International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2001), Atlanta, GA, November 2001, to appear.

Heads and Tails: A Variable-Length Instruction Format Supporting Parallel Fetch and Decode

Heidi Pan and Krste Asanović

MIT Laboratory for Computer Science
{xoxo, krste}@lcs.mit.edu

Abstract. Existing variable-length instruction formats provide higher code densities than fixed-length formats, but are ill-suited to pipelined or parallel instruction fetch and decode. This paper presents a new variable-length instruction format that supports parallel fetch and decode of multiple instructions per cycle, allowing both high code density and rapid execution for high-performance embedded processors. In contrast to earlier schemes that store compressed variable-length instructions in main memory then expand them into fixed-length in-cache formats, the new format is suitable for direct execution from the instruction cache, thereby increasing effective cache capacity and reducing cache power. The new head-and-tails (HAT) format splits each instruction into a fixed-length head and a variable-length tail, and packs heads and tails in separate sections within a larger fixed-length instruction bundle. The heads can be easily fetched and decoded in parallel as they are a fixed distance apart in the instruction stream, while the variable-length tails provide improved code density. A conventional MIPS RISC instruction set is re-encoded in a variable-length HAT scheme, and achieves an average static code compression ratio of 75% and a dynamic fetch ratio (new-bits-fetched/old-bits-fetched) of 75%.

1 Introduction

Many embedded systems have severe cost, power consumption, and space constraints. Reducing code size is a critical factor in meeting these

constraints. Program code is often the largest consumer of memory in control-intensive applications, affecting both system cost and size. Also, instruction fetches are responsible for a significant fraction of system power and memory bandwidth.

Architects of CISC instruction sets had similar motivations for reducing program size and instruction fetch bandwidth, because early systems had small, slow magnetic core memories with no caches. These variable-length CISC instructions tend to give greater code density than fixed-length instructions. However, fixed-length RISC-style instruction sets became popular after inexpensive DRAMs reduced the cost of main memory and large semiconductor instruction caches became feasible to reduce memory bandwidth demands. Fixed-length instructions simplify high performance implementations because the address of the next instruction can be determined before decoding the current instruction (ignoring branches and other changes in control flow). Therefore, they allow fetch and decode to be easily pipelined or performed in parallel for superscalar machines.

Although embedded processors have traditionally had simple single-issue pipelines, newer designs have deeper pipelines or superscalar issue [5, 16, 19] to meet higher performance requirements. Fixed-length ISAs reduce the complexity of pipelined and superscalar fetch and decode, but incur a significant code size penalty.

In this paper, we present a new *heads-and-tails* (HAT) format, which allows compressed variable-length instructions to be held in the cache yet remain easily indexable for parallel fetch and decode. Therefore, we take advantage of the high code density of variable-length instructions while enabling deeply pipelined or superscalar machines.

The paper is structured as follows. In Section 2, we review related work in instruction compression and superscalar variable-length instruction decoding. Section 3 gives a general overview of the HAT instruction format and describes a straightforward hardware implementation. In Section 4, we present an example that packs MIPS RISC [12] instructions into the HAT format using a simple compression scheme. Using MIPS-HAT as a concrete example, we also describe more sophisticated hardware schemes that remove branch penalties. Section 5 presents results for MIPS-HAT using programs taken from the Mediabench benchmark suite. Section 6 concludes the paper.

2 Related work

The ARM Thumb [18] and MIPS16 [13] instruction sets provide alternate 16-bit versions of the base fixed-length RISC ISA (ARM and MIPS

respectively) to improve code density. Decompression is a straightforward mapping from the short instruction format to the wider instruction format in the decode stage of the pipeline. Both ISAs allow dynamic switching between full-width and half-width instruction formats at subroutine boundaries. The half-width formats reduce static code size by around 30–40%. However, since they can only encode a limited subset of operations and operand addressing modes, more dynamic instructions are required to execute a given task. The reduced fetch bandwidth can compensate for the increased instruction count when running directly from a 16-bit memory system, but for systems with an instruction cache, performance is reduced by around 20% [18]. Although they are fixed length, the reduced performance makes these short instruction formats unattractive for a superscalar implementation, as a simpler approach to boosting performance would be to revert back to the higher-performing wider format.

An alternative technique that reduces the static code size of a RISC ISA while allowing parallel fetch and decode is to hold instruction cache lines compressed in main memory but then expand them into fixed-length instruction lines when refilling the cache. This idea was introduced with the CCRP scheme [20], and a variety of similar techniques have subsequently been developed and commercialized [10, 15]. Earlier techniques developed for VLIW machines [8] only removed NOP fields within a VLIW instruction, reducing code size to about that of a RISC ISA. The processor remains unchanged with these techniques, as it sees regular easy-to-decode fixed-length instructions in the cache. Caching the uncompressed instructions avoids the additional latency and energy consumption of the decompression unit on cache hits, but decreases the effective capacity of the primary cache and increases the energy used to fetch cached instructions. Cache miss latencies increase for two reasons. First, because the processor uses regular program counter (PC) addresses to index the cache, cache miss addresses must be translated through an additional memory-resident lookup table (the Line Address Table [20]) to locate the corresponding compressed block in main memory, although a miss address translation cache can be added to reduce this penalty (the CLB in [20]). Second, the missing block is often encoded in a form that must be decompressed sequentially, increasing refill latency particularly when the requested word is not the first word in the cache line. For systems with limited memory bandwidth, however, the compressed format can actually reduce total miss latency by reducing the amount of data read from memory [20].

Dictionary-based compression schemes have also been used on instruction streams, where fixed-length code words in the instruction

stream point to a dictionary holding commonly occuring instruction sequences [2, 7, 9]. The program code is scanned to determine the commonly occuring strings, which are replaced with codewords pointing into a dictionary. Branch addresses must also be modified to point to locations in the compressed instruction stream. The dictionary is preloaded before program execution starts and forms an additional component of the process state, although it could potentially be managed as a separate cache. The main advantage of these techniques is that decompression is just a fast table lookup. On the other hand, these schemes have several disadvantages. Preloading the table before each program is executed complicates multi-programmed systems, and the table fetch adds latency into the instruction pipeline increasing branch mispredict penalties. Many dictionary schemes interleave variable length code words with uncompressed instructions, severely complicating a highly pipelined or superscalar implementation. Although it might be possible to have parallel fetch and decode from the sequences stored in the dictionary, the common strings tend to be short — often only a single instruction [2, 3, 7]. Dictionary schemes fetch full-size instructions from the dictionary RAM, which is often comparable in size to a primary instruction cache, adding additional instruction fetch energy overhead on top of the fetch of codeword bits from the primary instruction stream.

Of course, the complexity of dynamically compressing instructions can be avoided by adopting a more compact base instruction set. Legacy CISC ISAs, including VAX and x86, provide denser encoding but were intended for microcoded implementations that interpret the instruction format sequentially. Parallel fetch and decode is complicated by the need to examine multiple bytes of an instruction before the start address of the next sequential instruction is known. Nevertheless, the economic importance of legacy CISC instruction sets, such as x86, has resulted in several high-performance superscalar variable-length CISC designs [1, 4, 6, 11]. These all convert complex variable-length instructions into fixed-length RISC-like internal "micro-ops". The Intel P6 microarchitecture can decode three variable-length x86 instructions in parallel, but the second and third instructions must be simple [6]. The P6 takes a brute-force strategy by performing speculative decodes at each byte position, then muxing out the correctly decoded instructions once the lengths of the first and second instructions are determined (further described below). The AMD Athlon design predecodes instructions during cache refill to mark the boundaries between instructions and the locations of opcodes, but still requires several cycles after instruction fetch to scan and align multiple variable-length instructions

[1]. The Pentium-4 design [4] improves on the P6 family by caching decoded fixed-length micro-ops in a trace cache, but similar to the CCRP scheme, cache hits require full-size fixed-length micro-op fetches and cache misses have longer latency due to the decoding process.

These legacy CISC ISAs were not designed with parallel fetch and decode in mind. In this paper, we introduce a new heads-and-tails (HAT) format designed to support parallel fetch and decode of compact variable-length instruction sets directly from cache. The HAT format helps an implementation deliver multiple, variable-sized, randomly-accessible instruction units to the CPU in a single cycle or alternatively enables a deeply-pipelined fetch of such units. This capability can be used in several ways. The HAT format can be used to hold variable-length instructions generated by other compression schemes, or alternatively hold a new ISA developed to take advantage of the format. The example evaluated in this paper uses HAT to hold a quickly-decodable variable-length re-encoding of the MIPS instruction set.

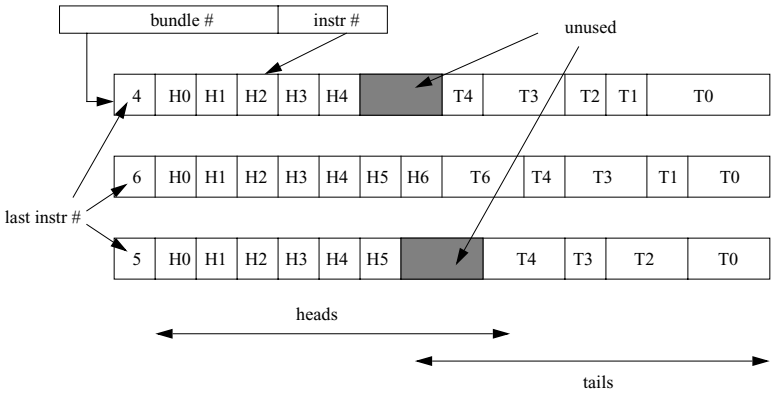

Fig. 1. Overview of heads-and-tails format.

3 Heads and tails format

The HAT format packs multiple variable-length instructions into fixed-length bundles as shown in Figure 1. The HAT format is used both in main memory and cache, although additional information might be added to the cached version to improve performance as described below. A cache line could contain one or more bundles. Bundles contain

varying numbers of instructions, so each bundle begins with a small fixed-length field holding the number of the last instruction in the bundle, i.e. a bundle holding N instructions has $N-1$ in this field. The remainder of the bundle is used to hold instructions.

Each instruction is split into a fixed-length head portion and a variable-length tail portion. The fixed-length heads are packed together in program order at the start of the bundle, while the variable-length tails are packed together in reverse program order at the end (i.e., the first tail is at the end of the bundle). Not all heads necessarily have a tail, though this can simplify some hardware implementations. The granularity of the tails is independent of the size of the heads, i.e., the heads could be 11-bits long while the tails are multiples of 5 bits, though there can be hardware advantages to making the head length a multiple of the tail granularity as discussed below. When packing compressed instructions into bundles, there can be internal fragmentation if the next instruction doesn't fit into the remaining space in a bundle, in which case the space is left empty and a new bundle is started.

The program counter (PC) in a HAT scheme is split into a bundle number held in the high bits and an instruction offset held in the low bits. During sequential execution, the PC is incremented as usual, but after fetching the last instruction in a bundle (as given by the instruction count stored in the bundle), it will skip to the next bundle by incrementing the bundle number and reseting the instruction offset to zero. All branches into a bundle have their target instruction offset field checked against the instruction count, and a PC error is generated if the offset is larger than the instruction count.

A PC value points directly to the head portion of an instruction and, because they are fixed-length, multiple sequential instruction heads can be fetched and decoded in parallel. The tails are still variable-length, however, and so the heads must contain enough information to locate the correct tail. One approach would be for each head to have a pointer to its tail, but this would usually require a large number of bits. Fewer bits are needed if the head just encodes the presence and length of a tail. This length information can often be folded into the opcode information to further reduce code size, as described below in the MIPS-HAT scheme. Similar to a conventional variable-length scheme, the tail size information in the head of one instruction must be decoded to ascertain the location of the start of the tail of the next instruction. But in the HAT format the length information for each instruction is held at a fixed spacing in the head instruction stream, independent of the length of the whole instruction. This makes the critical path to determine tail alignment for multiple parallel instructions much shorter

than in a conventional variable-length scheme, where the *location* of the length information in the next instruction depends on the length of the current instruction.

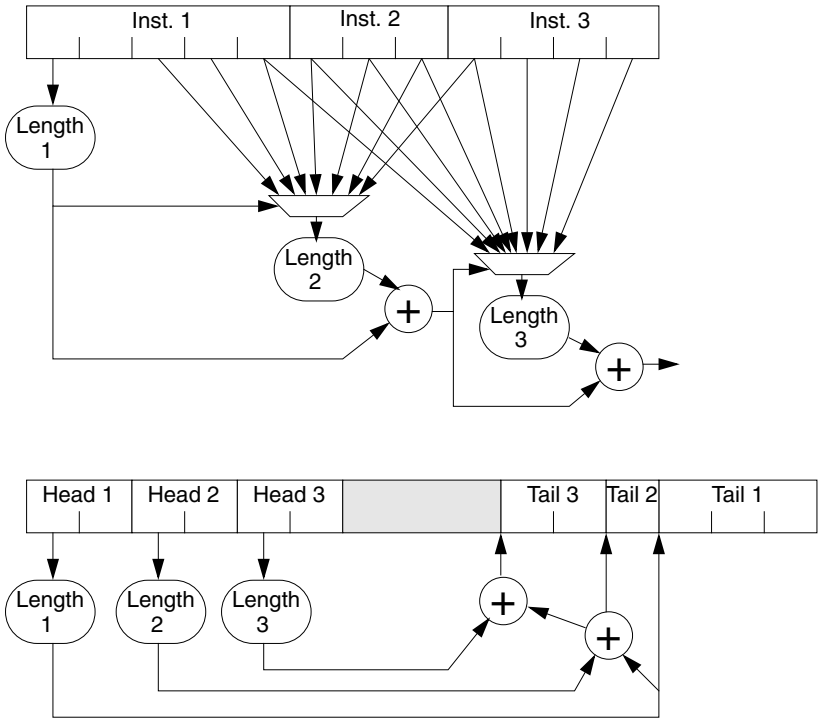

Fig. 2. Comparison of variable-length decoding in a conventional variable-length scheme and a HAT scheme.

This difference between a regular variable-length scheme and a HAT scheme is illustrated in Figure 2. The Figure shows a three-issue superscalar length decoder for a conventional variable-length ISA and a HAT ISA scheme. In both cases, instructions vary from 2–8 bytes and length information is encoded in the first byte. In the conventional scheme, the length decoder for the second instruction cannot produce a value until the first length decoder drives the mux to steer the correct byte into the second length decoder. Similarly, the third length decoder has to wait for the first two to complete before its input settles. The output of the third decoder is needed to determine the correct amount to

shift the instruction input buffer for the next cycle. This scheme scales poorly, as $O(W^2)$ area and delay for issue width W, because the number of inputs to the length byte muxes grows linearly with the number of parallel instructions. The Intel P6 family reduces this critical path by replicating simple decoders at every byte position, then muxing out the correct instructions. This requires considerable die area and additional power, and still scales as $O(W^2)$ albeit with a smaller constant for delay. In contrast, the HAT scheme operates all the length decoders in parallel, and then sums their outputs to determine tail alignments. This addition can be organized as a parallel prefix sum using a carry-save adder tree, and so delay scales logarithmically with issue width $O(\log W)$, and hardware costs grow as $O(W \log W)$.

The tails in a HAT scheme are delayed relative to the heads, but the head and tail fetches can be pipelined independently. The performance impact of the additional latency for the tails can be partly hidden if more latency-critical instruction information is located in the head portions.

3.1 Handling branches in HAT

While fetching sequentially within a bundle, the HAT instruction decoder is consuming head bits from one end of the bundle and tail bits from the other end. To avoid having to fetch and decode a new bundle before locating its first instruction's tail bits, we place tails in reverse order starting at the end of the bundle. When execution moves sequentially on to a new bundle, the initial head and tail data can be simply found at either end of the new bundle.

Branches create the biggest potential problems for the HAT scheme. Whereas a branch target address points at the entire target instruction in a conventional scheme, it only locates the head within a bundle in a HAT scheme. One simplistic approach to locate the tail of a branch target is to scan all earlier heads from the beginning of the target bundle, summing their tail lengths to get a pointer to the start of the branch target's tail. Although correct, this scheme would add a substantial delay and energy penalty to taken branch instructions. Next, we describe three different approaches to finding branch target tails in a HAT scheme: tail-start bit vectors, tail pointers, and an enhanced branch target buffer.

Tail-start bit vector We can reduce branch penalties for locating the target tail by storing auxiliary data structures in the cache alongside each bundle. These data structures do not impact static code size as

they are only present in the cache, but they increase cache area and the number of dynamic bits fetched from the cache, potentially increasing cache hit energy. The simplest scheme would be to hold a separate tail pointer for each possible instruction in a bundle, but this incurs a large overhead of $H \log(T)$ bits per bundle, where H is the maximum number of heads and T is the number of possible tail positions. A more compact approach is to store a single bit per tail position (T bits total per bundle), each bit indicating the possible start of a tail. A branch into a bundle would then read the bit vector to find the start of the N^{th} tail. This bit vector approach handles both fixed and indirect jumps, but adds some additional latency to taken branches to process the bit vector. This scheme also requires that every instruction has a tail, otherwise a second bit vector would be required to determine which instructions had tails.

Tail pointers A different approach to finding branch target tails is to change branch and jump instruction encodings to include an additional tail pointer field pointing to the tail portion of the branch target. This is filled in by the linker at link time. The tail pointer removes all latency penalties for fixed-target branch instructions, but increases code size slightly. This approach, however, cannot be used for indirect jumps where the target address is not known until run time.

There are two schemes that can be used to handle indirect jumps with tail pointers. The first scheme is to expand all PC values to contain a tail pointer in addition to the bundle and instruction offset numbers. Jump-to-subroutine instructions would then write these expanded PCs into the link register as return PC values, and jump indirect instructions would expect tail pointers in the PC values held in registers as jump targets. A minor disadvantage of this scheme is that it reduces the virtual address space available for user code by the number of bits taken for the target tail pointer ($\log(T)$ bits). Another disadvantage is that it becomes possible to branch to the middle of a tail if the user manipulates the target tail pointer directly.

The second scheme treats each type of indirect jump separately. There are three main uses of indirect jumps: indirect function calls (e.g., virtual functions in C++), switch statement tables, and subroutine returns. We can eliminate penalties on function calls and switch tables by noting that a branch to the start of a bundle can always find the tail bits of the first instruction at the end of the bundle. Therefore by simply placing function entry points and case statement entry points at the start of a bundle (which might be desirable for cache performance in any case), we eliminate branch penalties for these indirect jumps.

Subroutine returns cannot be handled as easily because the subroutine call could be anywhere within a bundle. One simple approach is to only allow instructions without tails between the subroutine call and the end of the current bundle, as a tail-less instruction does not need the tail pointer to be restored correctly after the subroutine returns. This is likely to reduce performance and waste code space, as NOPS will have to be inserted if an instruction with tail is required. Another approach is to store the return PC tail pointer on the subroutine return address stack, if the microrachitecture has one to predict subroutine returns. If the return address stack prediction fails, execution falls back to the naive algorithm that scans heads from the beginning of the target bundle.

BTB for HAT branches The third general approach to handling branches in a HAT scheme stores target tail pointer information in the branch target buffer (BTB). This can handle both fixed and indirect jumps. Again, if the prediction fails, the target bundle can be scanned from the beginning to locate a tail in the middle of the bundle. This approach does not increase static code size, but increases BTB size and branch mispredict penalty.

3.2 HAT advantages

To summarize, the HAT scheme has a number of advantages over conventional variable-length schemes.

- Fetch and decode of multiple variable-length instructions can be pipelined or parallelized.
- Unlike conventional variable-length formats, it is impossible to jump into the middle of an instruction (except if PCs are expanded to include a tail pointer field as described above).
- The PC granularity is always in units of a single instruction, and is independent of the granularity at which the instruction length can be varied. This allows branch offsets to be encoded with fewer bits than a conventional variable-length ISA, where PC granularity and instruction length granularity are identical (e.g., in bytes). This helps counteract the code size increase if tail pointers are added to branch target specifiers.
- The variable alignment muxes needed are smaller than in a conventional variable-length scheme, because they only have to align bits from the tail and not from the entire instruction length. The fixed-length heads are handled using a much simpler and faster mux.

– The HAT format guarantees that no variable-length instruction straddles a cache line or page boundary, simplifying instruction fetch and handling of page faults.

4 MIPS-HAT

In this section, we demonstrate the HAT format using a compressed variable-length re-encoding of the MIPS RISC ISA [12] as an example.

4.1 MIPS-HAT compression techniques

The MIPS compression scheme we use is based partly on a previous scheme by Panich [17]. To keep instruction decoding simple, we choose to never split MIPS register specifier fields, and so use a 5-bit granularity for our tail encoding. Our minimum size instruction is 15 bits and the maximum size is 40 bits. As discussed later in the hardware section, tail lookup can be simplified if every instruction has a tail and so we chose heads that are 10 bits long but always with a tail, giving a minimum instruction size of 15 bits. The following techniques were used to compress the MIPS instructions.

1. Use the minimum number of 5-bit fields to encode immediates.
2. Eliminate unused register and operand fields.
3. Certain instructions often use a specific value for a register or immediate, for example, the BEQ instruction often (≈90%) has zero as one operand. We provide new opcodes for these cases.
4. We provide two-address versions of instructions that frequently have a source register the same as the destination register.
5. We re-encode some common instruction sequences as a single instruction. We re-encode only the simplest but most common two types of instruction sequence: branch instructions with a NOP in the delay slot and multiple sequential loads. New opcodes for branches and jumps indicate that they are followed by a NOP. The multiple load instructions are used by subroutines to restore saved registers from consecutive offsets from the stack pointer and can be combined into a single instruction by specifying the initial register, initial offset, and the number of load instructions in the sequence. We considered a multiple store instruction but this did not provide sufficient savings to be justified (we believe this asymmetry was because the compiler often interleaves code from the start of a function with the register save code in the prologue whereas the register restore in the function epilogue is not polluted in the same way).

Each instruction can be one of six sizes, ranging from 15–40 bits. One way to specify the size would be to attach three overhead bits per instruction. However, each instruction type, e.g., ADDI (add-immediate), typically only uses a few sizes, so we fold instruction sizes into new opcodes, e.g. ADDI10b for a 10-bit add-immediate.

This substantially increases the number of possible opcodes, but only a small subset of these new opcodes is frequently used. We select the most popular opcodes, together with several different "escape" opcodes, and encode these in a 5-bit primary opcode field in the head. The escape opcodes indicate that a secondary opcode is placed in the tail, but also includes critical information required for decode, such as the size of the instruction and its general category (e.g., arithmetic versus branch). Table 1 and Table 2 show the most popular primary opcodes and escape opcodes together with the frequency that they occur across the Mediabench benchmarks. The "Break" escape opcode is used for all instructions that will cause opcode traps, including SYSCALL and BREAK.

Table 1. The 32 MIPS-HAT primary opcodes.

Instruction	Size	Freq	Instruction	Size	Freq
Specific Primary Opcodes					
addu(rt=0)	15	8.7%	lw(imm=0)	15	2.2%
sw	25	5.2%	sw	20	1.9%
lw	25	4.7%	addu	20	1.8%
addiu	25	4.5%	lw	20	1.7%
noop	15	4.3%	addiu(-1)	15	1.6%
lui	30	3.6%	jr	15	1.5%
addiu(+1)	15	3.2%	bne(rt=0)	15	1.4%
jal	25	3.2%	beq(rt=0)	15	1.3%
addu(rs=rd)	15	2.6%	addiu(rs=rd)	15	1.2%
sw(rw=r2)	20	2.6%	addiu(rs=rd)	20	1.2%
addiu	20	2.4%	addiu	30	1.1%
j	25	2.2%			
Escape Opcodes					
I-Load/Store	30	10.0%	I-Arithmetic	40	1.5%
R	25	7.2%	I-Load/store	40	0.4%
I-Branch	30	6.7%	I-Branch	40	0.0%
I-Arithmetic	30	5.4%	J	40	0.0%
Break	35	3.3%			

Table 2. MIPS-HAT primary opcodes by category.

Instruction	Size	Freq	Instruction	Size	Freq
R					
addu(rt=0)	15	8.7%	addu(rs=rd)	15	2.6%
ESC	25	7.2%	addu	20	1.8%
noop	15	4.3%	jr	15	1.5%
I-Arithmetic					
ESC	30	5.4%	addiu(-1)	15	1.6%
addiu	25	4.5%	ESC	40	1.5%
lui	30	3.6%	addiu(rs=rd)	15	1.2%
addiu(+1)	15	3.2%	addiu(rs=rd)	20	1.2%
addiu	20	2.4%	addiu	30	1.1%
I-Branch					
ESC	30	6.7%	beq(rt=0)	15	1.3%
bne(rt=0)	15	1.4%	ESC	40	0.0%
I-Load/Store					
ESC	30	10.0%	lw(imm=0)	15	2.2%
sw	25	5.2%	sw	20	1.9%
lw	25	4.7%	lw	20	1.7%
sw(rw=r2)	20	2.6%	ESC	40	0.4%
J					
jal	25	3.2%	ESC	40	0.0%
j	25	2.2%			
Break					
ESC	35	3.3%			

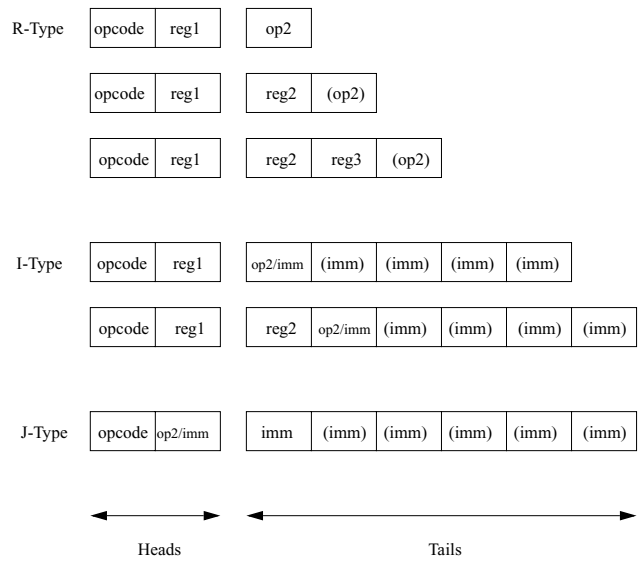

Fig. 3. Compressed MIPS instruction formats.

Figure 3 shows the formats of the three types of MIPS-HAT instructions — register (R), immediate (I), and jump (J). All fields are five bits wide. The fields in parenthesis are optional, depending on the instruction length.

4.2 Bundle format

We evaluated use of both 128-bit and 256-bit bundles for MIPS-HAT. The 128b bundle is split into a three-bit instruction count field and 25×5b units, holding up to 8×10b heads and up to 16×5b tail units. The 256b bundle has a four-bit instruction count field, two empty bits, and 50×5b units which can hold up to 16×10b heads and up to 32×5b tail units.

Note that we restrict the size of the head and tail regions to reduce the number of bits needed for the instruction count field and for the tail-start bit vector if present. Neither the head nor tail region completely spans the bundle, although the boundary between the regions is flexible. In practice, it is rare for bundle packing to be affected by this restriction.

4.3 HAT cache implementation

MIPS-HAT is designed to be directly executed from cache, and instructions remain in the same format after being fetched from memory to cache, avoiding additional cache miss latencies. The new format is only slightly more complex than regular MIPS to decode, and the decompression is just folded into the decoder.

A conventional variable-length ISA would fetch words of data sequentially from the cache into fetch buffers that can rotate the data to the correct alignment for the instruction decoder. MIPS-HAT would use the same scheme for the tails, but in addition would be fetching a second stream for the fixed-length heads which would not require an alignment circuit. The cache RAM does not require a second read port to provide the head data stream, as the heads are always from the same bundle as the tails and hence would be on the same cache line. The cache RAM sense-amps just need a separate set of bus drivers onto the head data bus.

Table 3. Static Compression Ratios

Input	128b	256b	128b BrTail	256b BrTail
adpcm-dec	78.5%	75.5%	82.6%	82.1%
adpcm-enc	78.6%	75.6%	82.6%	82.0%
epic-dec	77.1%	74.0%	80.4%	79.5%
epic-enc	78.7%	75.5%	81.6%	80.8%
g721-dec	78.0%	75.0%	82.3%	81.6%
g721-enc	78.0%	75.0%	82.3%	81.6%
gsm-dec	79.8%	76.8%	85.2%	84.4%
gsm-enc	79.8%	76.8%	85.2%	84.4%
jpeg-dec	74.3%	71.5%	78.4%	77.5%
jpeg-enc	74.2%	71.5%	78.7%	77.9%
mpeg2-dec	80.6%	77.9%	85.5%	85.2%
mpeg2-enc	81.6%	79.1%	86.4%	86.4%
pegwit-dec	80.0%	76.6%	84.4%	84.4%
pegwit-enc	80.0%	76.6%	84.4%	84.4%
Average	78.5%	75.5%	82.8%	82.3%

Because head information is needed to extract the tails, the tail instruction bits always lag the heads. To reduce the impact of this additional latency on the execution pipeline, MIPS-HAT places the instruction category in the head so that the instruction can be steered

Table 4. Instruction Size Distribution

	15b	20b	25b	30b	35b	40b
Average (w/o BrTail)	22.1%	13.0%	47.5%	3.8%	3.3%	10.4%
Cumulative	22.1%	35.1%	82.6%	86.4%	89.6%	100.0%
Average (w/ BrTail)	19.8%	16.1%	35.1%	17.3%	3.3%	8.4%
Cumulative	19.8%	35.9%	70.9%	88.2%	91.6%	100.0%

to an appropriate functional unit before the tail arrives, allowing the tail to be sent directly to the appropriate unit for further decoding.

5 Experimental results

To test the effectiveness of the MIPS-HAT scheme, we selected benchmarks from the Mediabench [14] benchmark suite, reencoded the MIPS binaries generated by a `gcc` cross-compiler (`egcs-1.0.3a -O2`), and took static and dynamic measurements. For the dynamic measurements, the Mediabench programs were run to completion on the provided input sets.

5.1 Static compression ratios

Table 3 gives the static compression ratios (compressed-size/original-size) for 128b and 256b versions of MIPS-HAT. The bundle ratios for the two sizes includes the overhead bits to count the instructions in each bundle and any wasted space due to fragmentation.

The average bundle compression ratio is 78.5% for the 128b bundle and 75.5% for the 256b bundle. The smaller bundle incurs relatively more overhead and has more internal fragmentation. If we adopt the scheme that adds target tail links to speed taken branches, the static code size increases, to a compression ratio of 82.8% for 128b bundles and 82.3% for the 256b bundles.

Table 4 shows the distribution of static instruction sizes averaged over the benchmark set, with and without the tail pointer scheme. Without target tails, over 80% of instructions are 25 bits or less.

5.2 Dynamic measures

We measured the reduction in dynamic bits fetched from the instruction cache using the MIPS-HAT scheme. We report this number as a dynamic fetch ratio

(new-bits-fetched/original-bits-fetched). We evaluated several different schemes to avoid taken branch penalties

Tables 5 and 6 show the dynamic fetch ratios for 128b and 256b bundles, respectively, for a variety of implementations. The baseline column shows the ratios including the cost of fetching the instruction count on every access to a new bundle. The 256b scheme has a slightly lower fetch ratio (75.0% versus 75.5%) as relatively fewer overhead bits are fetched.

The BrBV column shows the large increase in dynamic fetch ratio when a tail-start bit vector (Section 3.1) is used to reduce branch taken penalties. The increase is less for the 128b bundles which have a 16b vector per line, such that these now have lower fetch ratios than 256b bundles, which must fetch a 32b vector on every taken branch.

The BrTail columns shows the fetch ratio for the tail pointer scheme, where branch instruction encodings include a tail pointer. These fetch ratios are much lower than for the BrBV approach, but this technique has a higher static code size.

Table 5. Dynamic Compression Ratios - 128b

Input	Line Ratio	BrBV	BrTail
adpcm-dec	72.0%	79.8%	75.0%
adpcm-enc	74.5%	84.0%	76.9%
epic-dec	75.2%	83.4%	77.7%
epic-enc	85.5%	89.3%	87.8%
g721-dec	75.3%	82.2%	78.4%
g721-enc	75.3%	82.2%	78.5%
gsm-dec	75.5%	79.6%	76.0%
gsm-enc	72.0%	74.1%	74.5%
jpeg-dec	68.2%	71.0%	69.1%
jpeg-enc	72.9%	79.9%	73.9%
mpeg2-dec	80.1%	85.3%	82.0%
mpeg2-enc	74.0%	79.1%	75.7%
pegwit-dec	79.1%	83.2%	80.8%
pegwit-enc	78.0%	82.3%	79.8%
average	75.5%	81.1%	77.6%

A BTB aproach to locating target tails would add nothing to the static code size, and would have a dynamic fetch ratio similar to the BrTail scheme, except now some of these bits would be fetched from

Table 6. Dynamic Compression Ratios - 256b

Input	Line Ratio	BrBV	BrTail
adpcm-dec	71.2%	86.9%	74.5%
adpcm-enc	73.5%	92.5%	76.4%
epic-dec	74.5%	91.0%	77.3%
epic-enc	85.1%	92.8%	87.1%
g721-dec	75.0%	88.9%	78.5%
g721-enc	73.8%	87.7%	78.4%
gsm-dec	74.8%	83.1%	77.5%
gsm-enc	71.3%	75.5%	72.2%
jpeg-dec	67.5%	80.7%	68.8%
jpeg-enc	72.4%	86.3%	75.3%
mpeg2-dec	79.7%	90.1%	79.7%
mpeg2-enc	76.1%	83.8%	75.1%
pegwit-dec	78.2%	86.5%	79.9%
pegwit-enc	77.1%	85.8%	78.8%
average	75.0%	86.5%	77.1%

the BTB structure. The BTB scheme will also incur additional latency penalties on BTB mispredicts.

5.3 Results discussion

The numbers show that there are tradeoffs between static code size, dynamic fetch ratio, and taken branch performance, depending on the bundle size and the branch penalty avoidance scheme. The larger bundle generally gives the best reduction in code size and bits fetched. Our dynamic results did not measure the expected increase in performance due to the effective increase in cache capacity, which should lower miss rates.

Other work has presented compression numbers for MIPS code. CCRP [20] achieves a compression ratio of 73% but has to uncompress instructions into cache to allow parallel fetch and decode. MIPS16 [13] obtains a compression ratio of around 60%, but at the cost of limiting operations and operand addressing modes which reduces performance. SAMC and SADC [15] use more complex algorithms to achieve a compression ratio nearly 50% on MIPS code but either with a long decoding delay or an added dictionary lookup step.

6 Conclusions

We have introduced a new head-and-tails (HAT) variable-length instruction format that separates instructions into fixed-length heads that can be easily indexed and variable-length tails that provide code compression. The format can provide high code density in memory and in cache, while allowing parallel fetch and decode for direct superscalar execution from cache. The HAT scheme makes it difficult to quickly locate an entire instruction at a branch target. A number of techniques are possible to reduce taken branch penalties, and these were shown to have differing effects on static code size, dynamic bits fetched, and branch penalties.

We developed a simple MIPS instruction compression scheme by re-encoding the MIPS ISA into a variable-length format, and mapping the resulting variable-length instructions into the HAT format. Our experiments showed that the MIPS-HAT format can provide a compression ratio of 75.5% and a dynamic fetch ratio reduction of 75.0% while supporting deeply pipelined or superscalar execution.

The HAT format can be applied to many other types of instruction encoding. For example, each instruction slot in a VLIW instruction could be encoded in a similar way as MIPS-HAT to give similar savings (over and above simple NOP compression). In future work, we are also investigating more aggressive instruction compression techniques tuned for the HAT format, as well as developing new instruction sets that take advantage of the HAT format to increase performance without sacrificing code density.

7 Acknowledgements

This work was funded by DARPA PAC/C award F30602-00-2-0562 and by NSF CAREER award CCR-0093354.

References

1. *AMD Athlon Processor x86 Code Optimization*, chapter Appendix A: AMD Athlon Processor Microarchitecture. AMD Inc., 220071-0 edition, September 2000.
2. C. Lefurgy et al. Improving code density using compression techniques. In *MICRO-30*, pages 194–203, Research Triangle Park, North Carolina, December 1997.
3. G. Araujo et al. Code compression based on operand factorization. In *MICRO-31*, pages 194–201, December 1998.

4. G. Hinton et al. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, Q1 2001.
5. J. Choquette et al. High-performance RISC microprocessors. *IEEE Micro*, 19(4):48–55, July/August 1999.
6. J. Circello et al. The superscalar architecture of the MC68060. *IEEE Micro*, 15(2):10–21, April 1995.
7. L. Benini et al. Selective instruction compression for memory energy reduction in embedded systems. In *ISLPED*, pages 206–211, August 1999.
8. R. P. Colwell et al. A VLIW architecture for a trace scheduling compiler. *IEEE Trans. Computers*, 37(8):967–979, August 1988.
9. S. Liao et al. Code optimization techniques for embedded dsp microprocessors. In *DAC*, 1995.
10. T. M. Kemp et al. A decompression core for PowerPC. *IBM J. Res. & Dev.*, 42(6):807–812, November 1998.
11. L. Gwennap. Intel's P6 uses decoupled superscalar design. *Microprocessor Report*, 9(2):9–15, February 1995.
12. G. Kane. *MIPS RISC Architecture (R2000/R3000)*. Prentice Hall, 1989.
13. Kevin D. Kissell. MIPS16: High-density MIPS for the embedded market. In *Proceedings RTS97*, 1997.
14. C. Lee, M. Potkanjak, and W. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communication systems. In *Micro-30*, pages 330–335, December 1997.
15. H. Lekatsas and W. Wolf. Code compression for embedded systems. In *DAC*, pages 516–521, San Francisco, CA, June 1998.
16. IBM Microelectronics. PowerPC 440GP embedded processor: High performance SOP for networked applications. Presentation from Embedded Processor Forum, June 2000.
17. M. Panich. Reducing instruction cache energy using gated wordlines. Master's thesis, Massachusetts Institute of Technology, August 1999.
18. S. Segars, K. Clarke, and L. Goudge. Embedded control problems, Thumb, and the ARMT7TDMI. *IEEE Micro*, 15(5):22–30, October 1995.
19. SiByte, Inc. SB-1 CPU fact sheet. at www.sibyte.com, October 2000. rev. 0.1.
20. A. Wolfe and A. Chanin. Executing compressed programs on an embedded RISC architecture. In *MICRO 25*, pages 81–91, Portland, Oregon, December 1992.