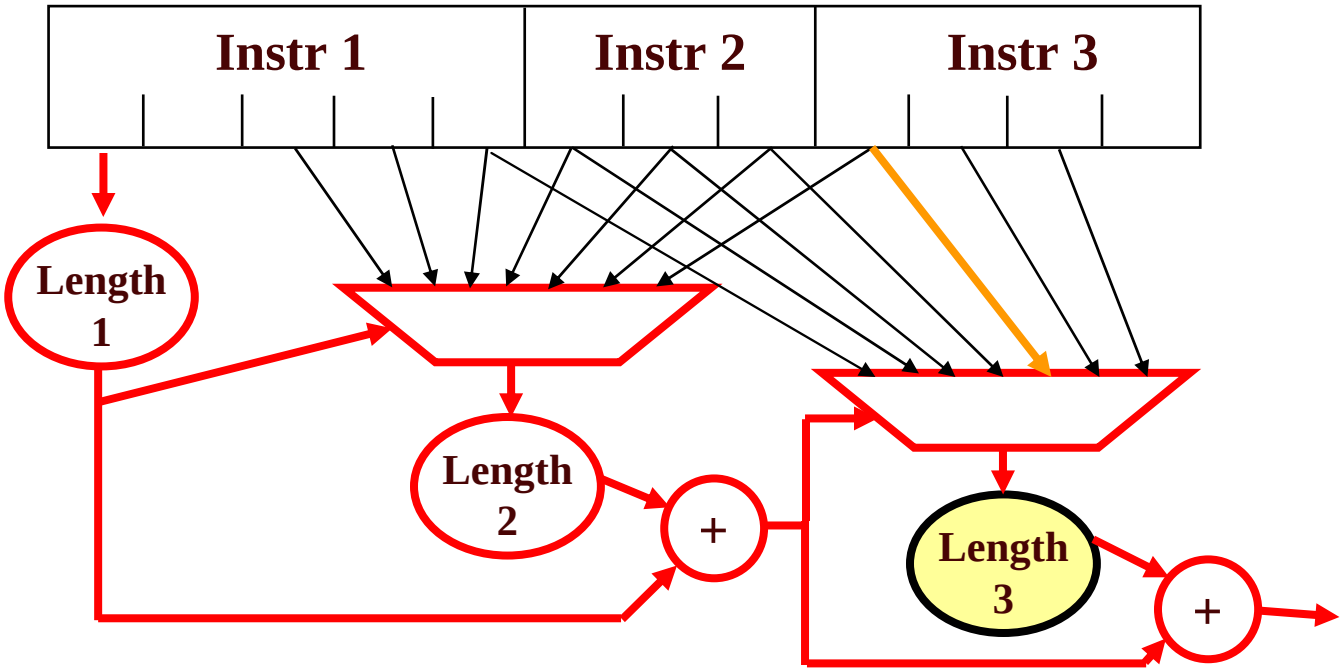
- *2nd length decoder needs to know Length1 first*

Conventional VL Length-Decoding



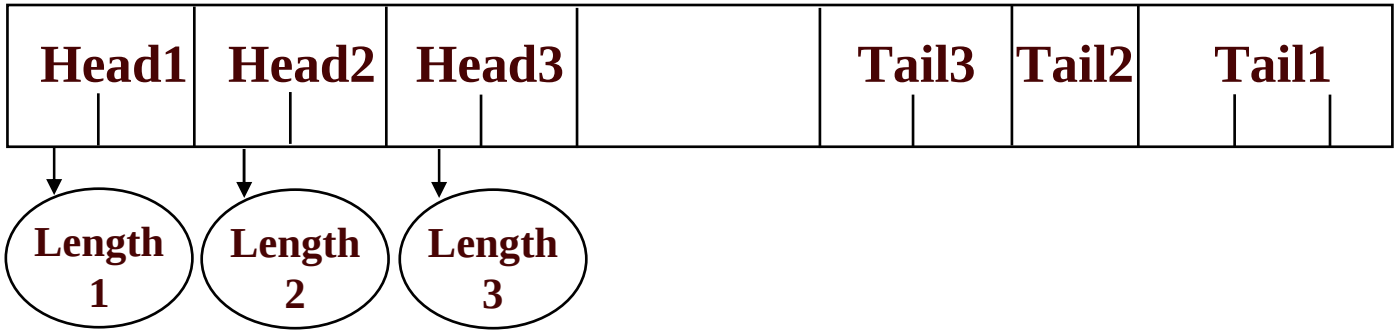
- *3rd length decoder needs to know Length1 & Length2*

Conventional VL Length-Decoding



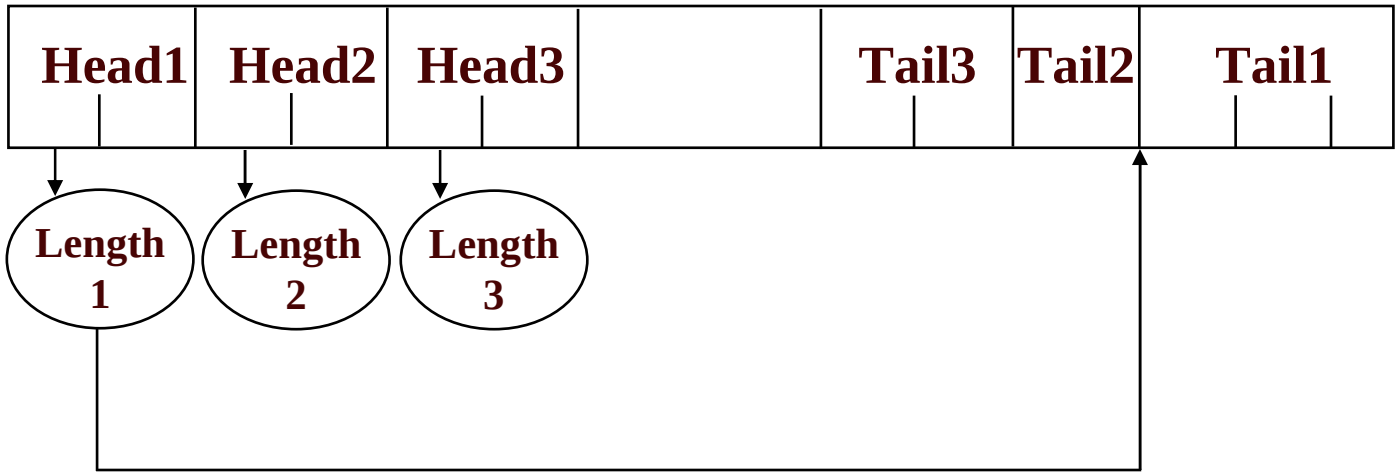
- *Need to know all 3 lengths to fetch and align more instructions.*

HAT Length-Decoding



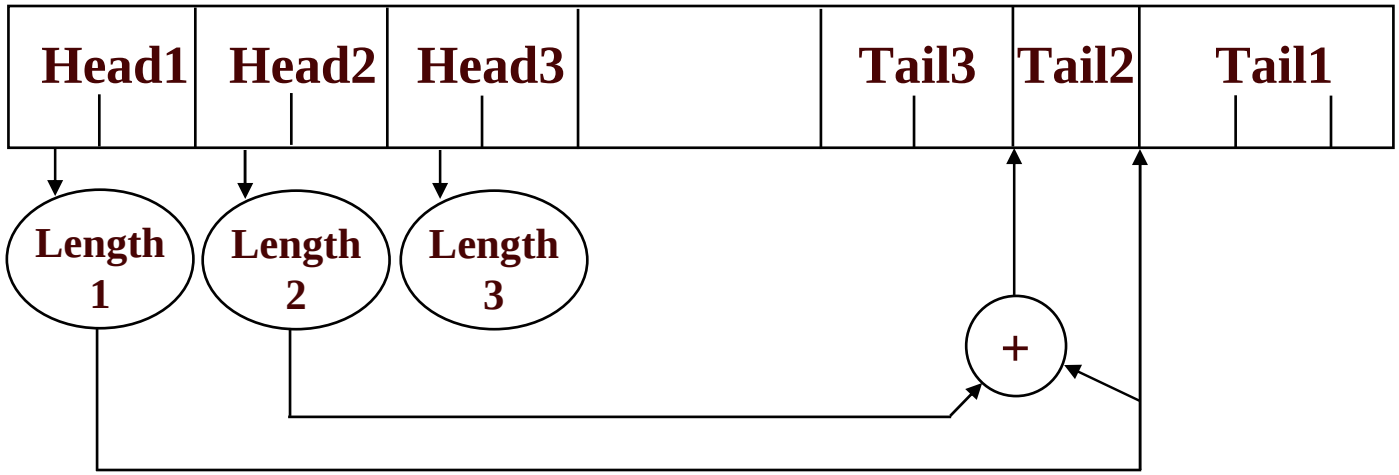
- *Length decoding done in parallel*

HAT Length-Decoding



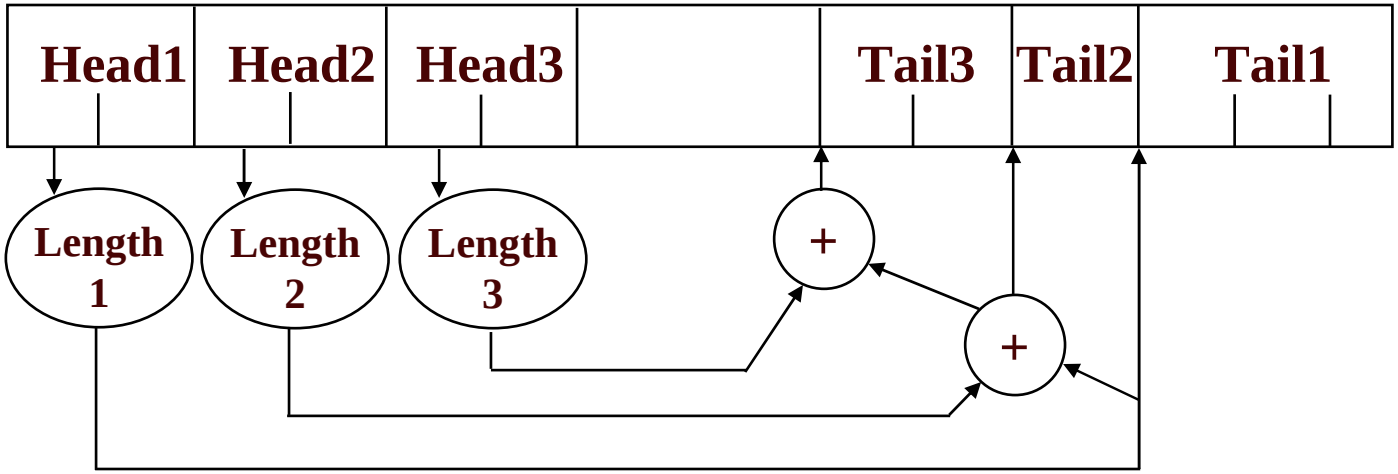
- *Length decoding done in parallel*
- *Only tail-length adders dependent on previous length information*

HAT Length-Decoding



- *Length decoding done in parallel*
- *Only tail-length adders dependent on previous length information*

HAT Length-Decoding



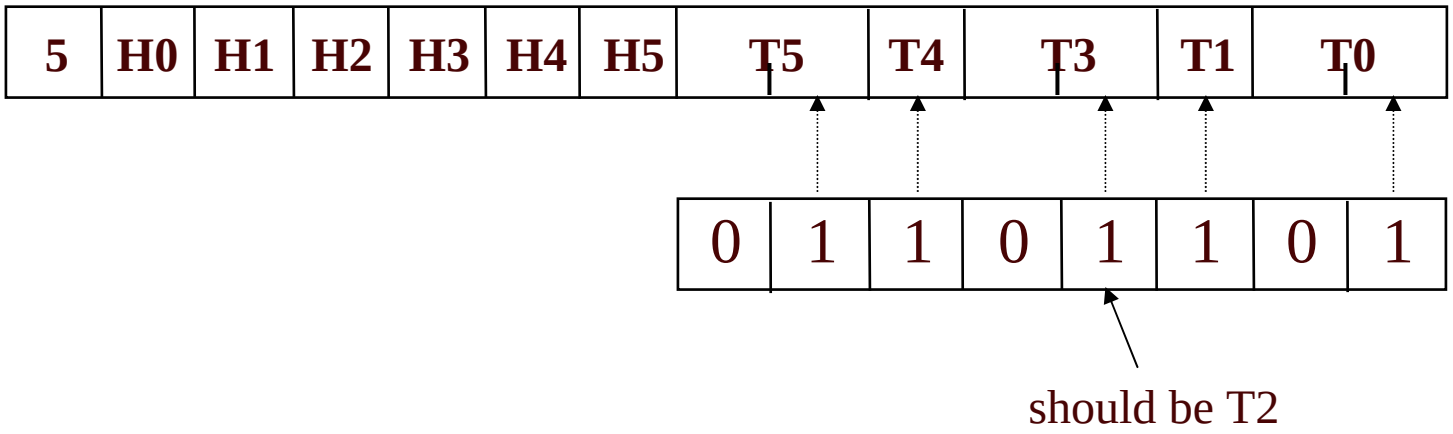
- *Length decoding done in parallel*
- *Only tail-length adders dependent on previous length information*

Branches in HAT

- When branching into middle of line, only head located, need to find tail
- Could scan all earlier heads and sum corresponding tail lengths, but substantial delay & energy penalty

Branches in HAT

- Approach 1: *Tail-Start Bit Vector*
  - Indicates starting locations of tails
  - Does not increase static code size, but increases cache area (cache refill time)
  - Requires that every head has a tail



Branches in HAT

- Approach 2: *Tail Pointers*
  - Uses extra field per head to store pointer to tail (filled in by linker at link time)
  - Removes latency, increases code size slightly
  - Cannot be used for indirect jumps (target address not known until run time)
    - Expand PCs to include tail pointer
    - Restrict indirect jumps to only be at beginning of bundle

Branches in HAT

- Approach 3: *BTB for HAT Branches*
  - Store target tail pointer info in branch target buffer
  - Resort back to scanning from the beginning of the bundle if prediction fails
  - Does not increase code size, but increases BTB size and branch mispredict penalty

HAT Advantages

- Fetch & decode of multiple variable-length instructions can be pipelined or parallelized
- PC granularity independent of instruction length granularity (less bits for branch offsets)
- Variable alignment muxes smaller than in conventional VL scheme
- No instruction straddles cache line or page boundary

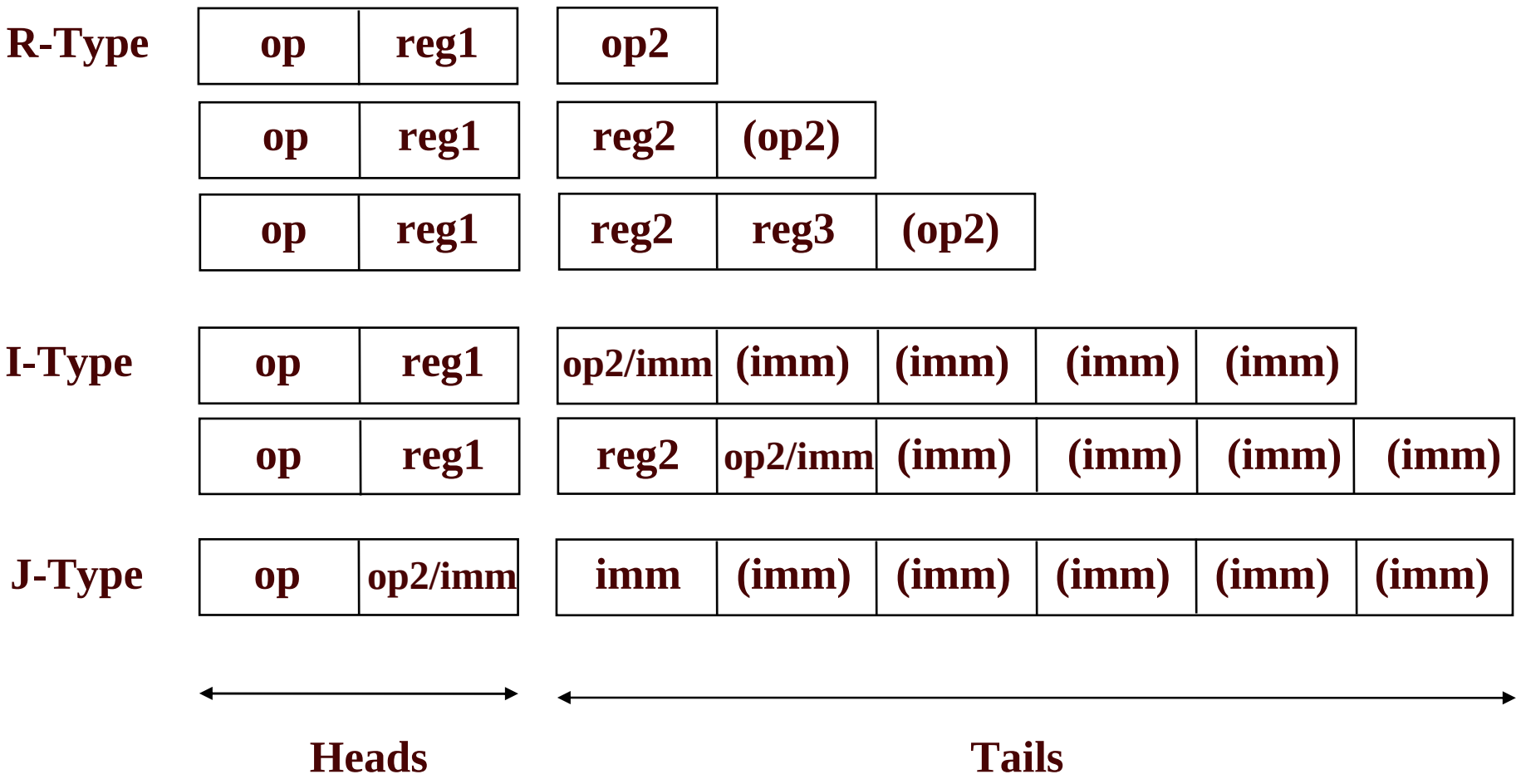
MIPS-HAT

- Example of HAT format: compressed variable-length re-encoding of MIPS
- Simple compression techniques
  - based on previous scheme by Panich99
- HAT format can be applied to many other types of instruction encoding

MIPS-HAT Design Decisions

- 5-bit tail fields (register fields not split)
- 15-40 bit instructions
  - 10-bit heads (to enable Tail-Start Bit Vector)
  - Every head has a tail

MIPS-HAT Format



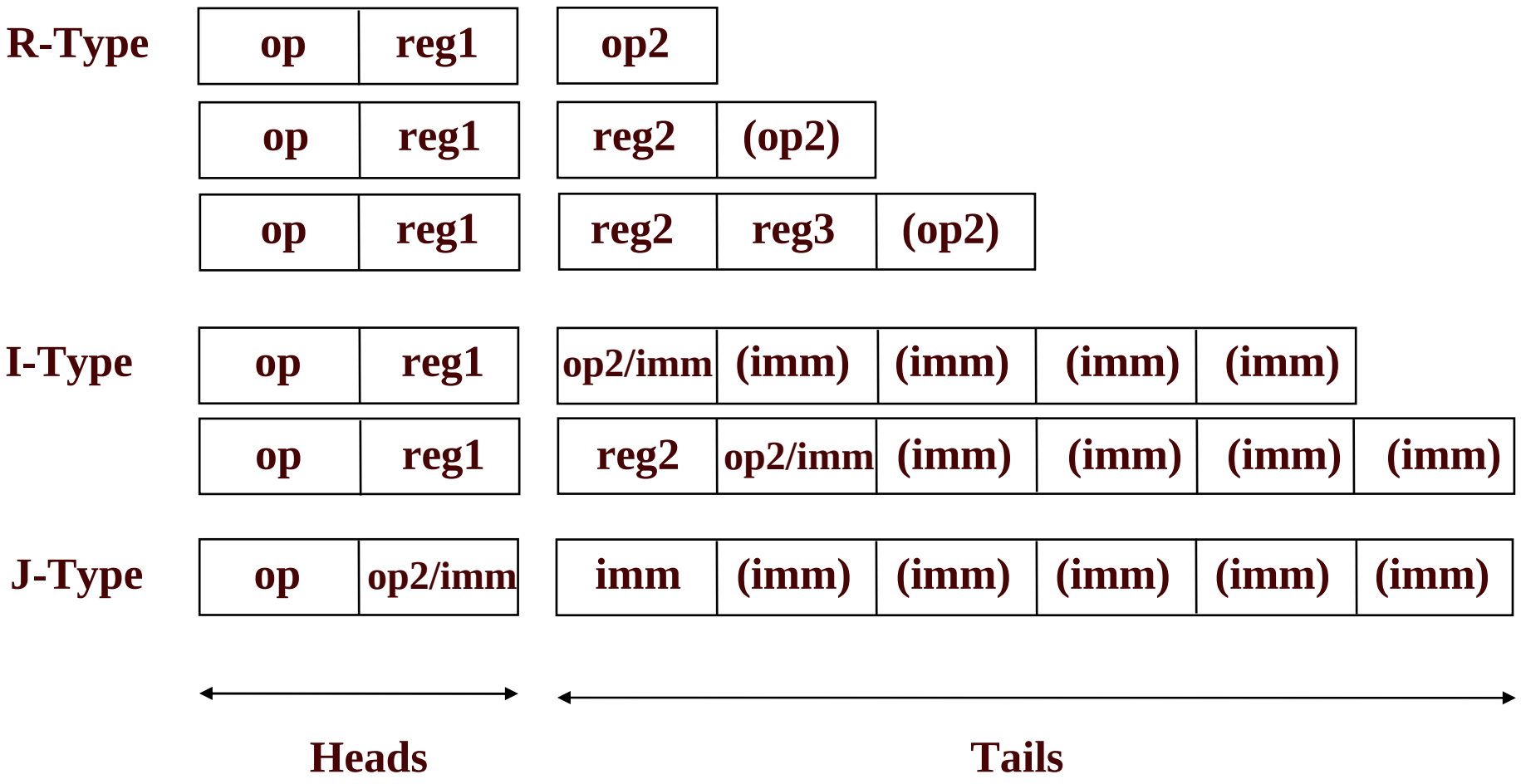
MIPS-HAT Opcodes

- Combine MIPS opcode fields
- Opcode determines length
  - 6 possible lengths; could use 3 overhead bits per instruction
  - Instead include size information in opcode but number of possible opcodes substantially increased
  - But only small subset frequently used
- Use 1-2 opcode fields
  - Most popular opcodes in primary opcode field (head)
  - All other opcodes use escape opcode and secondary opcode field (tail)

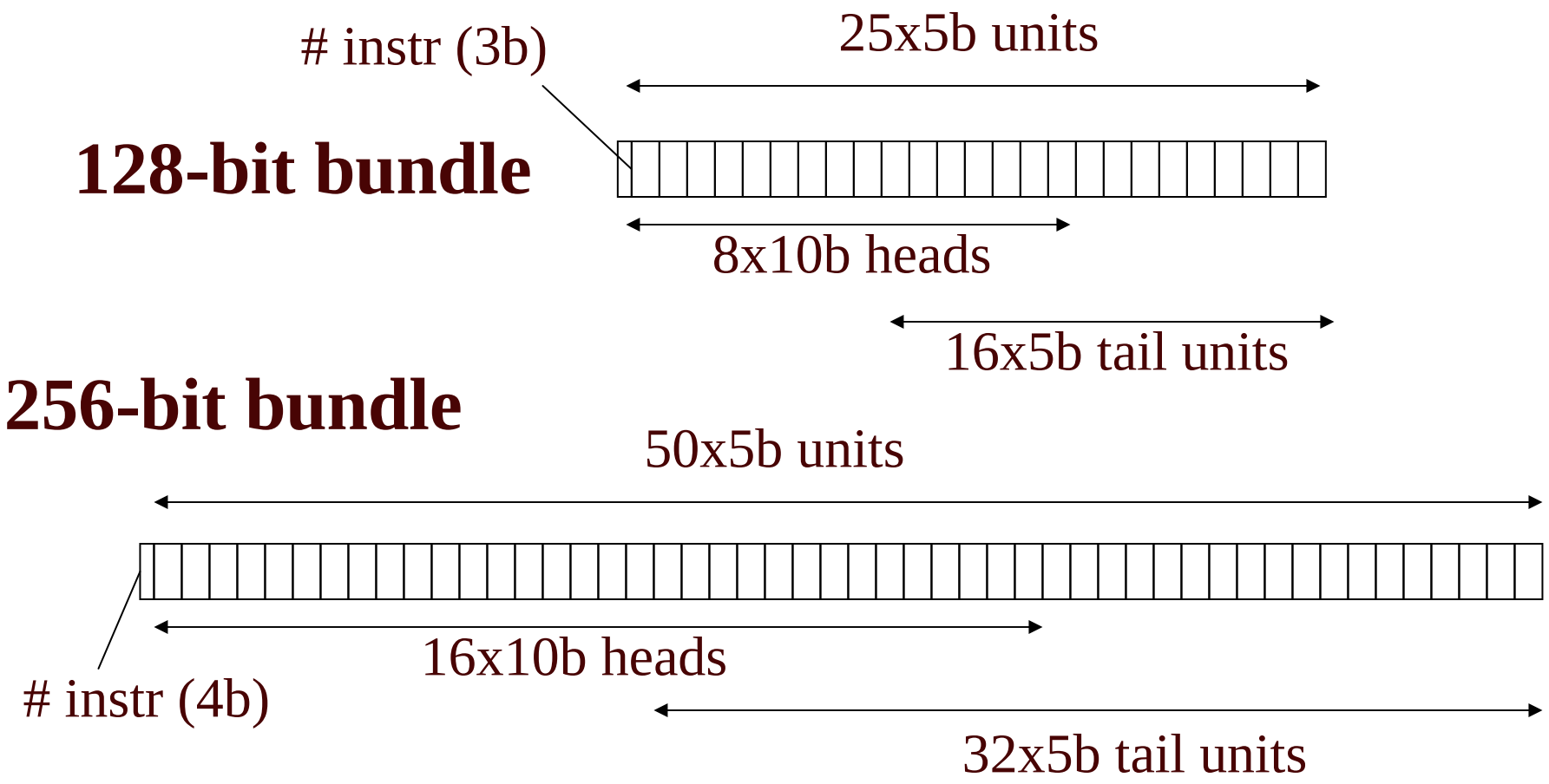
MIPS-HAT Compression

- Use the minimum number of 5-bit fields to encode immediates
- Eliminate unused operand fields
- New opcodes for frequently used operands
- Two address versions of instructions with same source & destination registers
- Common instruction sequences re-encoded as a single instruction

MIPS-HAT Format

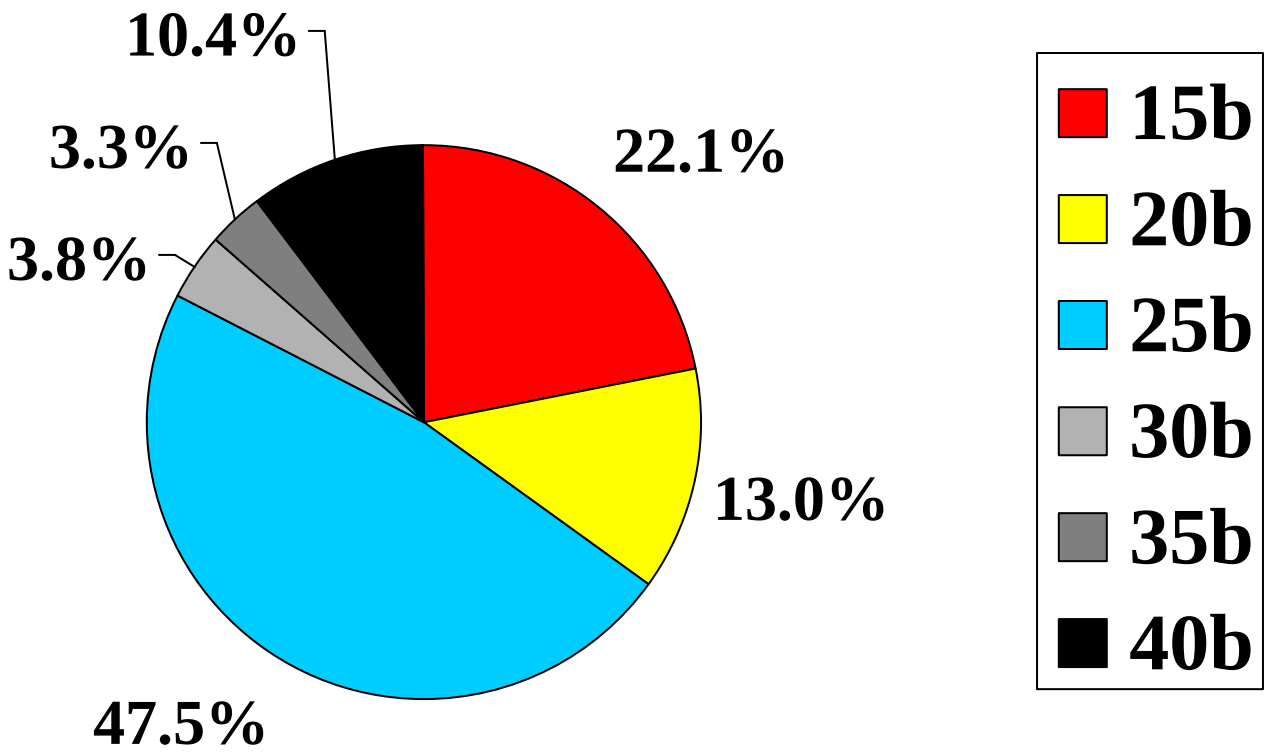


MIPS-HAT Bundle Format



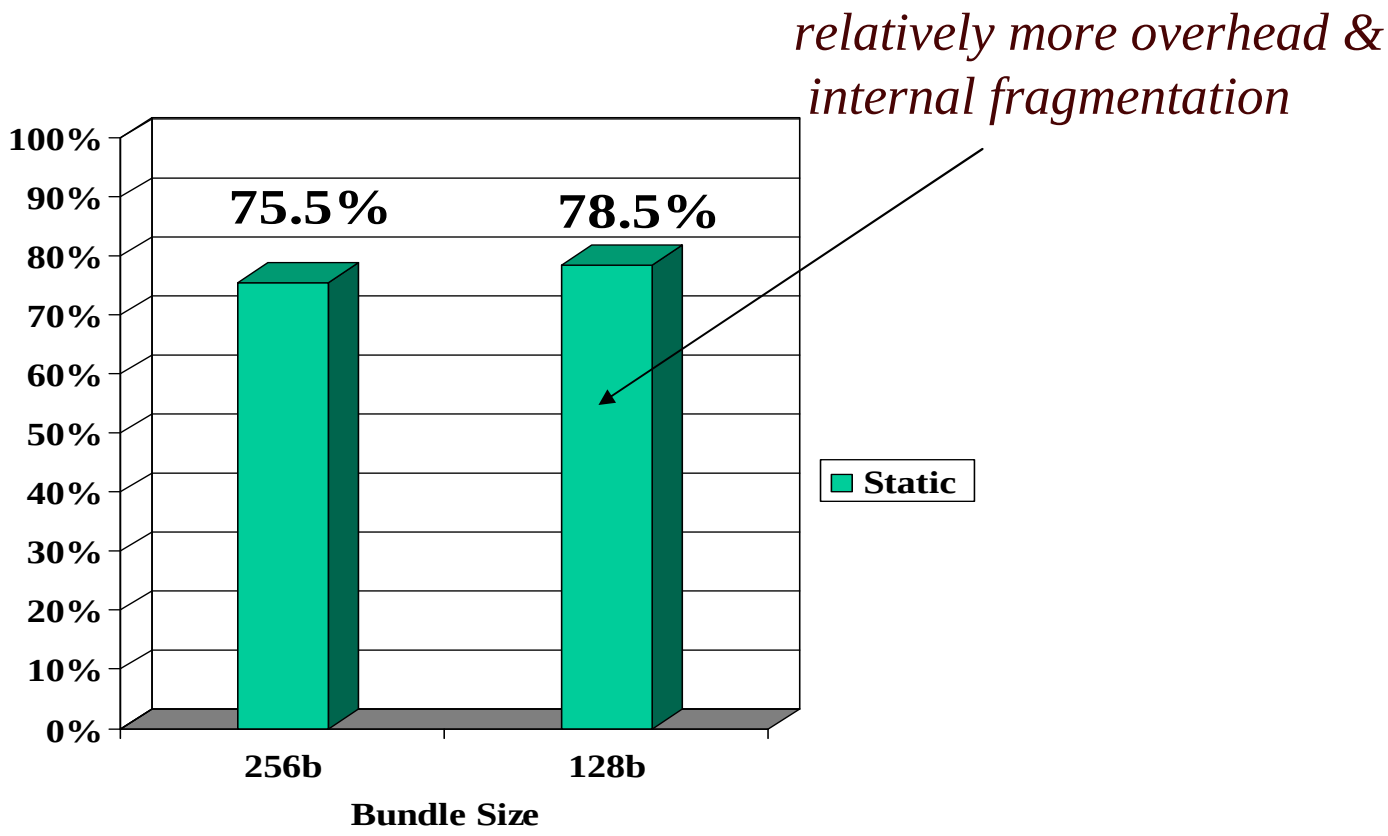
Instruction Size Distribution

- *Most instructions fit in 25 bits or less.*



Compression Ratios

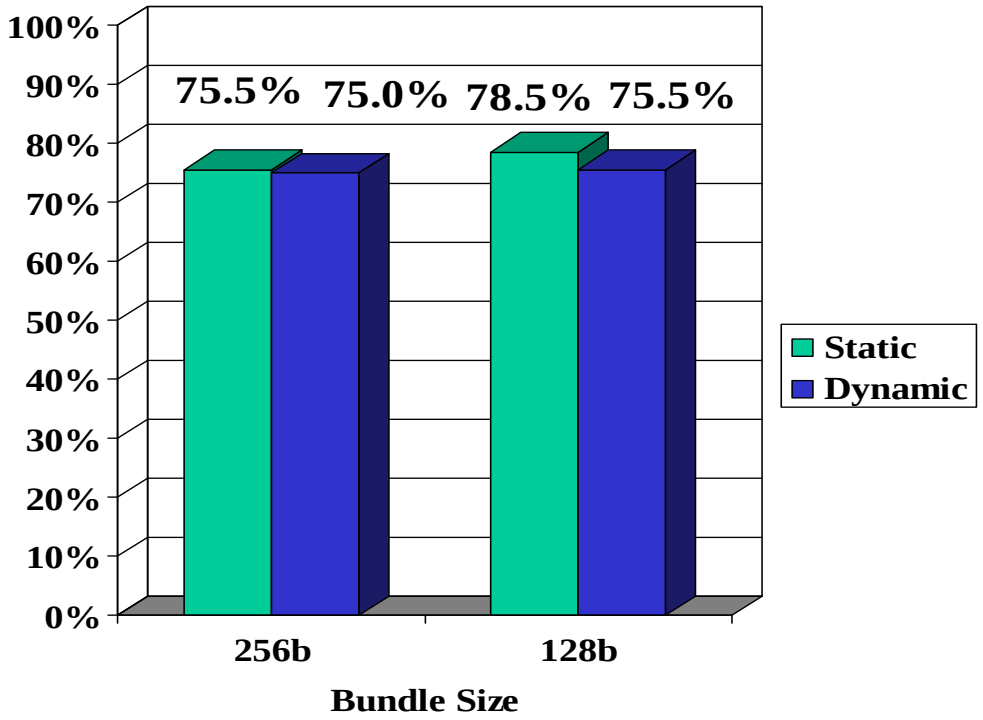
$$\textit{Static Compression Ratio} = \frac{\text{compressed code size}}{\text{original code size}}$$



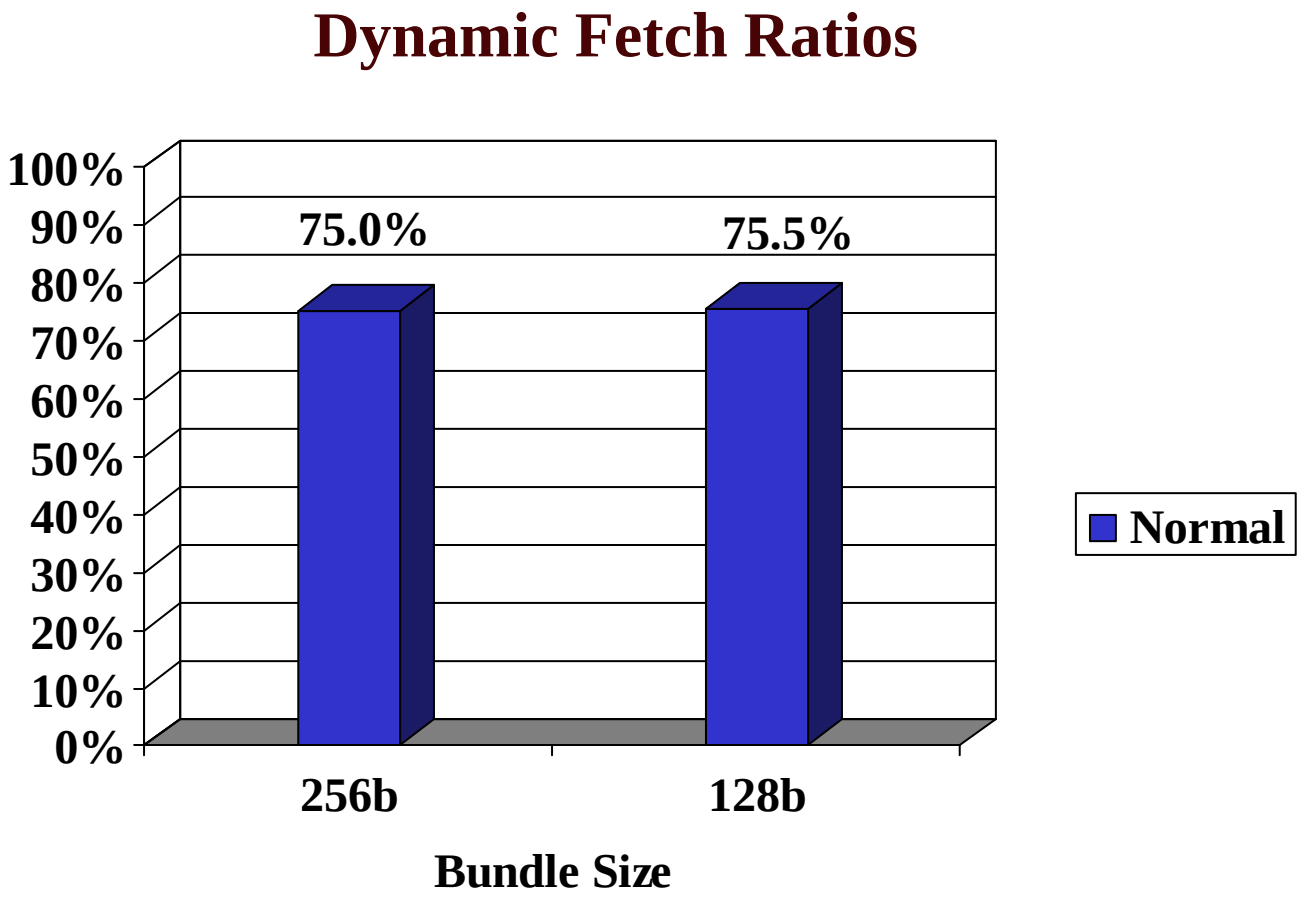
Compression Ratios

$$\textit{Static Compression Ratio} = \frac{\text{compressed code size}}{\text{original code size}}$$

$$\textit{Dynamic Fetch Ratio} = \frac{\text{new bits fetched}}{\text{original bits fetched}}$$

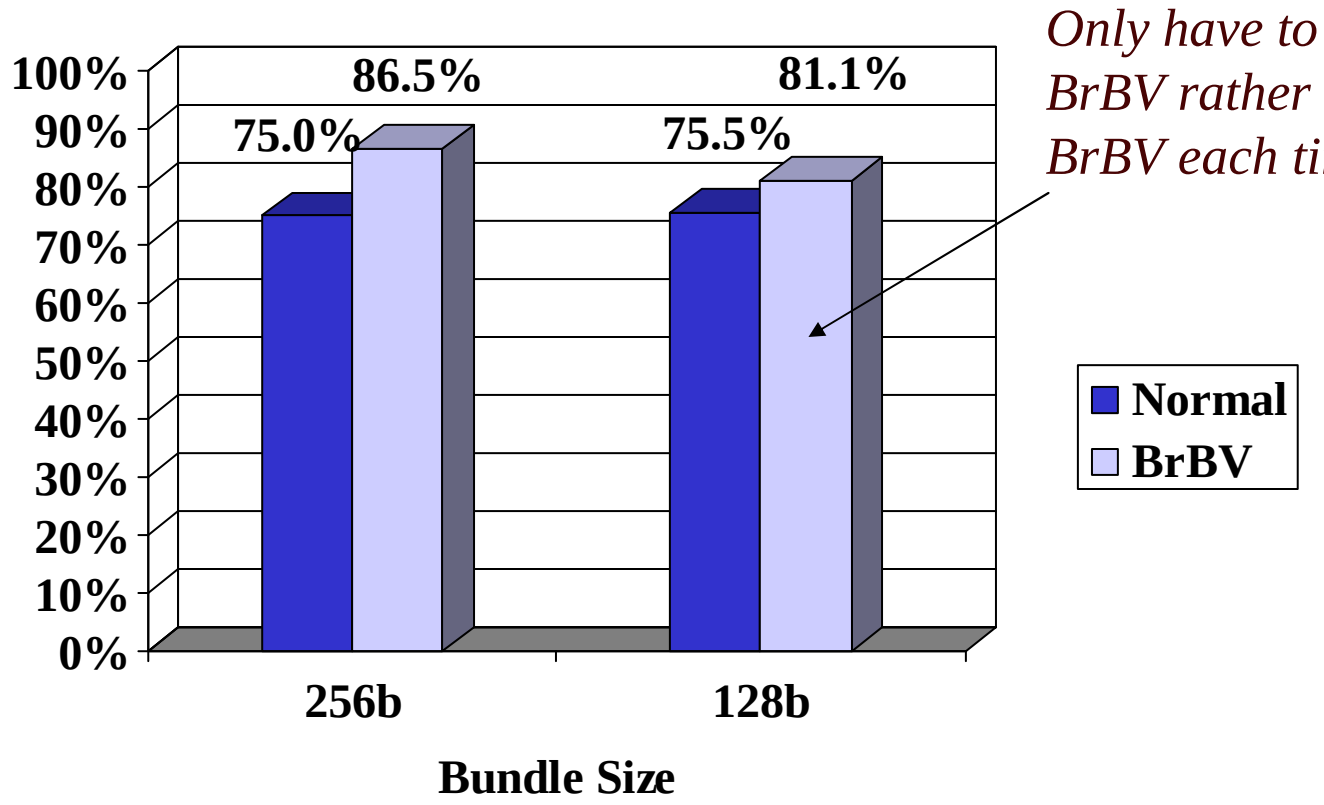


Impact of Branch Schemes on Compression



Tail-Start Bit Vector Effects

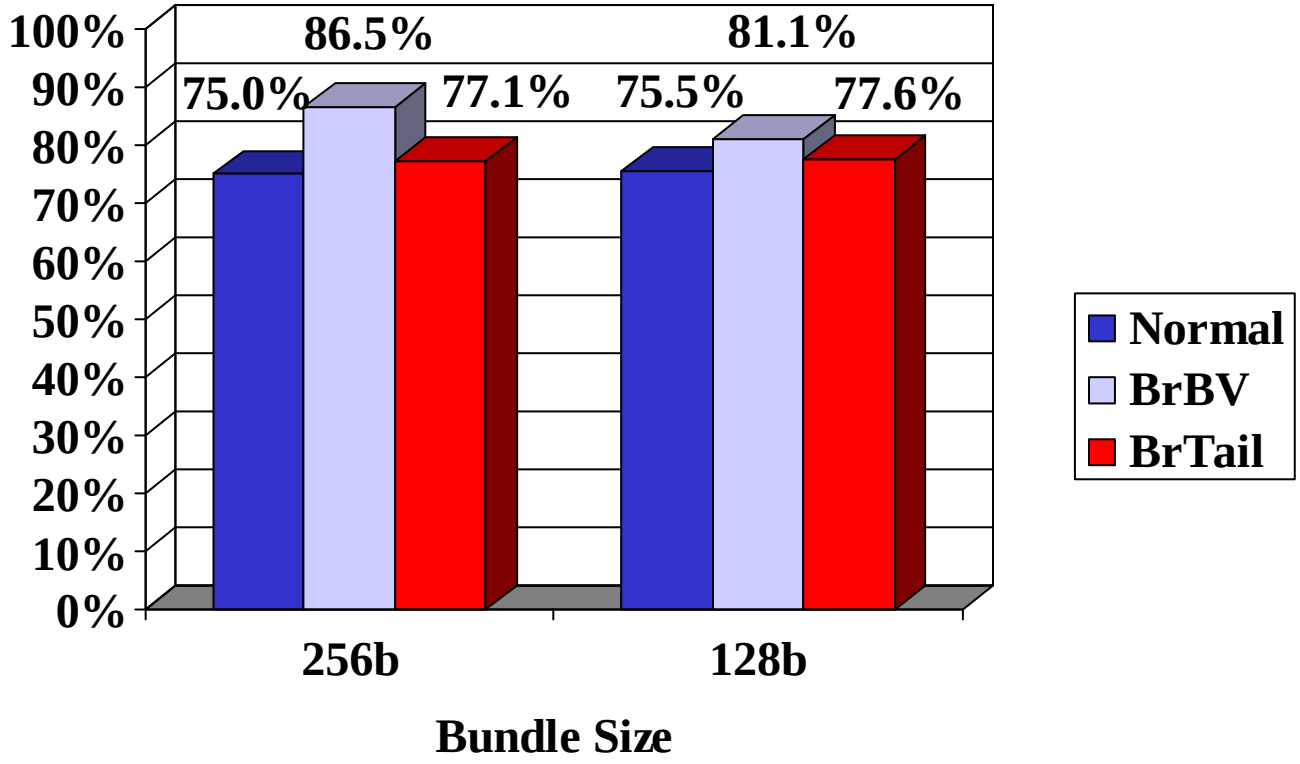
Tail-Start Bit Vector: Large increase in dynamic fetch ratio.



Only have to fetch 16b BrBV rather than 32b BrBV each time

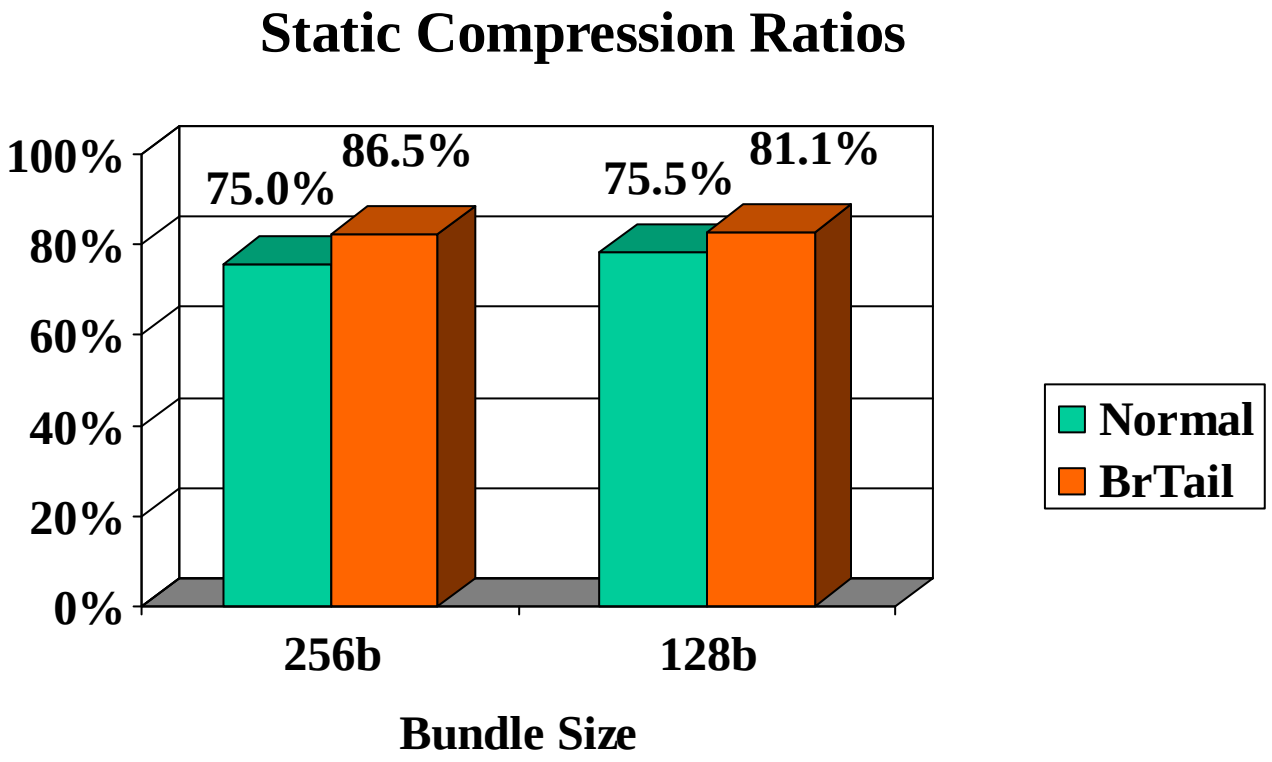
Tail Pointer Effects

Tail Pointer: Much lower cost than tail-start bit vector...

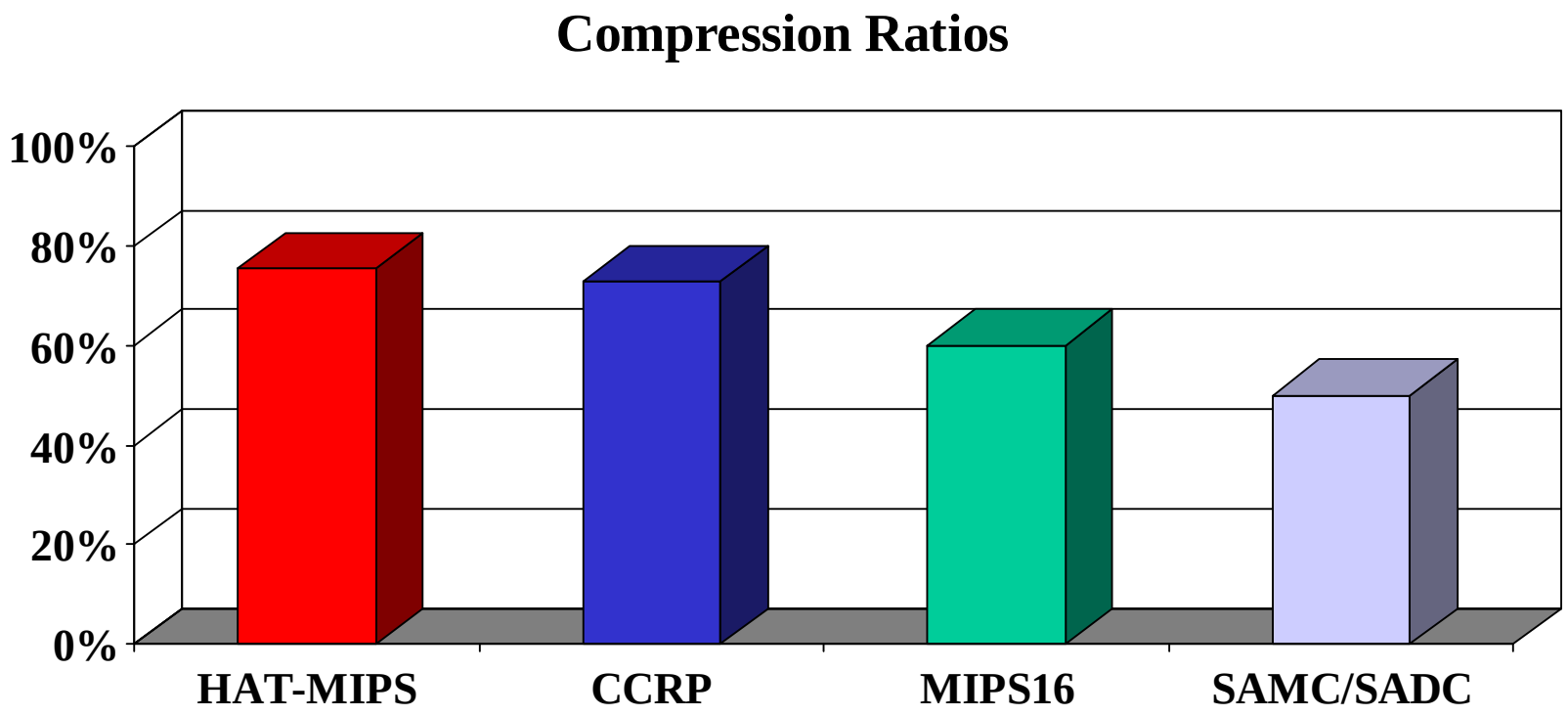


Tail Pointer Effects

Tail Pointer: But increases static code size.



Comparison to Related Schemes



Conclusion

- New heads-and-tails instruction format
  - High code density in both memory & cache
  - Allows parallel fetch & decode
- MIPS-HAT
  - Simple compression scheme to illustrate HAT
  - Static compression ratio = 75.5%
  - Dynamic fetch ratio = 75.0%
  - Several branching schemes introduced

Future Work

- HAT format can be applied to many other types of instruction encoding
  - Aggressive instruction compression techniques
  - New instruction sets that take advantage of HAT to increase performance w/o sacrificing code density