Hardware/Software Tradeoffs for Increased Performance

John Hennessy, Norman Jouppi, Forest Baskett, Thomas Gross, and John Gill

Departments of Electrical Engineering and Computer Science
Stanford University

Abstract

Most new computer architectures are concerned with maximizing performance by providing suitable instruction sets for compiled code and providing support for systems functions. We argue that the most effective design methodology must make simultaneous tradeoffs across all three areas: hardware, software support, and systems support. Recent trends lean towards extensive hardware support for both the compiler and operating systems software. However, consideration of all possible design tradeoffs may often lead to less hardware support. Several examples of this approach are presented, including: omission of condition codes, word-addressed machines, and imposing pipeline interlocks in software. The specifics and performance of these approaches are examined with respect to the MIPS processor.

1. Introduction

Until recently, the design of new computer architectures and the demands of high level language compilation were not well integrated. Instead, architectures were dominated by concerns of assembly language programs and maintaining capability with features contained in old architectures. Recently, architectures have been more conscious of their major role as host machines for high level language programs. This development is evident in the modern microprocessors (e.g. the Z8000 and the 68000) and larger machines such as the VAX.

The MIPS project has been supported by the Defense Advanced Research Projects Agency under contract # MDA903-79-C-0680. Thomas Gross is supported by an IBM Graduate Fellowship.

These architectures certainly address the concerns of running compiled programs more than past efforts have done. However, they assume a fixed and existent compiler technology. In most cases, compilers for other machines are used as a basis for examining the instruction set design [9, 13]. While this methodology is certainly better than a nonquantative approach, it is inherently flawed. We propose a methodology where the compiler, and in fact even new compiler technology, are major inputs to the instruction set design process. The notion of the compiler and its code generator playing a vital role in the instruction set design process has been used in the design of a processor to run C [8]. In this paper, we will advocate more radical applications of compiler technology.

Adequate support for constructing systems software, e.g. the operating system, and for programs compiled from a high level language, is a necessity. While instruction set design issues are largely ones of performance, cost, and reliability, inadequate architectural support for the operating system often leads to real limitations in using the processor. Such limitations can determine which memory management techniques (swapping versus paging) are possible or feasible and might restrict any system containing such a processor. On the other hand, including elaborate support in the processor architecture can often cause an overall decrease in performance because of the increased overhead. In many architectures, system issues (e.g., orthogonal handling of interrupts and traps) are given low consideration. This trend prevails in many current microprocessor architectures; for example, in several architectures page faults are not accommodated. In other architectures, systems issues are the focus of the design to the detriment of performance, as in the Intel iAPX-432.

We will argue that architecture/compiler/system tradeoffs are inevitable, and that by correctly making choices in all three areas the designer can obtain a simpler design that will have increased performance and reliability for lower cost. We demonstrate several applications of this approach and give specific examples and empirical performance data from the MIPS project [5]. We are primarily concerned with procedural languages (e.g., ALGOL, Pascal, C, and Ada) and with von-Neuman architectures.

2. Architectural support for compiled languages

Architectures can support compiled languages in many ways. For high performance in both executed code and within the compilers themselves, architectures can best support high level languages by providing simple, fast instruction sets that are easy for a compiler to use.

2.1. Simple and fast instructions

This approach of a simpler instruction set and its attractiveness has been argued and demonstrated by the RISC project [11]. From the compiler viewpoint, there are three advantages of a simple instruction set:

1. Because the instruction set is simpler, individual instructions execute faster.
2. The compiler is not faced with the task of attempting to utilize a very sophisticated instruction that doesn't quite fit any particular high level construct. Besides slowing down other instructions, using these instructions is sometimes slower than a customized sequence of simpler instructions [12].
3. Although these architectures may require more sophisticated compiler technology, the potential performance improvements to be obtained from faster machines and better compilers are substantial.

2.2. Load/store architectures

Compilers find load/store architectures to be a natural problem decomposition: first get the operands, then use them. Requiring compilers to do both at the same time when the architecture is not completely orthogonal is more complex. Load/store architectures can also increase the performance of compiled languages.

Load/store architectures can yield performance increases if frequently-used operands are kept in registers. Not only is redundant memory traffic decreased, but addressing calculations are saved as well. Software support for load/store architectures does not appear to be a problem either: there are efficient register allocation algorithms which produce good assignments [3, 1]. Although these algorithms require powerful and sophisticated compiler technology, they pay off by yielding very dense and near-optimal register assignments. Simpler register allocation algorithms are possible. However, these algorithms are not as effective because they do not consider all variables as equal candidates to be assigned to a register and because they must conservatively avoid difficulties that can arise from potential aliasing.

Heavy use of registers will also improve code density. Code space can increase greatly when each operation has multiple memory addresses, even of the form displacement(base). Load and store instructions in MIPS are at most 32 bits in length and are of five types: long immediate, absolute, displacement(base), (base,index), and base shifted by n, $0 \leq n \leq 15$. These addressing modes require at most one ALU operation on data in general registers and immediate data from the instruction. The last three forms are less than 32 bits and may be packed with a possibly unrelated ALU or shifter operation. All MIPS instructions, although they may be made of up to two instruction pieces, are 32 bits in length and execute in one data memory cycle time. Because the two instruction pieces are disjoint they may be used for parts of two independent computations.

Orthogonal immediate fields can additionally increase the code density as they reduce the number of those loads which are executed to load a register with a constant. In the MIPS instruction format every operation can optionally contain a four bit constant in the range 0-15 in place of a register field. Additionally, a move immediate instruction will load an 8-bit constant into any register. Table 2-1 contains the distribution of constants (in magnitudes) found in a collection of Pascal programs including compilers and VLSI design aid software.

Absolute Value	Percentage
0	24.8%
1	19.0%
2	4.1%
3 - 15	20.8%
16 - 255	26.8%
> 255	4.5%

Table 2-1: Constant distribution in programs

The large majority of the constants in the range 16 - 255 represent character constants. Most of the large constants (>255) represent values that are directly related to the function of the program and do not relate to constants in other programs. Thus, a 4 bit constant should cover approximately 70% of the cases; the special 8 bit constant will catch all but 5%. To obtain small negative constants two approaches are possible: provide for a sign bit in the constants or provide reverse operators that allow the constants to be treated as negative. MIPS uses the latter approach because it allows more constants to be expressed and eliminates the need for sign extension in the constant insertion hardware.

2.3. Condition codes and control flow

Many architectures have included condition codes as the primary method of implementing conditional control flow. Condition codes provide an implicit communication between two otherwise disjoint instructions. Condition codes can make life difficult for compiler writers, especially in nonorthogonal architectures. Even in reasonably orthogonal architectures (e.g., the M68000), working with condition codes is not simple. Difficulties occur largely because condition codes are side effects of instruction execution.

Condition codes are typically used for conditional control flow breaks, evaluation of boolean expressions, overflow detection and multiprecision arithmetic. Table 2-2 shows a typical set of features associated with condition codes and various architectures with these features. After discussing the disadvantages of condition codes from a hardware viewpoint, we will examine their use in each one of these instances and propose alternatives.

	Has condition code		No Condition code
	Set on moves	Set on operations	
Conditional set		M68000	MIPS
Branch Access	VAX	360	PDP-10

Table 2-2: Condition code operations

Condition codes are difficult to implement primarily because they are an irregular structure. As opposed to registers, where the references of updates of objects are explicit, condition codes are updated as a side-effect. In architectures where some but not all instructions set the condition code, additional complexity is introduced in decoding and condition code control. Because of these irregularities, implementing condition codes is particularly painful on a heavily pipelined machine. With architectures which set the condition code heavily (such as the VAX), the hardware can assume that all instructions set the condition code, then the hardware would force branch instructions to wait until the pipe has cleared. This may result in some performance loss when the instructions immediately before the branch do not affect the condition code. The implementation becomes substantially more difficult when the set of instructions that affect the condition code is large but not close to the set of all instructions (as in the 360/370). In this case, a significant amount of hardware is needed to determine whether condition codes are affected by a particular instruction. Because fewer instructions set the condition code, when dealing with a branch instruction, a designer is tempted to find the last instruction that affected the condition code. However, because the set of instructions that affect the condition code may be nontrivial to determine, finding the last instruction is extremely difficult.

2.3.1. Conditional control flow breaks

Typically, condition codes are used to implement conditional control flow breaks. A compare instruction (or other arithmetic instruction) is used to set the condition code; a conditional branch instruction uses the condition code to determine whether to take a branch. The most obvious disadvantage of this approach is that two instructions are required to effect the usual compare and branch. This is not a major disadvantage since a single compare and branch instruction would take longer to execute and more instruction bytes to encode. From a hardware implementation viewpoint, it is also useful to know the branch condition explicitly.

There are two primary *hypothetical* advantages for condition codes:

1. They save instructions by allowing branches to use the results of computations that are already done.
2. Condition codes model three way (<,>,=) branches.

Table 2-3 contains empirical data that shows that the number of instructions saved by condition codes is so small as to be essentially useless. Three way branches are a Fortran anomaly introduced into the language because of the architecture of the 704. In any event, we believe that the vast majority of the three way branches in Fortran contain only two branch destinations.

To implement conditional branches on a condition code machine, first the condition, and then is set then a conditional branch is used. Most architectures set condition codes by ALU operations only; the VAX sets the condition code on all move operations.

MIPS and a few other architectures such as the Cray-I and the PDP-10 implement conditional control flow using compare and branch instructions. In MIPS, all instructions including the compare and branch instructions take the same amount of execution time. Thus, the comparison is to some extent free. In cases where explicit comparisons (the dominant case : see Table 2-3) are needed on condition code machines, MIPS' approach actually *saves* instructions.

MIPS supports conditional control flow breaks using a compare and branch instruction with one of 16 possible comparisons. The 16 comparisons include both signed and unsigned arithmetic (e.g., X<Y) as well as logical (e.g., X and Y = 0) comparisons. Most conditional control flow breaks can be implemented with one comparison and the displacement(base) branch address available in this instruction. This, combined with the fact that compare and branch instruction executes in one data memory cycle (as do all other instructions), makes this instruction a very fast means of implementing conditional control flow breaks.

Compares without condition codes	2469
Compares saved using condition codes set by operators only	1.1%
Compares saved using condition codes set by operators and moves	733
Moves used to set condition code	706
Savings for condition codes set by operators and moves	1.1%

Table 2-3: Use of condition codes

2.3.2. Evaluating boolean expressions

Handling boolean expressions with numeric comparisons can be difficult in many architectures. For example, consider:

```
Found := (Rec = Key) OR (I = 13);
```

where each variable except `Found` is an integer. On a condition code machine where the condition is accessible only by conditional branches (e.g., the VAX), typical code sequences are given in Figure 2-1. Clearly, early-out evaluation is much better. early-out evaluation is frequently usable either because the language explicitly permits it or because the absence of side effects makes it possible. The high percentage of branches in this code is perhaps the most disturbing point. The cost of branches on modern pipelined architectures is far more than the cost of a typical compute-type instruction.

```
Full Evaluation                    Early-out Evaluation

  str  0,r1                          str 1,Found
  comp Rec,Key                       comp Rec,Key
  bne  L                             beq D
  str  1,r1                          comp I,13
L:comp I,13                          beq D
  bne  D                             str 0,Found
  str  1,r1                        D:
D:str  r1,Found

8 static instructions              6 static instructions
2 branches                         2 branches
Average of 7 instructions          Average of 4.25 instructions
   executed                           executed
Always executes 2 branches         Executes one branch on average
```

Figure 2-1: Evaluating boolean expressions with condition codes

An improvement over this can be gained by including instructions that conditionally set values based on the condition codes (as on the M68000). With such instructions, the improved code sequences in Figure 2-2 can be used. Although the average dynamic instruction count is slightly higher for this instruction sequence, it would execute faster on almost all machines since it has no branches.

```
    comp Rec,Key
    seq  Found      ; R1 is set to bit that represents equal
    comp I,13
    seq  r1
    or   r1,Found

    5 static/dynamic instructions
    No branches
```

Figure 2-2: Boolean expression evaluation using conditional set

On an architecture without condition codes some other method is needed to set the values. One approach is to duplicate the code used for a condition code machine without a conditional set instruction. Instead, MIPS provides a powerful *Set Conditionally* instruction with the same 16 comparisons found in conditional branches. This instruction performs a comparison, and sets a register to zero or one based on the result. Using this instruction, the code sequence for our example is shown in Figure 2-3.

```
    seq Rec,Key,r1
    seq I,13,r2
    or  r1,r2,Found

    3 static and dynamic instructions
    No branches
```

Figure 2-3: Boolean expression evaluation using set conditionally

In addition to boolean expressions that are assigned to variables, boolean expressions appear in conditional tests. In this case, early-out evaluation will result in similar numbers of instruction counts. The conditional set approach will be superior only in the case of complex expressions (more than one boolean operators).

Average operators/boolean expression	1.86
Boolean expressions ending in jumps	80.9%
Boolean expressions ending in stores	19.1%

Table 2-4: Boolean expressions

Table 2-4 shows the distribution of boolean expression types for our Pascal data set. Table 2-5 shows the number of operations needed per boolean operator to evaluate boolean expressions using different architectural support.

Compare/Register/Branch instructions per boolean operator	Static	Dynamic
Set Conditionally instruction	2/1/0	2/1/0
CC and set conditionally based on CC	2/3/0	2/3/0
Only CC and branch, full evaluation	2/2/2	2/2/2
Only CC and branch, early-out	2/0/2	2/0/1.5

Table 2-5: Operations needed to evaluate a boolean expression

Without a conditional set operation, evaluation of a boolean expression to be stored will require an extra assignment. When evaluating a boolean expression for a conditional branch, the branch instruction will be part of the normal evaluation in the case of a condition branch-branch evaluation but will be required in addition to the evaluation when conditional set evaluation is used. Using the data from previous tables, Table 2-6 shows the effectiveness for conditional set assuming that register operations take time 1, compares take time 2, and branches take time 4.

Type	Operations Evaluation	Full	Early-out
Store	Set conditionally/no CC	9.3	9.3
Store	CC/conditional set	14.9	14.9
Store	CC with only branch	27.9	20.5
Jump	Set conditionally/no CC	13.3	13.3
Jump	CC/conditional set	18.9	18.9
Jump	CC with only branch	26.9	19.5
Total	Set conditionally/no CC	12.5	12.5
Total	CC/conditional set	18.0	18.0
Total	CC with only branch	26.9	19.7
Improvement	Conditional set/CC	33.0%	8.6%
Improvement	Set conditionally	53.5%	36.5%

Table 2-6: Cost of evaluating boolean expressions

2.3.3. Overflow and multiple precision arithmetic

In a machine with condition codes, overflow bits may need to be explicitly tested to trap to exception handlers, as is required on the M68000. This results in significant performance degradation if each result is tested via conditional traps, or a loss in reliability if no or few tests are made. Other machines trap automatically via hardware mechanisms. MIPS traps if overflow detection is enabled and stores the trap type in a *surprise register*. This is discussed in greater detail in Section 3, Systems Support.

Carry bits are mainly used for multiprecision arithmetic. This is most important for 8 and 16-bit machines and to a lesser degree for larger machines without floating point hardware. MIPS is in the second category. For intensive floating point applications, the use of a numeric coprocessor such as the Intel 8087 is envisioned. For more common occasional use, multiprecision arithmetic can be synthesized with 31 bit words. This does not cause problems unless precisions of 2w-1, 2w, 3w-2,... bits (where the wordsize is w) are required. Multiprecision numbers are usually smaller than an integral multiple of the wordsize, such as fractions in doubleword floating point numbers.

3. Architectural Support for Systems

Central processors are usually only one component of a computing system that includes memory, mass storage, network subsystems, and perhaps other central processors. The architecture of the central processor should assist its integration into such systems. Since multiprocessing is a major component of all modern computing systems, the processor architecture should permit reliable implementation of multiprocessing by efficiently supporting:

- Memory management: virtual memory support with protected address spaces (the address spaces of a process should be protected from unauthorized access by other processes).
- Context switching: the processor should correctly save the state of a process when an exception occurs.

Memory management can be supplied entirely by external hardware for some applications. However, architectural support by the central processor is needed if the address space of a process is not fixed at the beginning of execution, for example, if stacks are allowed to grow into previously unallocated memory, or if demand paging is desired. Unless the state of the CPU is properly saved when a memory fault occurs, restarting an instruction following a fault may be difficult to do reliably. For a load/store architecture, this state saving problem is minimal; a faulting instruction can simply be re-executed. By contrast, processors with autoincrement addressing modes must deal with the problem of exactly when an address register is modified as a side effect of a memory reference.

Context switching can occur for several reasons, including memory faults, external interrupts, arithmetic exceptions, or system calls. The cost of a context switch is the time required to save the state of a process plus the time to select a new process as determined by the cause of the interrupt. The major portion of the process state for a general register machine is its register set. The time required to save the registers can be reduced by allowing each process access to only a subset of the registers; this technique is used by processors with multiple register banks or more recently by RISC with overlapping register banks. In addition to saving processor state, the processor must execute the appropriate interrupt service routine. Usually this routine is selected from a table indexed by a small integer that represents the cause of the interrupt.

An alternative architectural approach is to explicitly include the reason for an interrupt, which we term the surprise code, as part of the state of the machine. The surprise code contains two parts. The processor supplied part contains information about the pipestage that caused the interrupt and the type of interrupt. Surprise codes can indicate page faults, internal traps (e.g. arithmetic overflow), external interrupts (caused by peripheral devices), and internal traps raised by user instructions. The information part of the surprise code is supplied by the interrupting instruction or external device. It can be used to supply additional information about the reason for the interrupt. In this scheme, all traps, interrupts, and faults are handled orthogonally.

There are several advantages to this approach:

- Because all interrupt types are handled in the same manner (with the possible addition of turning off writes to memory), the hardware is substantially simplified.
- A single central interrupt service routine can interpret the surprise code and dispatch to the required subroutine. This code, common to all interrupt service routines, can be included in the central service routine. In MIPS, for example, the central service routine would save the addresses of the three instructions currently being executed and some or all of the general registers, then adjust the privileged state of the CPU if necessary.
- The surprise code can be fairly large and its interpretation can be performed by software. Since the surprise code is a part of the machine state, it can be examined by interrupt service routines, and a single routine can deal with a variety of interrupts. Processors using vectored interrupts can often jump to a common interrupt service routine after first loading a register with an interrupt specific value.
- A fixed size table of addresses of interrupt service routines is not needed.
- Extraneous interrupts, for example from input/output devices that the operating system does not know about, can easily be disregarded.

The chief disadvantage of the surprise code approach is that it is slightly slower than using a direct interrupt vector. Several additional instructions may be needed to interpret the surprise code. Moreover, since it is part of the state of the machine, the surprise code must be saved whenever an interrupt occurs, if nested interrupts are possible. In MIPS, the surprise code is part of the program status word, which includes the privilege state and arithmetic overflow trap enable, so that no significant additional cost is incurred for saving the surprise code.

4. Compiler support for fast architectures

Compilers can help create speedier architectures by relaxing their requirements on the instruction set. For example, consider an instruction to perform function *F* that is fairly complicated with respect to its demands on the hardware. The operation *F* may be a perfectly natural instruction in terms of a variety of source languages and compilers. However, incorporating *F* as an instruction in the architecture may be a bad decision for several reasons:

1. When *F* is implemented, it may be slower than a custom tailored version of *F*, which is built from simpler instructions.
2. When *F* is added to the architecture, all the other instructions may suffer some performance degradation (albeit small in most cases). This phenomenon has been called the "n+1 instruction"phenomenon and has been observed in several VLSI processor designs [10, 5].

Of course, whether or not *F* slows down the overall performance is dependent both on an implementation of the architecture and on the frequency of use of *F*. Instances of this type of behavior can be seen in the VAX 11/780. For example, the Index and CallS instructions in most instances are slower than obvious simpler code sequences [4, 12]

A clear case can be made for eliminating more complex instructions, where the speed benefits and usefulness of *F* are dubious. By advancing the case one step further, one can argue that even "natural" instructions should be examined for their usefulness, and performance when measured against possibly faster, customized sequences of simpler instructions.

4.1. word-addressed machines

Most newer computer architectures, from micros up to large machines, have supported some sort of byte addressing. The primary reason for this is that character and logical data is often handled as bytes (at least when it is in an array) and byte addressing simplifies code generation and makes the machine faster.

While it is true that byte addressing makes code generation simpler, it is not at all clear that it has a positive influence on overall performance. Memory interfaces that must support byte addressability as well as word (and probably halfword) addressability are significantly more complicated. Our estimates are that a byte addressable memory interface would add from 15 to 20% additional overhead to the critical path of the MIPS VLSI processor. Because many processors assume that operand fetch times are constant (ignoring issues such as caches), all operand fetches will pay the cost of this overhead. Most of this overhead results from the necessity of doing byte insert or extract operations (we assume only a single memory access is needed and do consider the complexity of each extra read needed to implement byte stores). Other overhead comes from the added control complexity needed to implement byte addressing. If all instructions have byte and halfword formats to preserve orthogonality, the size of the control structure may be greatly increased. The overhead estimate that we give below ignores the cost associated with this extra hardware (i.e., the control hardware) and the resultant larger chip area, which will cause additional performance degradation.

An alternative is not to include byte addressability. The performance impact of such a choice depends on three factors:

1. The estimated cost of supporting byte addressing in terms of added operand fetching time for all operands.
2. The percentage of occurrences of byte-sized objects.
3. The cost of performing the byte operations on a word-addressed machine.

We will address the latter two issues. Tables 4-1 and 4-2 contain data on storage references in terms of loads and stores. Table 4-1 allocates all objects as words unless they occur in a packed structure. Table 4-2 allocates all characters and booleans as bytes. Block movements of data are not included as any either byte or word references. The source for the data is a collection of Pascal programs including compilers, optimizers, and VLSI design aid software; the programs are reasonably involved with text handling and little or no compute. intensive (e.g. floating point) tasks are included. The global activation records of the word-based allocation version average 20% larger.

All data references -	71.2 % loads, 28.7 % stores
8 bit loads	2.6 %
32 bit loads or larger	68.6 %
8 bit stores	2.6 %
32 bit stores or larger	26.2 %
Character references -	66.7 % loads, 33.3 % stores
8 bit character loads	14.7 %
32 bit character loads	52.0 %
8 bit character stores	21.5 %
32 bit character stores	11.8 %

Table 4-1: Data reference patterns in word-allocated programs

All data references -	71.2 % loads, 28.7 % stores
8 bit loads	6.6 %
32 bit loads or larger	64.6 %
8 bit stores	5.9 %
32 bit stores or larger	22.9 %

Table 4-2: Data reference patterns in byte-allocated programs

The major observations we can make from this data are

- Objects allocated as full words dominate byte-allocated objects.
- Character reference patterns have a much higher percentage of stores than that of non-character references.
- When unpacked character data is allocated in words, most of the data references are to the unpacked characters.

To evaluate the effectiveness of byte processing using a word-addressed machine, we need to examine two questions:

1. How can the compiler help us?
2. What instructions are available for extracting and isolating words from bytes?

The compiler can help by attempting to transform character at a time processing to word at a time processing. Since many of the operations that deal with characters concern copying and comparing strings, the potential benefits are substantial. This is an interesting code optimization problem whose benefits and difficulties are nonobvious.

To explore the cost of character processing with only word addressing, we will look at the instructions in the MIPS instruction set and the typical code sequences. Given these and estimates of the overhead associated with byte addressing support, we can get a rough performance comparison.

Although MIPS does not have byte addressing, it has special instructions for byte objects in packed arrays. Packed byte arrays are typically accessed via a byte index from the beginning of a word-aligned array; byte pointers can be regarded as the special case where the array is located at memory location 0.

MIPS has four instructions to support byte objects: load and store base shifted and insert and extract byte operations. Load and store base shifted can be used for accessing packetl arrays of 2**n bit objects, where $0 \leq n \leq 15$ (i.e., bits through words). These instructions take a packed array pointer consisting of a 32-(5-n) bit word address in the high order bits and a packed array index in the low order n bits. This word is loaded/stored by shifting the packed array pointer n bits and reading/writing from that location as in every other load/store.

Bytes are accessed with insert/extract instructions. These insert or extract the byte specified by the low order two bits of a byte pointer. In the case of extract the byte pointer may be anywhere; for insert the byte pointer must be moved to a special register.

Loading a byte can be performed in two steps: first load the word containing the byte, then extract the byte from one of four locations within the word.

If a byte pointer is in R0 (the high order 30 bits contain a word address),then the following MIPS code sequence is equivalent to a load byte instruction:

```
    ;word at (r0/4) into r1
ld  (r0>>2),r1
    ;extract byte from r1 into r1
xc  r0,r1,r1
```

Thus, one memory access instruction and one ALU instruction are required to fetch a byte.

Storing into a byte array requires between two and three steps:

1. Fetch into a register the word that contains the destination byte (this step is often not needed because the word is in a register).
2. Replace the desired byte within the word register with the source byte.
3. Store the updated word into memory.

With the aid of an insert character instruction, a MIPS code sequence for store byte becomes

```
    ;word at (r0/4) into r2
ld  (r0>>2),r2
    ;r1 into byte selector lo
mov r1,lo
    ;low order byte of r3 into r2
ic  lo,r3,r2
    ;return word in memory
st  r2,(r0>>2)
```

This code sequence utilizes 1-2 memory reference instructions and two ALU instructions.

The cost of addressing operations using byte-addressed MIPS with/without overhead (15%) and using MIPS byte insert/extract instructions is shown in Table 4-3. We assume that the cost of an instruction is equal to the number of clock cycles needed to execute that instruction (or instruction piece). We also assume that non-array data can be accessed with the displacement field present in load and store instructions. Because word displacements are larger, word addressing has an advantage when displacements are too large for the byte-addressed case.

Operation	Cost with byte operations	Cost with overhead	Cost with MIPS operations
load from array	4	4.6	6
store into array	4	4.6	8-12
load byte	6	6.9	8
store byte	6	6.9	10-18
load word	4	4.6	4
store word	4	4.6	4

Table 4-3: Cost of various byte operations

Operations	Word-allocated cost	Byte-allocated cost
byte loads on MIPS	.156	.476
byte stores on MIPS	.208-.312	.486-.75
word loads on MIPS	2.744	2.584
word stores on MIPS	1.048	.916
Total loads and stores on MIPS	4.156-4.26	4.162-4.426
byte loads on byte-addressed MIPS	.12	.396
byte stores on byte-addressed MIPS	.12	.347
word loads on byte-addressed MIPS	3.202	2.972
word stores on byte-addressed MIPS	1.205	1.053
Total loads and stores on byte-addressed MIPS	4.647	4.768
Byte addressing performance penalty	9% - 11.8%	7.7 - 14.6%

Table 4-4: Cost of byte and word-addressed based architectures

Table 4-4 analyzes the cost of word based addressing versus byte based addressing. The cost of addressing is computed from the cost of each type of addressing times its frequency. Because we ignore the extra addressing range of word offsets, use a minimum overhead factor, and ignore the extra read required to implement byte stores, these figures should be regarded as minimum improvements attributable to word based addressing. When all the factors are considered, improvements in the range of 20% for

word-allocated programs and 23% for byte-allocated should be expected.

4.2. Applying better compiler technology

Another approach to faster, cheaper architectures is to require more software support in the compiler. This is an attempt to the shift burden of the cost from hardware to software. The shifting of the complexity from hardware to software has several major advantages:

- The complexity is paid for only once during compilation. When a user runs his program on a complex architecture, he pays the cost of the architectural overhead each time he runs his program.
- It allows the concentration of energies on the software, rather than constructing a complex hardware engine, which is hard to design, debug, and efficiently utilize. Software is not necessarily easier to construct, but VLSI-based implementations make hardware simplicity important.

Naturally, when attempting to place more emphasis on the software aspects, the overall performance must be improved. We also hope to improve the cost effectiveness by using compiler technology that is more powerful but not much more complex.

4.2.1. Software-imposed pipeline interlocks

The interlock hardware in a pipeline processor normally provides these functions:

- If an operand is fetched from memory, the interlock mechanism delays subsequent instructions that attempt to reference the data until the operand is available.
- The pipeline is cleared after the execution of a flow-control instruction. If a branch is taken, the interlock mechanism guarantees that the next instruction executed is the instruction at the destination of the branch.
- Arithmetic operations may require different amounts of time to execute, e.g., multiply and divide. The interlock hardware will prevent the next instruction from executing if there are source-destination or destination-source dependencies.

We will discuss a software-based implementation of the first two functions and present an algorithm for the first function and some empirical data on its performance.

In a processor with interlock hardware that provides the functions listed above, pipelining can be seen as an optimization implemented by hardware. Basically, the hardware will execute the program faster, subject to the interlocks which prevent illegal optimizations. This approach allows the compiler (or other user of the machine-level instruction set) to make simple assumptions about the execution of individual machine instructions.

An alternative approach is to move these optimizations from hardware to software. In that case there is no hardware interlock mechanism. Instead, the functions described above have to be provided by software, either by rearranging the code sequence or by inserting no-ops. This approach has the potential of producing code which executes faster, at the expense of the additional effort required to reorganize the instruction stream. no-ops will only explicitly delay the execution as compared to the invisible delays imposed by the hardware in an architecture with interlocks.

Such a reorganization scheme makes use of knowledge about the interdependencies of the individual instructions. The techniques developed for code optimization can be adopted to handle the requirements of the reorganization algorithms. The MIPS architecture employs the approach outlined here; there are no hardware interlocks. The current scheme provides the reorganization as a post-processing of the code generator's output. This reorganizer performs several major functions:

1. It takes the pipeline constraints into account and reorganizes the code to avoid interlocks when possible, and otherwise inserts no-ops.
2. It packs instruction pieces into one 32 bit word.
3. It assembles instructions.

Thus, the reorganizer also works on programmer-written assembly language code and reorganizes, packs, and assembles it.

Since the code reorganization process is part of every compilation, we must concentrate on solutions which have acceptable run-time performance and still produce good results in most cases. Finding an optimal code sequence is very expensive, as the problem is NP-hard [6]. All code reorganization is done on a basic block basis. The algorithm is discussed in detail in [6].

The basic steps in the algorithm are

1. Read in a basic block and create a machine-level dag that represents the dependencies between individual instruction pieces.
2. Given the set of instructions generated so far, determine sets of instructions that can be generated next.
3. Eliminate any sets that cannot be started immediately.
4. If there are no sets left, emit a no-op and return to step 2. Otherwise, choose from among the sets remaining.

The choice of the next instruction to be scheduled (step 4 above) is made heuristically from the set of legal instructions. Typically, an instruction that fits in a hole in a nonfull instruction is preferred; this provides the instruction packing.

When determining which sets of instructions may be processed, the reorganizer must examine both the interlocks from earlier instructions and register usage in parallel dags. The use of registers in parallel dags partially determines what set of code reorderings are possible. The algorithm must also avoid reordering loads and stores that might be aliased.

All branches in MIPS are delayed branches with a single instruction delay. If instruction i is a branch to L and the branch is taken, then the sequence of instructions executed is i, $i+1$, L. There are three major schemes for dealing with delayed branches of delay n:

1. Move n instructions from before the branch till after the

branch.

2. If the branch is a backwards loop branch, then duplicate the first n instructions in the loop and branch to the n+1 instruction.
3. If the branch is conditional, move the next n sequential instructions so they immediately follow the branch.

Of course, if the branch is conditional, the outcome of the test must not depend on any of the moved instructions. Often the front end of the compiler is able to handle delayed branches better than the reorganizer; in this case it emits a pseudo-op which tells the reorganizer that this sequence is not to be touched. The branch delay optimization algorithm and its performance are discussed in [7].

Figure 4-1 shows how a code fragment is affected by reorganization. In this case, it is assumed that r2 is "dead" outside of the section shown, therefore it can be modified even if the branch in line 2 is taken. Note also that the store instruction is not moved as it effects memory.

```
Legal Code                    Reorganized
with No-ops                   Code

ld 2(ap), r0                  ld 2(ap), r0
ble r0, #1, L11               ble r0, #1, L11
No-op                         No-op
No-op                         sub #1, r0, r2
sub #1, r0, r2                st r2, 2(sp)
st r2, 2(sp)
.
.
.
ld  3(sp), r5                 bra L3
add r5, r0                    ld 3(sp), r5 add r5, r0
add #1, r4                    add #1, r4
bra L3

L3: ...                       L3: ...
```

Figure 4-1: Reorganization, packing, and branch delay

To show the effectiveness of these optimizations, we ran versions of a program that does reorganization, packing, and branch delay elimination of three input programs. The input programs consist of an implementation of computing Fibbonacci numbers and two implementations of the Puzzle benchmark [2]. All the programs were written in C and compiled to instruction pieces by a version of the Portable C Compiler. The data in Table 4-5 shows the improvements in static instruction counts.

Optimization	Fibbonacci	Puzzle 0	Puzzle 1
None (no-ops inserted)	63	843	1219
Reorganization	63	834	1113
Packing	55	776	992
Branch delay	50	634	791
Total Improvement	20.6%	24.8%	35.1%

Table 4-5: Cumulative improvements with postpass optimization

5. Conclusions

We have argued that the most effective performance can be obtained by a design methodology that makes tradeoffs across the boundaries between hardware and software. This approach is just the opposite of some architectures that advocate extensive hardware support. Several experimental projects, including MIPS and RISC, are persueing the goal of a simplified hardware implementation.

We examine the issue of instruction set design to support the execution of compiled code. As opposed to "language-oriented" developments such as stack machines, we advocate two major alterations: the use of load/store architectures and the absence of condition codes. Both of these design alternatives have two major advantages: simpler hardware implementation, and potentially higher individual instruction efficiency.

Second, we discuss the issue of supporting the construction of systems by providing the necessary primitive functions in hardware. We note that in many cases, the minimum functionality is not available. This limits the type of design approach that can be used. We consider the complete support of page faults and interrupts in detail, and discuss the disadvantages of approaches that provide extensive system support services.

The issue of word-based versus byte-based addressing is explored. Based on a set of empirical data, we conclude that for many applications architectures will have higher performance with word addressing. Word based addressing gains its advantages from two primary points: it has a lower overhead associated with each fetch or store, and word references occur much more frequently than byte references. To make a word based approach feasible, special support for accessing bytes (as in the MIPS instruction set) are needed.

Lastly, we discuss approaches that rely extensively on improved compiler technology. The compiler is taken into account both in terms of the instruction set and in terms of radically simplified hardware designs. The software imposition of interlocks is presented as an example of this approach.

We explore the concept of simultaneously dealing with the hardware implementation, systems requirements, and the compiler technology. Following such an approach may lead to a reduction in hardware, as opposed to additional hardware support. Whether such an approach is effective and efficient depends on a wide variety of factors, including resulting improvements in hardware performance, the usefulness of the feature, and the alternative cost without "hardware support." Using several specific examples, we have shown that such tradeoffs can produce significant improvements in overall performance.

References

1. Aho, A.V. and Ullman J.D.. *Principles of Compiler Design.* Addison Wesley, Menlo Park, 1977.

2. Baskett, F. Puzzle: an informal compute bound benchmark. Widely circulated and run.

3. Chaitin, Auslander, Chandra, Cocke, Hopkins, Markstein. Register Allocation by Coloring. Research Report 8395, IBM, , 1981.

4. *DEC VAX11 Architecture Handbook.* Digital Equiment Corp., Maynard, MA., 1979.

5. Hennessy, J.L., Jouppi, N., Baskett, F., and Gill,J. MIPS: A VLSI Processor Architecture. Proc. CMU Conference on VLSI Systems and Computations, October, 1981.

6. Hennessy, J.L. and Gross, T.R. Code Generation and Reorganization in the Presence of Pipeline Constraints. Proc. Ninth POPL Conference, ACM, January, 1982.

7. Hennessy, J.L. and Gross, T.R. Optimizing Branch Delays. Computer Systems Lab., Stanford University, 1981.

8. Johnson, S.C. A 32-Bit Processor Design. Tech. Rept. Computing Science #80, Bell Labortories, Murray Hill, April, 1979.

9. Lunde, A. "Empirical Evaluation of Some Features of Instruction Set Processor Architectures." *CACM 20*, 3 (March 1977), 143-152.

10. Murphy, B.T. and Molinelli, J.J. A 32-Bit Single Chip CMOS Microprocessor. Seminar given at the Integrated Circuits Laboratory, Stanford University, May 22, 1981.

11. Patterson, D.A. and Sequin C.H. RISC-I: A Reduced Instruction Set VLSI Computer. Proc. of the Eighth Annual Symposium on Computer Architecture, Minneapolis, Minn., May, 1981.

12. Patterson, D.A. and Ditzel, D.R. "The Case for the Reduced Instruction Set Computer." *Computer Architecture News 9*, 3 (October 1980).

13. Shustek, L.J. *Analysis and Performance of Computer Instruction Sets.* Ph.D. Th., Stanford University, May 1977. Also published as SLAC Report 205.