High Performance, Variable-Length Instruction Encodings

by

Heidi Pan

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2002

Author . . . . . .
Department of Electrical Engineering and Computer Science
May 24, 2002

Certified by . . . . . .
Krste Asanović
Assistant Professor
Thesis Supervisor

Accepted by . . . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

High Performance, Variable-Length Instruction Encodings

by

Heidi Pan

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2002, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Minimizing program code size reduces power consumption and space, which is especially important in embedded systems. 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 thesis 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 processors. The new heads-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. Compared to earlier schemes that expand compressed formats on cache refills, the new format is suitable for direct execution from the instruction cache, thereby increasing effective cache capacity and reducing cache power. Various implementations of the HAT format have been evaluated on re-encoded RISC and VLIW instruction sets, yielding compression ratios between 60% and 75% using only simple statistical compression techniques.

Thesis Supervisor: Krste Asanović
Title: Assistant Professor

Acknowledgments

First, I would like to thank Krste for being such a knowledgeable and supportive advisor. I would also like to thank the SCALE group for their help and support. Lastly, I would like to thank my parents for their love and unwavering encouragement for me to pursue graduate school.

Contents

1 Introduction **9**

2 Related Work **11**

2.1 Compression of RISC ISAs 11

2.2 Compression of VLIW ISAs 13

2.3 CISC ISAs 14

3 Heads and Tails Format **15**

3.1 Comparison with Conventional Variable-Length Schemes 17

3.2 Handling Branches in HAT 19

3.3 Two-Level Extension of Heads and Tails: VLIW-HAT 21

4 MIPS-HAT **25**

4.1 MIPS Instruction Set Overview 25

4.2 MIPS-HAT Compression Techniques 27

4.3 Bundle Format 31

4.4 HAT Cache Implementation 31

4.5 Experimental Results 32

4.5.1 Static Compression Ratios 32

4.5.2 Dynamic Measures 32

4.5.3 Results Discussion 34

5 IA64-HAT **37**

5.1 IA-64 Instruction Set Overview 37

5.2 IA64-HAT Compression Techniques 38
5.3 Experimental Results 40
5.3.1 Static Compression Ratios 42
5.3.2 Dynamic Measures 44
5.3.3 Compression Discussion 46
5.3.4 Performance Discussion 46

6 Conclusion **49**

Chapter 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.

Existing instruction set architectures generally fall into three categories: Complex Instruction Set Computer (CISC), Reduced Instruction Set Computer (RISC), and Very Long Instruction Word (VLIW). CISC instruction sets were designed with similar motivations as embedded systems to reduce 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. On the other hand, VLIW architectures have recently become popular in high-performance and embedded applications. By shifting the responsibility for detecting instruction-level parallellism to the compiler, complex runtime control circuitry is

avoided in VLIW schemes, allowing simple implementations to achieve high performance. The fixed-length instructions and instruction bundles also have the same high-performance benefits as fixed-length RISC instructions. However, VLIW architectures have a significantly larger code size than both RISC or CISC designs.

Although embedded processors have traditionally had simple single-issue pipelines, newer designs have deeper pipelines or superscalar issue [6, 23, 26] 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.

This work presents 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 high code density of variable-length instructions while enabling deeply pipelined, superscalar, or VLIW machines.

The thesis is structured as follows. Chapter 2 reviews related work in instruction compression and variable-length instruction encoding. The general overview of the HAT instruction format is described in Chapter 3. Chapter 4 and Chapter 5 give two examples that pack MIPS [19] RISC instructions and IA-64 [17] VLIW instructions, respectively, into a HAT format using a simple compression scheme. Chapter 6 concludes the thesis.

Chapter 2

Related Work

Two main approaches to achieving both high performance and high code density are to compress a fixed-length instruction set and to use a variable-length instruction set. Compression reduces the code size using compressed instructions while enabling the high performance of executing uncompressed fixed-length instructions. Variable-length instruction sets provide denser code without the complexity of compression and decompression, but are ill-suited for pipelined or superscalar execution.

2.1 Compression of RISC ISAs

Most of the previous work in code compression has focused on RISC architectures. The ARM Thumb [25] and MIPS16 [20] 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%, but can only encode a limited subset of operations and operand addressing modes and so require more dynamic instructions 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% [25]. 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.

Dictionary-based compression is an alternative approach to reducing code size [3, 8, 11]. Common occurring strings in the instruction stream are replaced by fixed-length code words pointing into a dictionary. Branch addresses are also 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, though it could potentially be managed as a separate cache. The main advantage of these techniques is that decompression is just a fast table lookup. However, these schemes have several disadvantages. The table must be preloaded before each program is executed, which complicates multi-programmed systems, and the table fetch adds latency into the instruction pipeline, increasing branch mispredict penalties. The interleaving of code words and uncompressed instructions impedes parallel or pipelined fetch and decode. 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 [3, 4, 8]. For each encoded string, the dictionary schemes fetch the code word bits from the primary instruction stream as well as the full-size instruction from the dictionary RAM, thereby expending more energy than simply fetching uncompressed instructions.

A third approach is to use statistical compression, such as Huffman or arithmetic coding, which generates variable-size code words by choosing the size of the code word based on the frequency of the characters to replace. These techniques achieve higher compression rates than dictionary compression, but require more complex hardware and greater decompression time. Because of the slow decompression, instructions are usually compressed in main memory then uncompressed when refilling the cache, so that the decompression latency would only affect the cache refill and not the execution from the cache. This idea was introduced by Wolfe and Chanin in the Compressed Code RISC Processor (CCRP) [29], and a variety of similar techniques have subsequently been developed and commercialized [22, 14]. 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 (LAT) [29]) 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 [29]). 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 [29].

2.2 Compression of VLIW ISAs

Early techniques developed to compress code for VLIW machines only removed NOP fields within a VLIW instruction held in memory, which was then expanded on refills to include NOPs in the cache [9, 13]. More recent schemes have adapted the above three approaches to VLIW architectures, but still retain the disadvantages of their RISC counterparts. StarCore's SC140 [27] uses mixed-width instruction sets, with half-width instructions that cannot utilize the full power of the architecture. Previous dictionary-based schemes [18, 12] are only applicable to traditional VLIW architectures, which have rigid instruction formats. Modern VLIW processors usually have flexible instruction formats, in which each sub-instruction within the long instruction word does not necessarily have to correspond to a specific functional unit. One statistical compression approach [30] has targeted flexible VLIW architectures, but still uncompresses instructions upon cache refill.

The statistical approach by Aditya et al. [10] is the closest to the HAT scheme for VLIW instruction set architectures proposed in this thesis. Instructions are held compressed in cache, and uncompressed on cache hits. Apart from increasing effective cache capacity, this avoids the need for a LAT or a CLB because program counter values are the same in cache and in memory. This scheme compresses code by creating special templates and other instruction fields to eliminate NOPs and encode frequently used operands. The emphasis

is on generating a custom encoding for a specific implementation of a VLIW designed to run a specific application, with the goal of reducing the circuitry complexity of instruction fetch and decode.

2.3 CISC ISAs

The complexity of 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 would 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 [7, 15, 1, 5]. 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 [7]. 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 [5] 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.

Chapter 3

Heads and Tails Format

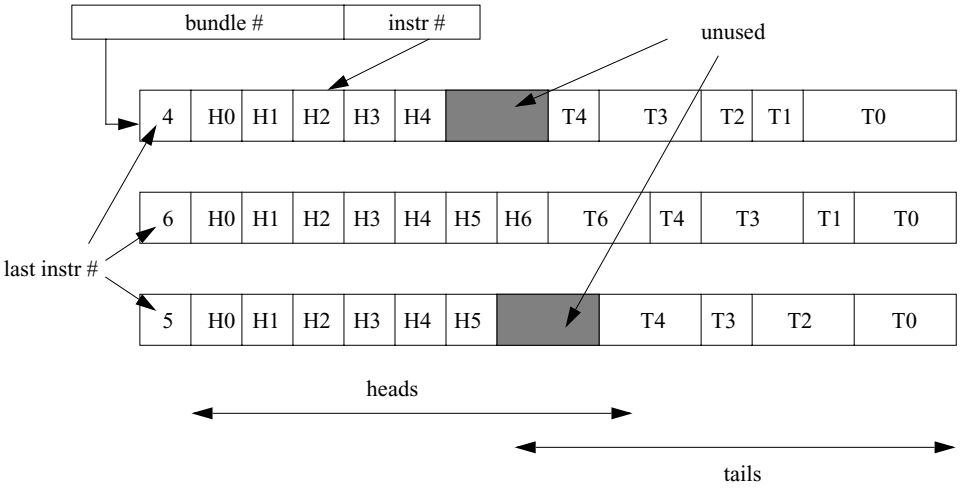

Figure 3-1: Overview of heads-and-tails format.

The HAT format packs multiple variable-length instructions into fixed-length bundles as shown in Figure 3-1. 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 each 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, whereas placing a tail pointer on each head would require a separately encoded field. 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 tail for the next instruction. The length information for each instruction is held at a fixed spacing in the head instruction stream, however, 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.

3.1 Comparison with Conventional Variable-Length Schemes

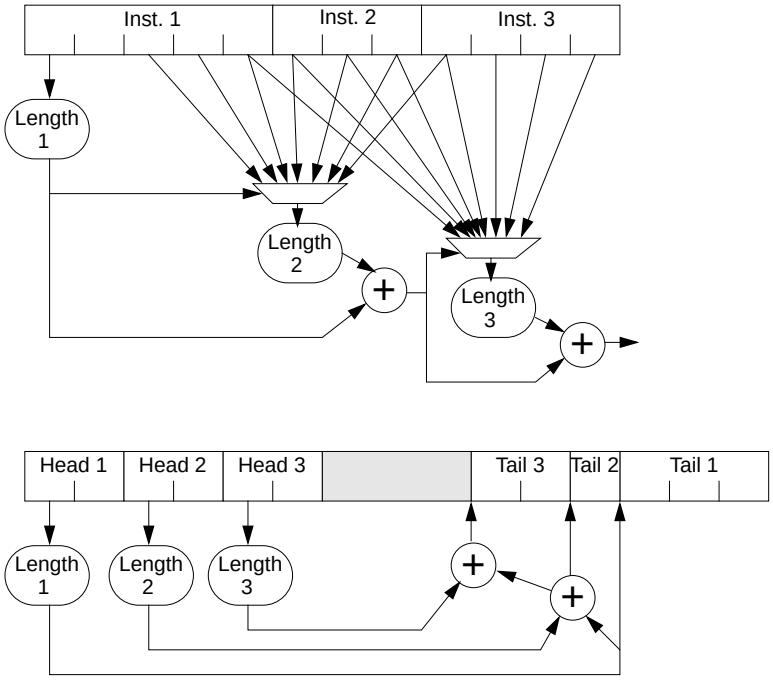

Figure 3-2: Comparison of variable-length decoding in a conventional variable-length scheme and a HAT scheme.

This difference between a regular variable-length scheme and a HAT scheme is illustrated in Figure 3-2. 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 carry-save adders, 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.

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 tail pointers 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.

3.2 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 decode a new bundle before locating the 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 in a conventional scheme the branch target address points at the entire instruction, in a HAT scheme it only locates the head within a bundle. One approach to locate the tail of a branch target is to scan all earlier heads from the beginning of the 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 branch instructions. We next describe three different approaches to handling branches 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 to locate the 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 fastest scheme would be to hold a separate tail pointer for each possible instruction in a bundle, but this incurs a large area overhead of $\log(N)$ bits per instruction. A more compact approach is to store a single bit per tail position, which indicates the 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 reducing branch penalty is to change branch and jump instruction encodings to include an additional tail pointer field that points 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. 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 for indirect jumps treats each type of indirect 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 in these cases. 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 microarchitecture has one to predict subroutine returns. If the return address stack prediction fails, execution falls back to the algorithm that scans heads from the beginning of the cache bundle.

BTB for HAT Branches

The third general approach to handling branches in a HAT scheme stores 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 the BTB mispredict penalty.

3.3 Two-Level Extension of Heads and Tails: VLIW-HAT

The above ideas can be extended into a two-level heads-and-tails instruction format suitable for VLIW architectures, which assemble multiple parallel primitive operations into a single long instruction word. In this thesis, the operations will be called *instructions* and the instruction words *bundles*. The bundles usually hold a fixed number of simple, RISC-like instructions. In traditional VLIW architectures, each instruction slot corresponds to a particular functional unit, and all the instructions in the bundle are executed in parallel; if a functional unit cannot be utilized by the bundle, that particular slot would be padded with a NOP. An in-memory compression scheme is usually used to remove the NOPs to reduce static code size, but the bundle is expanded when brought into cache [9]. More recent VLIW architectures have a more flexible encoding, where only useful instructions are represented and additional bundle stop bits are used to indicate the boundary between bundles [17, 28]. These more flexible VLIWs have better compatibility across different implementations and avoid the overhead of holding NOPs in cache, but have to provide instruction dispersal networks that route instruction bits from slots in the bundles to the appropriate functional unit.

To improve the density of VLIW instructions further, it is desirable to compress each instruction into a variable-length form before packing multiple instructions into bundles.

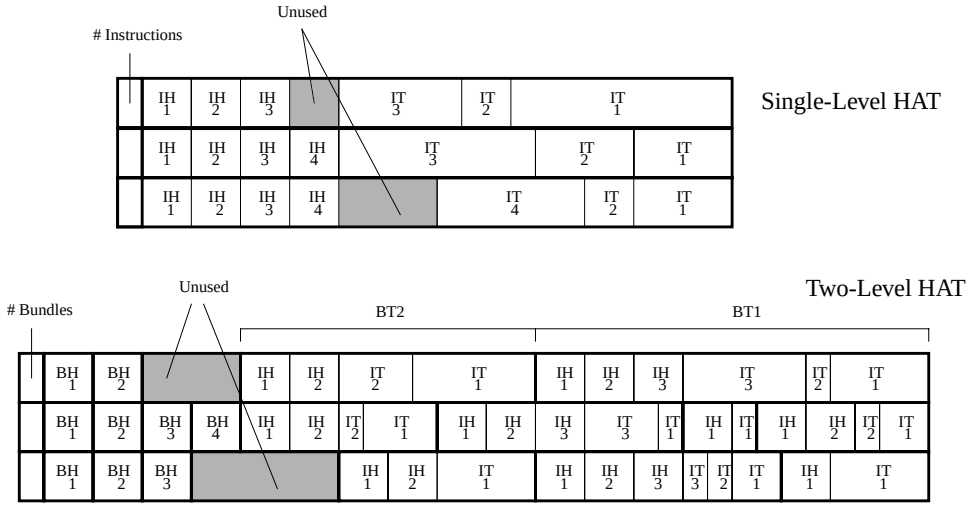

Figure 3-3: Single-Level vs Two-Level Heads and Tails Format.

This provides greater code density than simple NOP compression, but potentially increases the complexity of parallel fetch and decode of a bundle. An adaptation of the heads-and-tails format would provide a higher bandwidth fetch and decode mechanism suitable for VLIWs.

Although the path to determine variable-length tail alignment is shorter than for conventional variable-length ISAs, it could still be too lengthy for a high-performance VLIW executing many instructions per cycle. We can further speed VLIW fetch and decode by applying the heads-and-tails format at two levels: instruction-level and bundle-level. At the instruction-level, which is similar to the general HAT format, we can pack multiple variable-length compressed instructions into bundles. Then at the bundle-level, we can also split each variable-length bundle into a fixed-length bundle-head and a variable-length bundle-tail, and pack multiple bundles into a fixed-length *super-bundle*. Super-bundles contain varying numbers of bundles, so each bundle begins with a small fixed-length field specifying the number of bundles, while the rest of the super-bundle holds the actual bundles. Figure 3-3 contrasts the two-level HAT format with the original single-level HAT format.

As shown in Figure 3-4, the fetch pipeline can now be split into two steps, where the first step locates bundles using the fixed-length bundle heads. This gives enough information to locate the start of the next bundle without decoding the instructions within a bundle.

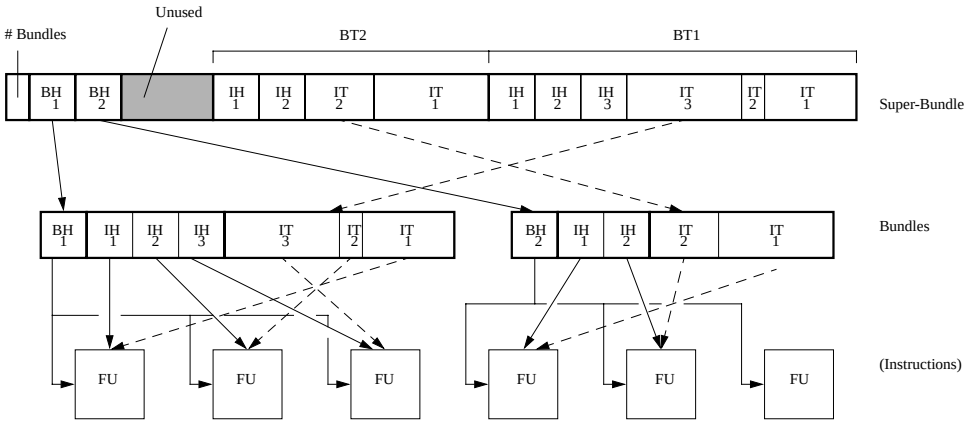

Figure 3-4: Two-Stage Fetch Pipeline.

During the first stage, the next few bundle heads are multiplexed into the second stage bundle decoders. During the second The second stage works within a bundle to split it into variable-length instructions for dispersal to the functional units. Although the diagram shows the conceptual path of instruction tails through the two stages, the instruction tail bits can be moved in one step from the super-bundle to the functional units.

Some flexible VLIW architectures pack instructions into fixed-length fetch units and have stop bits separating variable-length parallel instruction groups. There are two choices of how to use a two-level VLIW-HAT scheme in this case. The super-bundle can either hold bundles that are fetch units, or bundles that are parallel instruction groups. Encoding parallel instruction groups as bundles poses the difficulty that the groups can potentially be very long giving a wide variation in bundle tail length. This is a limitation of template schemes that assume a canonical instruction format [10]. The fetch units are better candidates for encoding in a bundle since they represent a fixed number of instructions.

When coping with branches, all three techniques described above for the single-level HAT scheme are also applicable to the VLIW-HAT scheme. With predication, VLIW machines encounter fewer branches in the dynamic instruction stream, reducing the penalty for some of the branch tail location strategies.

Chapter 4

MIPS-HAT

In this section, the HAT format is illustrated using a compressed variable-length re-encoding of the MIPS RISC ISA [19] as an example.

4.1 MIPS Instruction Set Overview

The MIPS II instruction set can be divided into the following categories:

- *Computational Instructions* – perform arithmetic, logic, and shift operations on values in registers.
- *Load/Store Instructions* – move data between memory and registers.
- *Coprocessor 0 Instructions* – perform operations on CP0 registers; handle memory management and exception handling.
- *Special Instructions* – move data between special and general registers; perform system calls and breakpoints.

These instructions can generally be represented by one of three instruction formats – R-type (register), I-type (immediate), and J-type (jump). The breakdown of these instruction types into different subfields are shown in Figure 4-1 and explained in Table 4.1.

The MIPS II instruction set has two types of opcodes. The major opcode is specified in the first 6-bit subfield [31:26]. If this major opcode field is zero, the last 6-bit subfield [5:0],

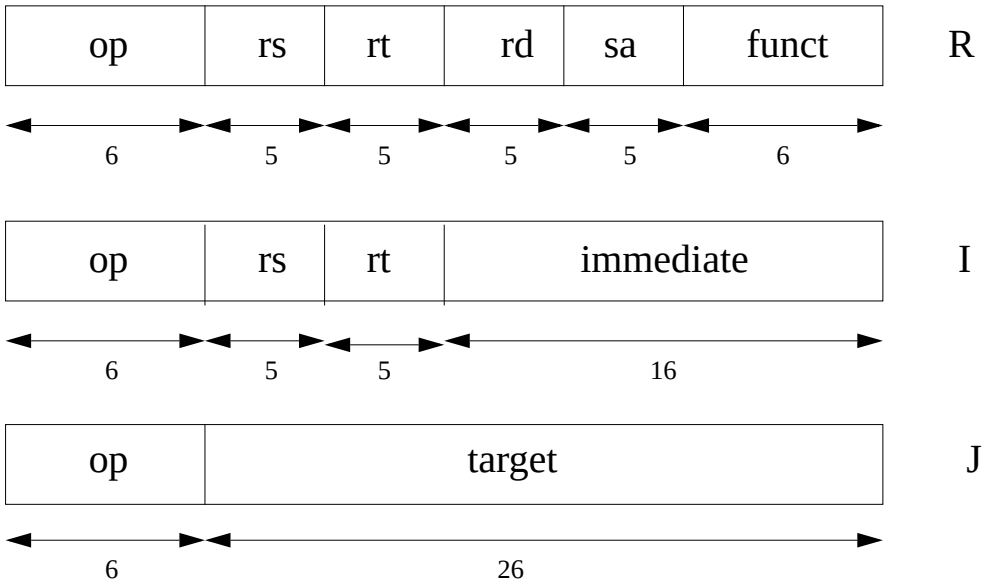

Figure 4-1: MIPS instruction formats.

Table 4.1: MIPS Subfield definitions.

Subfield	Bits	Definition
op	6	major operation code
rs	5	source register specifier
rt	5	target (source/destination) register or branch condition
immediate	16	immediate, branch, or address displacement
target	26	jump target address
rd	5	destination register specifier
sa	5	shift amount
funct	6	function field

Table 4.2: MIPS Primary and Secondary Opcode Encodings.

OPCODE

31..29 \ 28..26	0	1	2	3	4	5	6	7
0	SPECIAL	REGIMM	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	COP0				BEQL	BNEL	BLEZL	BGTZL
3								
4	LB	LH		LW	LBU	LHU		
5	SB	SH		SW				
6								
7								

SPECIAL FUNCTION

5..3 \ 2..0	0	1	2	3	4	5	6	7
0	SLL		SRL	SRA	SLLB		SRLY	SRAV
1	JR	JALR			SYSCALL	BREAK		SYNC
2	MFHI	MTHI	MFLO	MTLO				
3	MULT	MULTU	DIV	DIVU				
4	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5			SLT	SLTU				
6								
7								

called the function field, specifies a secondary opcode used for three register arithmetic and logic instructions. Table 4.2 shows the encodings of these two opcode fields for all integer instructions. All blanks in these tables signify undefined opcodes.

4.2 MIPS-HAT Compression Techniques

The MIPS compression scheme used is based partly on a previous scheme by Panich [24]. To keep instruction decoding simple, the MIPS register specifier fields are never split, so a 5-bit granularity for the tail encoding is used. Instructions range from 15 bits to 40 bits long. As discussed later in the hardware section, tail lookup can be simplified if every instruction has a tail, so the 10-bit heads yeild a minimum instruction size of 15 bits.

As seen in the previous section, the MIPS opcode space is very sparce. Therefore, it is easy to condense the opcode space into less than 12 bits and still have extra space to

encode additional information to eliminate operand fields or even additional instructions. Specifically, the following techniques were used to compress the MIPS instructions by taking advantage of the extra opcode space.

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 ($\approx$90%) has zero as one operand. New opcodes are provided for these cases.

4. Two-address versions are provided for instructions that frequently have a source register the same as the destination register.

5. Some common instruction sequences are re-encoded as a single instruction. Only the simplest but most common two types of instruction sequences are re-encoded: branch instructions with a NOP in the delay slot and multiple 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. A multiple store instruction was considered, but this did not provide sufficient savings to be justified.

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 the instruction sizes are folded 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. The most popular opcodes are selected, 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

Table 4.3: 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%			

also includes critical information required for decode, such as the size of the instruction and its general category (e.g., arithmetic versus branch). Though both the original and compressed formats contain two opcode fields, where the secondary field is only used in conjunction with the escape primary opcode, the functionalities of the secondary field are very different. In the original MIPS format, the secondary field (also called the function field) encodes three register arithmetic and logic instructions, and both instruction fields are very sparce. On the other hand, the secondary field of the compressed format only encodes the less frequently used opcodes that cannot fit in the primary field

Table 4.3 and Table 4.4 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.

Figure 4-2 shows the compressed forms of the three types of MIPS instructions — register (R), immediate (I), and jump (J). All fields are 5 bits wide. The fields in parenthesis

Table 4.4: 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%			

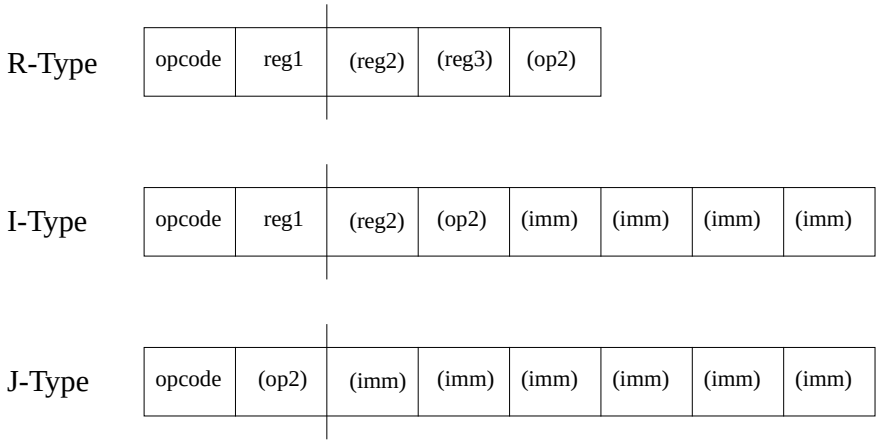

Figure 4-2: Compressed MIPS instruction formats.

are optional, depending on the actual instruction.

4.3 Bundle Format

Both 128-bit and 256-bit bundles are evaluated 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 the head and tail regions are restricted so that neither completely spans the bundle, but the boundary between head and tail sections is flexible and depends on the instructions encoded.

4.4 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. The cache RAM sense-amps just need a separate set of bus drivers onto the head data bus (additional bus drivers are only needed for the bits in the middle of a span that could either be heads or tails).

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 to an appropriate functional unit before the tail arrives, allowing the tail to be sent directly

to the appropriate unit for further decoding.

4.5 Experimental Results

To test the effectiveness of the MIPS-HAT scheme, we selected benchmarks from the Mediabench [21] 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.

4.5.1 Static Compression Ratios

Table 4.5 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 for the 128b bundle is 77.3% and for the 256b bundle is 74.1%. The smaller bundle incurs relatively more overhead and has more internal fragmentation. If we adopt the scheme that adds branch tail links to speed taken branches, the static code size gets worse, 80.0% for 128b bundles and 77.9% for the 256b bundles.

Table 4.6 shows the distribution of static instruction sizes averaged over the benchmark set, with and without the tail pointer scheme. The majority of instructions are 25 bits or less.

4.5.2 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 branch penalties

Tables 4.7 and 4.8 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

Table 4.5: Static Compression Ratios

Input	128b	256b	128b-BrTail	256b-BrTail
adpcm-dec	77.1%	73.8%	80.4%	77.7%
adpcm-enc	77.2%	74.0%	80.4%	77.8%
epic-dec	75.5%	72.2%	78.3%	76.1%
epic-enc	76.5%	73.2%	79.3%	77.2%
g721-dec	76.9%	73.6%	79.6%	77.8%
g721-enc	76.9%	73.6%	79.7%	77.9%
gsm-dec	79.1%	76.1%	82.3%	80.0%
gsm-enc	79.1%	76.1%	82.3%	80.0%
jpeg-dec	73.5%	70.5%	75.8%	74.3%
jpeg-enc	73.3%	70.1%	76.0%	74.1%
mpeg2-dec	79.1%	76.0%	81.9%	79.7%
mpeg2-enc	79.0%	75.8%	81.4%	79.4%
pegwit-dec	79.3%	76.0%	81.5%	79.6%
pegwit-enc	79.3%	76.0%	81.5%	79.6%
average	77.3%	74.1%	80.0%	77.9%

Table 4.6: Instruction Size Distribution

	15b	20b	25b	30b	35b	40b
Average (w/o BrTail)	20.8%	20.1%	43.4%	7.9%	2.2%	5.6%
Cumulative	20.8%	40.9%	84.3%	92.2%	94.4%	100.0%
Average (w/ BrTail)	18.1%	23.1%	29.0%	22.5%	2.4%	4.9%
Cumulative	18.1%	41.2%	70.2%	92.7%	95.1%	100.0%

Table 4.7: 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	71.8%	80.1%	74.6%
epic-enc	76.8%	80.6%	78.9%
g721-dec	75.3%	82.2%	78.4%
g721-enc	75.6%	82.2%	78.5%
gsm-dec	75.5%	79.6%	75.9%
gsm-enc	72.0%	74.1%	74.7%
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	74.7%	80.2%	76.7%

has a slightly lower fetch ratio (74.1% versus 74.7%) as relatively fewer overhead bits are fetched.

The BrBV column shows the large increase in dynamic fetch ratio when a bit vector scheme (Section 3.2) is used to reduce branch taken penalties. The increase is less for the 128b bundles which have a 16b vector per line, and now these 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 numbers are much lower than for the BrBV approach.

4.5.3 Results Discussion

The numbers show there is tradeoff 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 the effective increase in cache capacity, which should lower miss rates.

Table 4.8: 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	71.8%	88.4%	74.3%
epic-enc	76.1%	83.8%	78.3%
g721-dec	75.0%	88.9%	78.5%
g721-enc	73.8%	87.7%	78.4%
gsm-dec	74.8%	83.1%	73.8%
gsm-enc	71.3%	75.5%	71.3%
jpeg-dec	67.9%	73.5%	68.8%
jpeg-enc	72.4%	86.3%	74.4%
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	74.1%	86.6%	75.6%

Chapter 5

IA64-HAT

In this section, the two-level HAT format is illustrated using a compressed variable-length re-encoding of the IA-64 ISA [17].

5.1 IA-64 Instruction Set Overview

The IA-64 instruction set architecture has 128-bit fetch units. Each contains a 5-bit template field and three 41-bit instructions, as shown in Figure 5-1.

The template field specifies the functional units and stops. The following are the different types of IA-64 functional units:

- M: Memory, ALU
- I: Shifts, MM, ALU
- B: Branch

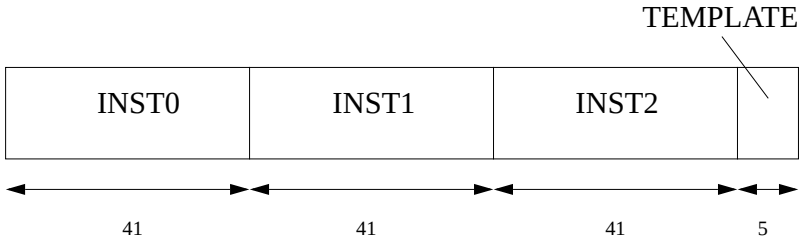

Figure 5-1: IA-64 instruction fields.

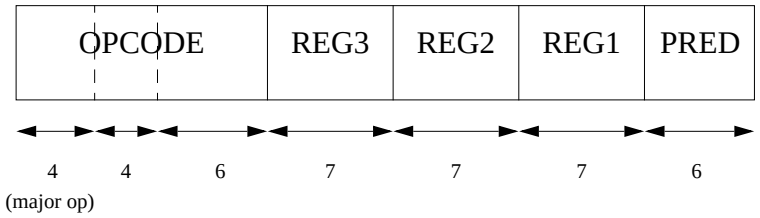

Figure 5-2: IA-64 instruction fields.

- F: Floating Point
- L+X: Long Immediate

The template types are listed in Table 5.1. A vertical bar after a function unit indicates a stop bit, which delimits a sequence of instructions with no register dependencies. Therefore, the templates enable a flexible instruction format in which instruction slots do not always correspond to the same functional unit and execution lengths are independent of fetch unit lengths.

Each instruction can be be divided into a 14-bit opcode field (containing a 4-bit major op field), three 7-bit register fields, and a 6-bit predicate field. However, many instructions have immediates and opcode extension bits that are encoded within the opcode and register fields.

5.2 IA64-HAT Compression Techniques

To compress IA-64 instructions, we use simple statistical techniques. At the instruction level, we divide an instruction into seven fields, as shown in Figure 5-2. We compress each instruction by creating new opcodes that specify values of one or more of these fields as well as the operation. The most frequently used opcodes are encoded in the instruction head, and all unspecified fields are encoded in the tail. The tails are fetched slightly after the heads, and so we attempt to pack critical information in the head to minimize the impact on pipeline latency. In an in-order flexible VLIW machine, source register information is critical as it is used to interlock on register reads. Destination register information is only needed to interlock instructions in the next cycle's instruction group and so can be delayed

Table 5.1: IA-64 Templates

Template	Instruction Slots
00	M I I
01	M I I |
02	M I | I
03	M I | I |
04	M L X
05	M L X |
06	
07	
08	M M I
09	M M I |
0A	M | M I
0B	M | M I |
0C	M F I
0D	M F I |
0E	M M F
0F	M M F |
10	M I B
11	M I B |
12	M B B
13	M B B |
14	
15	
16	B B B
17	B B B |
18	M M B
19	M M B |
1A	
1B	
1C	M F B
1D	M F B |
1E	
1F	

and placed in the tail. The IA64-HAT encoding always places the first source register field in the head, while instructions that use a second source register almost always have that information encoded in the head as a new opcode.

We re-encode instructions into one of seven sizes, ranging from 14 to 56 bits. The encoding is designed to be straightforward to decode to reduce the complexity of the instruction decoder with a only a few formats. The 14-bit heads contain a 7-bit opcode field and a 7-bit source register field. The opcode field encodes the top 31 most frequently used opcodes plus an escape opcode for less frequently used opcodes, as shown in Table 5.2. The opcode field also has two additional bits indicating the existence of the predicate and r2 fields in the tail portion, since these fields are often not used in the actual instruction encoding. The tails are also encoded in 7-bit increments. The length of the tail can always be determined by the head, since the opcodes determine which fields are encoded in the tail. One of the schemes to handle branches in HAT requires a tail pointer be added to all branch instructions to point to the location of the tail of the branch target within the target bundle, as described in Section 3.2. These tail pointers are filled in at link time.

Although the IA-64 is a flexible VLIW architecture, NOPs still occur frequently because the template encoding is not fully general and requires NOP padding. At the bundle level, we can eliminate NOPs by encoding their positions within the bundle head. Between 0–3 NOPs can occur in a bundle, and we specify their eight possible positions in a 3-bit field. These special templates are listed in Table 5.3. Because of NOP compression, each bundle now encodes a variable number of instructions. The bundle heads also contain the 5-bit template field, and a 5-bit tail size field since the bundle tails range from 0 to 168 bits. An extra padding bit is added onto the bundle head to make it 14 bits long, so that the head length is a multiple of the tail granularity. Having a consistent instruction field granularity throughout the super-bundle simplifies the fetch and alignment hardware.

5.3 Experimental Results

We evaluate the use of both 512-bit and 1024-bit super-bundles to pack IA64-HAT bundles. The 512b super-bundle is split into a four-bit bundle-count field and 72×7b units holding

Table 5.2: Special Opcodes

Instruction Group		Major Op	Op [36:33]	Op [32:27]	Reg1 [12:6]	Bits Saved
A1	ALU	8	0	0		14
A1	ALU	8	0			8
A2	Shift L and Add	8	0			8
A4	Add Imm14	8	8	0	15	21
A4	Add Imm14	8	8	0	1	21
A4	Add Imm14	8	8	0	8	21
A4	Add Imm14	8	8	0	14	21
A4	Add Imm14	8	8	0	33	21
A4	Add Imm14	8	8	0		14
A5	Add Imm22	9	0	0		14
A5	Add Imm22	9				4
A6	Compare		0			4
A6	Compare		4			4
A8	Compare Imm8	14	12	7	6	21
A8	Compare Imm8	14	12	6	7	21
B1	IP-Relative Branch	4	0	0	0	21
B1	IP-Relative Branch	4	4	0	0	21
B1	IP-Relative Branch	4	6	0	0	21
B2	Counted Branch	4	5	63	0	21
B2	Counted Branch	4	5	63		14
B3	IP-Relative Call	5	1		64	15
M1	Int Load	4	0	24	14	21
M1	Int Load	4	0	24	15	21
M1	Int Load	4	0	0		14
M1	Int Load	4	0	8		14
M1	Int Load	4	0	16		14
M1	Int Load	4	0	24		14
M4	Int Store	4	12	24	0	21
M4	Int Store	4	12		0	15
I29	Sxt/Zxt/Czx	0	0			8
	NOP					21

Table 5.3: IA-64 Templates

Original Template	NOP Position(s)
09 M M I |	2
0A M | M I	2
0B M | M I |	2
0D M F I |	0,2
0D M F I |	1,2
10 M I B	0,1
11 M I B |	0,1
11 M I B |	1

up to 16 bundles. The 1024b super-bundle is split into a five-bit bundle-count field and 145$\times$7b units holding up to 32 bundles. The limit on the maximum number of bundles held in a super-bundle is to reduce the number of bits for the bundle-count field, but this restriction rarely affects the bundle packing.

To test the effectiveness of the IA64-HAT scheme, we selected benchmarks from the Mediabench [21] and SPECint95 [2] benchmark suites, re-encoded the IA-64 binaries generated by the Intel Open Research Compiler [16], and took static and dynamic measurements. For the dynamic measurements, the benchmark programs were run to completion on the provided input sets.

5.3.1 Static Compression Ratios

Table 5.4 gives the static compression ratios (compressed-size/original-size) for 512b and 1024b versions of IA64-HAT. The compression ratios for the two sizes include the overhead bits to specify the number of bundles in the super-bundle and any wasted space due to fragmentation.

The average compression ratio is 61.0% for the 512b super-bundle and 58.2% for the 1024b super-bundle. The smaller super-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 63.1% for 512b super-bundles and 60.2% for the 1024b super-bundles. The impact on code size is less than in the MIPS-HAT study because branches are less frequent in the VLIW code.

Table 5.4: Static Compression Ratios

Input	512b	512b BrTail	1024b	1024b BrTail
adpcm-dec	59.6%	61.7%	56.9%	58.9%
adpcm-enc	59.6%	61.7%	56.9%	58.9%
epic-dec	60.6%	62.7%	57.9%	59.9%
g721-dec	60.2%	62.4%	57.4%	59.6%
g721-enc	60.2%	62.4%	57.4%	59.6%
jpeg-dec	63.6%	65.3%	60.7%	62.4%
jpeg-enc	62.1%	64.0%	59.4%	61.1%
mpeg2-dec	62.0%	64.0%	59.1%	61.1%
compress	60.1%	62.3%	57.4%	59.4%
ijpeg	65.0%	66.6%	62.0%	63.6%
li	59.2%	61.3%	56.6%	58.6%
lzw	60.1%	62.3%	57.4%	59.5%
average	61.0%	63.1%	58.2%	60.2%

Table 5.5: Static Compression Amounts by Level (W/O BrTail)

Input	512b Bundle	512b Instr	1024b Bundle	1024b Instr
adpcm-dec	10.6%	29.8%	16.6%	26.5%
adpcm-enc	10.6%	29.8%	16.6%	26.5%
epic-dec	12.7%	26.7%	18.5%	23.6%
g721-dec	10.0%	29.8%	16.1%	26.5%
g721-enc	10.0%	29.8%	16.1%	26.5%
jpeg-dec	11.7%	24.7%	17.7%	21.6%
jpeg-enc	13.0%	24.9%	18.8%	21.8%
mpeg2-dec	12.2%	25.8%	18.2%	22.7%
compress	11.1%	28.8%	17.1%	20.7%
ijpeg	12.9%	22.1%	18.7%	19.3%
li	15.1%	25.7%	20.6%	22.8%
lzw	10.1%	29.8%	16.2%	26.4%
average	11.7%	27.3%	17.6%	23.8%

Table 5.6: Instruction Size Distribution

	0b	14b	21b	28b	35b	42b	49b	56b
Average (w/o BrTail)	32.5%	8.6%	20.5%	16.0%	8.7%	12.0%	1.8%	0.0%
Cumulative	32.5%	41.0%	61.5%	77.5%	86.2%	98.2%	100.0%	100.0%
Average (w/ BrTail)	32.5%	8.4%	18.3%	12.0%	14.4%	11.9%	1.4%	1.1%
Cumulative	32.5%	40.9%	59.2%	71.2%	85.6%	97.5%	98.9%	100.0%

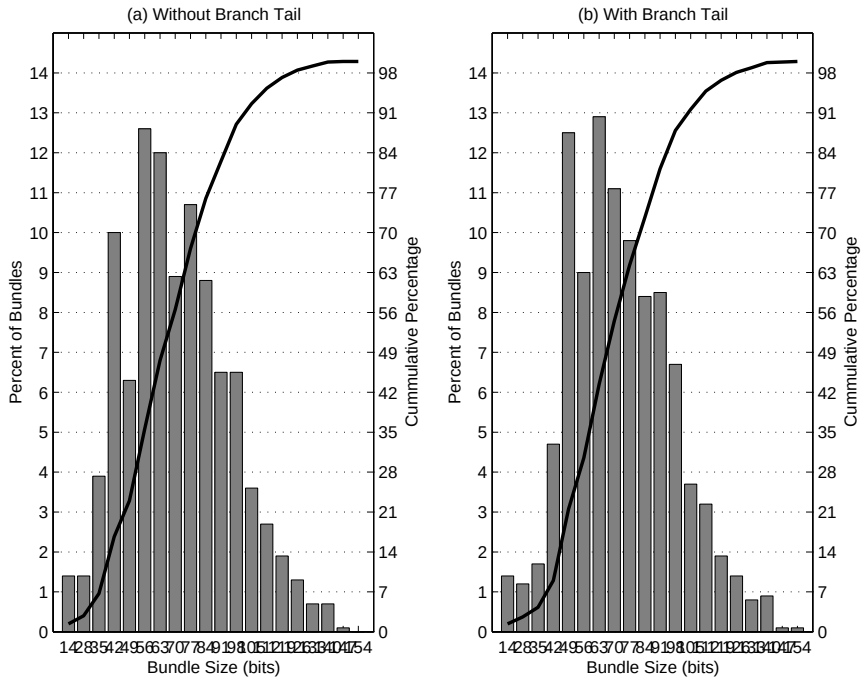

Figure 5-3: Bundle Size Distribution.

Table 5.5 shows how much the static code size is reduced by each level of compression. Eliminating NOPs condenses the code by about 11–18%, while re-encoding of the individual instructions condenses the code by a further 23–27%.

Table 5.6 and Figure 5-3 give the distribution of compressed instruction sizes and bundle sizes, respectively. Almost half of the bundles are compressed by 50% or more from the original 128-bit wide bundle.

5.3.2 Dynamic Measures

We also measured the reduction in dynamic bits fetched from the instruction cache using the IA64-HAT scheme. We report this number as a dynamic compression ratio (new-bits-fetched/original-bits-fetched). The ratio includes all overhead bits that have to be fetched when executing a new bundle or super-bundle.

Table 5.7 shows the dynamic fetch ratios for 512b and 1024b super-bundles. The average compression ratio is 60.0% for the 512b super-bundle and 57.8% for the 1024b super-bundle. The dynamic fetch ratio is only affected by overhead and not internal fragmentation, so the smaller super-bundle incurs a smaller penalty in the dynamic ratios than in

Table 5.7: Dynamic Compression Ratios

Input	512b	512b BrTail	1024b	1024b BrTail
adpcm-dec	65.0%	66.7%	64.7%	66.4%
adpcm-enc	77.4%	79.1%	77.2%	78.8%
epic-dec	50.9%	51.7%	50.8%	51.6%
g721-dec	63.4%	64.9%	63.4%	64.3%
g721-enc	45.5%	46.8%	45.5%	46.8%
jpeg-dec	59.0%	60.4%	58.8%	60.2%
jpeg-enc	55.7%	57.2%	55.6%	57.0%
mpeg2-dec	56.4%	57.9%	56.2%	57.9%
compress	61.2%	64.4%	61.1%	64.4%
ijpeg	59.4%	61.2%	59.2%	61.0%
li	53.2%	55.4%	53.1%	55.4%
lzw	58.1%	55.6%	55.3%	55.5%
average	60.0%	61.7%	57.8%	59.9%

Table 5.8: Dynamic Compression Amounts by Level (W/O BrTail)

Input	512b Bundle	512b Instr	1024b Bundle	1024b Instr
adpcm-dec	13.9%	21.1%	14.2%	21.1%
adpcm-enc	8.6%	14.0%	9.1%	13.7%
epic-dec	33.2%	15.9%	33.3%	15.9%
g721-dec	11.6%	25.0%	12.2%	24.4%
g721-enc	34.5%	20.0%	34.6%	19.9%
jpeg-dec	21.3%	19.7%	21.5%	19.7%
jpeg-enc	25.0%	19.3%	25.3%	19.1%
mpeg2-dec	21.1%	22.5%	21.3%	22.5%
compress	20.4%	18.4%	20.7%	18.2%
ijpeg	22.6%	18.0%	22.8%	18.0%
li	23.9%	22.9%	24.0%	22.9%
lzw	37.0%	4.9%	37.0%	7.7%
average	22.8%	18.5%	23.0%	18.6%

the static ratios. The added tail pointer fields also slightly increases the dynamic fetch ratios, to about 61.7% for 512b super-bundles and 59.9% for 1024b super-bundles. This increase is even less than the increase in static compression ratios, because the encoded branch instructions are rarely executed.

The dynamic compression at each level is shown in Table 5.8. The NOP compression reduces the code size by almost a quarter, and the individual instruction compression reduces it by another 18%. The bundle-level compression is more significant in the dynamic compression than in static code reduction.

5.3.3 Compression Discussion

Other researchers have presented compression numbers for VLIW architectures. Previous dictionary schemes [18, 12] have obtained compression ratios of 46–60% and 63–51%, respectively, but are only applicable to rigid VLIW architectures. The more complex compression scheme given in [30] achieves 70–80% compression ratio on TMS320C6x code. The PICO-VLIW format [10] is similar but uses a customizable template scheme targeted for application-specific implementations, whereas VLIW-HAT is designed to provide a portable flexible encoding with low control complexity.

5.3.4 Performance Discussion

The heads-and-tails format enhances performance because it allows instructions to remain compressed in the cache, thereby increasing the effective cache capacity and lowering the cache miss rate. On the other hand, it also reduces performance because the decoding of HAT instructions is more complicated than native VLIW instructions, so an extra decode stage may have to be added to the pipeline.

Tables 5.9 and 5.10 show the average cache miss rates of compressed and uncompressed instructions using different sized caches and different sized cache line sizes to execute Mediabench bencharks. The uncompressed instructions always yield a worse miss rate, anywhere from 1.5–6 times that of the compressed instructions. In order to achieve a comparable miss rate using uncompressed instructions, the size of the cache needs to be

Table 5.9: Cache Miss Rates of Different Cache Sizes (64B Cache Line, 4-way Associativity, 512b Superbundle)

Cache Size	Compressed	Uncompressed
8 KBytes	0.213%	0.602%
16 KBytes	0.015%	0.090%
32 KBytes	0.007%	0.012%
64 KBytes	0.006%	0.009%

Table 5.10: Cache Miss Rates of Different Cache Line Sizes (32 KB Cache, 4-way Associativity, 512b Superbundle)

Cache Line Size	Superbundles/Line	Compressed	Uncompressed
64 Bytes	1	0.007%	0.012%
128 Bytes	2	0.004%	0.007%
256 Bytes	4	0.003%	0.005%

increased. Longer cache lines also reduce the miss rate, but compressed instructions in the HAT format still achieve a better miss rate than uncompressed instructions for the same sized cache line. In addition, embedded systems that have L1 cache refilling directly from off-chip memory incur severe cache miss penalties, which could be ameliorated by HAT's decreased cache miss rates. On-chip caches also account for a large percentage of embedded systems' area and power, and the HAT format enables a much smaller cache to yield similar performance.

The additional pipeline stage for decoding HAT instructions will reduce the performance by increasing branch mispredict penalties. However, in VLIW designs such as IA-64, predication reduces the number of branch instructions, and complex branch prediction schemes decrease the misprediction rate, so the increased branch misprediction penalties should not affect performance significantly.

Chapter 6

Conclusion

This thesis has introduced a new heads-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. A two-level version of this format can also provide both high density and performance for VLIW schemes. 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. A simple MIPS instruction compression scheme was developed, and the resulting variable-length instructions were mapped into the HAT format. These experiments showed that the MIPS-HAT format can provide a compression ratio of 74.7% and a dynamic fetch ratio reduction of 74.1% while supporting deeply pipelined or superscalar execution. The two-level HAT format was applied to the IA-64 flexible VLIW format, and achieved a compression ratio of around 60%.

Bibliography

[1] *AMD Athlon Processor x86 Code Optimization*, chapter Appendix A: AMD Athlon Processor Microarchitecture. AMD Inc., 220071-0 edition, September 2000.

[2] Standard Performance Evaluation Corporation. Spec95, 1995. http://www.spec.org.

[3] C. Lefurgy et al. Improving code density using compression techniques. In *MICRO-30*, pages 194–203, Research Triangle Park, North Carolina, December 1997.

[4] G. Araujo et al. Code compression based on operand factorization. In *MICRO-31*, pages 194–201, December 1998.

[5] G. Hinton et al. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, Q1 2001.

[6] J. Choquette et al. High-performance RISC microprocessors. *IEEE Micro*, 19(4):48–55, July/August 1999.

[7] J. Circello et al. The superscalar architecture of the MC68060. *IEEE Micro*, 15(2):10–21, April 1995.

[8] L. Benini et al. Selective instruction compression for memory energy reduction in embedded systems. In *ISLPED*, pages 206–211, August 1999.

[9] R. P. Colwell et al. A VLIW architecture for a trace scheduling compiler. *IEEE Trans. Computers*, 37(8):967–979, August 1988.

[10] S. Aditya et al. Code size minimization and retargetable assembly for custom EPIC and VLIW instruction formats. *ACM Trans. Design Automation of Electronic Systems*, 5(4):752–773, October 2000.

[11] S. Liao et al. Code optimization techniques for embedded dsp microprocessors. In *DAC*, 1995.

[12] S. Nam et al. Improving dictionary-based code compression in VLIW architectures. *IEICE Trans. Fundamentals*, E82-A(11):2318–2324, November 1999.

[13] T. Conte et al. Instruction fetch mechanism for VLIW architectures with compressed encodings. In *MICRO-29*, pages 201–211, Paris, France, December 1996.

[14] T. M. Kemp et al. A decompression core for PowerPC. *IBM J. Res. & Dev.*, 42(6):807–812, November 1998.

[15] L. Gwennap. Intel's P6 uses decoupled superscalar design. *Microprocessor Report*, 9(2):9–15, February 1995.

[16] IA-64 Open Research Compiler. http://ipf-orc.sourceforge.net/.

[17] *Intel IA-64 Architecture Software Developer's Manual*. Intel, January 2000.

[18] N. Ishiura and M. Yamaguchi. Instruction code compression for application specific VLIW processors based on automatic field partitioning. In *The Workshop on Synthesis and System Integration of Mixed Technologies*, pages 105–109, December 1997.

[19] G. Kane. *MIPS RISC Architecture (R2000/R3000)*. Prentice Hall, 1989.

[20] Kevin D. Kissell. MIPS16: High-density MIPS for the embedded market. In *Proceedings RTS97*, 1997.

[21] 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.

[22] H. Lekatsas and W. Wolf. Code compression for embedded systems. In *DAC*, pages 516–521, San Francisco, CA, June 1998.

[23] IBM Microelectronics. PowerPC 440GP embedded processor: High performance SOP for networked applications. Presentation from Embedded Processor Forum, June 2000.

[24] M. Panich. Reducing instruction cache energy using gated wordlines. Master's thesis, Massachusetts Institute of Technology, August 1999.

[25] S. Segars, K. Clarke, and L. Goudge. Embedded control problems, Thumb, and the ARMT7TDMI. *IEEE Micro*, 15(5):22–30, October 1995.

[26] SiByte, Inc. SB-1 CPU fact sheet. at www.sibyte.com, October 2000. rev. 0.1.

[27] Starcore sc140. http://www.starcore-dsp.com/technology/sc140core.html.

[28] J. Turley and H. Hakkarainen. TI's new 'C6x DSP Screams at 1,600 MIPS. *Microprocessor Report*, 11(2):14–15, February 1997.

[29] A. Wolfe and A. Chanin. Executing compressed programs on an embedded RISC architecture. In *MICRO 25*, pages 81–91, Portland, Oregon, December 1992.

[30] Y. Xie, W. Wolf, and H. Lekatsas. A code decompression architecture for VLIW processors. In *MICRO-34*, pages 66–75, Austin, Texas, December 2001.