Tags and Type Checking in LISP: Hardware and Software Approaches

Peter Steenkiste and John Hennessy

Computer Systems Laboratory
Stanford University

Abstract

One of the major factors that distinguishes LISP from many other languages (Pascal, C, Fortran, etc.) is the need for run-time type checking. Run-time type checking is implemented by adding to each data object a tag that encodes type information. Tags must be compared for type compatibility, removed when using the data, and inserted when new data items are created. This tag manipulation, together with other work related to dynamic type checking and generic operations, constitutes a significant component of the execution time of LISP programs. This has led both to the development of LISP machines that support tag checking in hardware and to the avoidance of type checking by users running on stock hardware. To understand the role and necessity of special-purpose hardware for tag handling, we first measure the cost of type checking operations for a group of LISP programs. We then examine hardware and software implementations of tag operations and estimate the cost of tag handling with the different tag implementation schemes. The data shows that minimal levels of support provide most of the benefits, and that tag operations can be relatively inexpensive, even when no special hardware support is present.

1. Introduction

In statically typed languages like Pascal, type checking is done at compile-time. Languages like LISP do not require the user to specify the type of each data item so *run-time type checking* is required. Run-time type checking is implemented by adding a *tag* to each data item to encode the type of that item; operations on the data can then be type checked. On general-purpose processors, the tag is usually stored together with the data, or with a pointer to the data, in a single word. On LISP machines, the word length is often extended to accommodate tag bits, which are then handled with separate hardware. General-purpose machines must explicitly extract and compare tags when checking types and remove tags when operating on tagged data. Dynamic type checking, while chiefly concerned with tag operations, also involves support of generic arithmetic.

An earlier study [18] on the run-time behavior of LISP showed that LISP programs spend an average of one fourth of their execution time on handling tags with type checking turned off. This cost was the primary motivation for the creation of LISP machines with a *tagged architecture*. LISP machines with tag support have instructions that can operate on the tag and the data part of an item, without having to disassemble it with separate instructions, and they usually support tag checking operations in parallel with other operations. For example, an integer add and the type check on the two operands occur simultaneously. Adding run-time checking to primitive LISP operations slows down our set of LISP programs by 25%, so overlapping some or all of this testing with other operations can give a substantial speedup.

In this paper we first study the cost of tag handling for ten LISP programs that were executed on the MIPS-X reduced-instruction-set processor. We then describe a number of tag implementations, including both software schemes, which can be used on general-purpose architectures, and hardware schemes for LISP machines. We compare how effective they are at reducing the cost of tag handling; finally we discuss generic arithmetic in Section 4.

2. Portable Standard Lisp on MIPS-X

The data presented in this paper are based on measurements of ten Portable Standard Lisp programs that were executed on an instruction-level simulator for MIPS-X, a high-performance microprocessor [10]. MIPS-X is used as a typical example of a reduced-

The MIPS-X research project has been supported by the Defense Advanced Research Project Agency under contract # MDA903-83-C-0335

instruction-set processor. An advantage of using a RISC architecture in this type of study is that one can measure both instruction counts and execution time easily, since the latter depends directly on the former (ignoring cache misses). The programs we have studied include a compiler front-end, a garbage collector, and a rational function evaluator, and three of the larger Gabriel benchmarks [7]; together the ten programs contain about 11000 lines of LISP code, without comments. Details about the benchmarks appear in the Appendix. Portable Standard Lisp [8, 9] is a small, efficient LISP dialect. In the remainder of this section we discuss the PSL implementation on MIPS-X.

2.1. The implementation of tags

The PSL implementation on MIPS-X uses a 5 bit tag that is stored in the most significant part of the word; the remaining 27 bits contain a pointer to the data. For some data types, the data item contains immediate data, e.g. symbols and integers that fit in 27 bits. There are four operations related to tags:

- *tag insertion*: given a piece of data, or a pointer, and its type (tag value), construct the data item,
- *tag removal*: given an item, extract the data item, that is, clear the tag and create a valid pointer or data item,
- *tag extraction*: given an item, extract the tag value,
- *tag checking*: given an item, test the value of its tag; this is implemented as a tag extraction followed by a conditional branch.

The tag value for positive integers is 0, and for negative integers, 31 (all 1's). As a result of this choice, the LISP representation for an integer is the same as its two's complement machine representation [9]. This means that integer arithmetic done on short (27-bit) integers without type checking can use the arithmetic instructions of the processor without any need for reformatting. This optimization speeds up all low level integer operations. Because of the special tag encoding for integers, type checking for integers is different from other data types (see Section 4.1). Testing for overflow for integer additions (and subtractions) can be implemented as a type checking operation: if we add two LISP integer items and overflow occurs, then the result will not be a LISP integer. This special treatment of integers is justified by their high frequency of use.

2.2. Run-time checking and generic operations

How much run-time type checking is done, and how it is done, strongly influence the number of tag checking operations that are executed. For this reason, we first optimized PSL run-time checking [19] to make its performance comparable to that of some newer, optimized LISP systems [3, 11]. In this section, we describe what data types are used in our test programs, and how type checking is done for those types.

For a lot of operations, run-time checking is equivalent to checking the tag of the operand. An important example is type checking on *list operations* such as *car* and *cdr*: the operand has to be a list, otherwise the operation is illegal. Type checking for a symbol also consists of a single tag check. Run-time checking for *vector accesses* is more complicated. Compilers for language like Pascal and C often allow the programmer to specify run-time bounds checking. In LISP, vector accesses with full run-time checking will not only do bounds checking, but will also check that the indexed object is a vector and that the indexing type is legal.

Because the type of the operands of an *arithmetic operation* is not known at compile time, the LISP run-time system has to deal with type conversion and has to pick an operator that matches the type of the operands. This generic arithmetic can be implemented by doing a type dispatch on the type of the operands, but integers are by far the most common type of numbers in LISP [24], and generic arithmetic can be speeded up by first specifically testing for integer operands, thus giving a fast result for the most common integer case. The expensive general sequence is only used if non-integers are involved. The integer tests and the integer operation are done inline.

Most LISP dialects define more data types than are used in our programs, but the data objects most actively used will be of the types we discussed (numbers, symbols, lists, or vectors). A lot of the other data types are also modeled after, or are implemented on top of one of the above types, for example: *structures*, *strings*, and *bit-vectors* [17]. Because the data types used in our programs, and the implementation of tag checking are both similar to what is found in other modern LISP systems, we expect that the numbers presented in this paper are representative for most LISP dialects.

3. Time spent on tag operations

In this section we look at how much time LISP programs spend on various tag operations. LISP usually requires run-time checking on all operations, but there are several important cases where these checks are not required. First, when the compiler can determine the type of an operand based on the program context [12], or when the programmer uses variable declarations or type specific operators [16, 13, 3], the type checking operations can be removed without affecting correctness or security. Second, many LISP compilers have a flag that determines whether the compiler will give priority to speed or to safety [17]. The importance of optimizing run-time type checking

cannot be accurately measured until we know the frequency of its occurrence in real programs. Because the amount of run-time checking depends on techniques to minimize the checks, we have collected data in two extreme situations: when no type checking is done, and when full run-time checking is done. A real LISP program will lie between these two extremes.

Adding run-time checking to our set of programs increases the execution time with 25% on average, but the slowdown for individual programs ranges from 6% to 88% (Table 1). Checking on list operations is responsible for most of the increase in execution time, but for *opt* and *trav*, the contribution of checking vector operations is significant, and *rat* does a fair amount of arithmetic.

	arith	vector	list	total
inter	0.63	0.00	19.04	19.68
deduce	0.09	0.00	12.27	12.36
dedgc	0.04	0.00	6.58	6.62
rat	4.85	0.00	13.69	18.54
comp	0.05	0.00	10.34	10.39
opt	2.68	11.76	27.99	42.43
frl	0.45	0.00	9.72	10.17
boyer	0.00	0.00	17.50	17.50
brow	0.03	0.00	19.91	19.94
trav	3.09	71.96	13.19	88.25
average	1.19	8.37	15.02	24.59

Table 1: Percentage increase in execution time when run-time checking is added

In this paper, the 'cost' without (with) run-time checking, is expressed as a percentage of the execution time of the programs without (with) run-time checking. The tags are implemented as described in Section 2.1, but changes in the implementation, like putting the tags in the low order bits, should not influence our results significantly.

3.1. Tag insertion

A tag has to be inserted each time when a new item is created. Inserting a tag in a data item when both the tag and the item are in a register costs two cycles: one to shift the tag to the most significant bits, and one to 'or' the tag and the item together. Because of our choice of tag values, no tag insertion is necessary when an integer is created. Figure 1 shows that the ten programs spend on average 1.5% of their time on the insertion of tags.

We will not discuss tag insertion any further in this document, both because it is not time critical, and because there is little possibility for improvement by simple changes in software or hardware. For example, keeping a *preshifted* list tag in a register (and thus reducing the cost of tag insertion for list cells to one cycle) would speed up our programs only 0.5%.

3.2. Tag removal

On MIPS-X, the tag has to be removed before the data part of an item can be used, except for integers. Removing the tag can be done in one cycle by masking it out with a mask kept in a register. Figure 1 shows that the programs without run-time checking spend 8.7% of their time on masking out tags. With run-time checking, the cost drops to 7%, because the total execution time has increased (due to time spent on extracting and checking tags) while the time spent on tag removal stays the same.

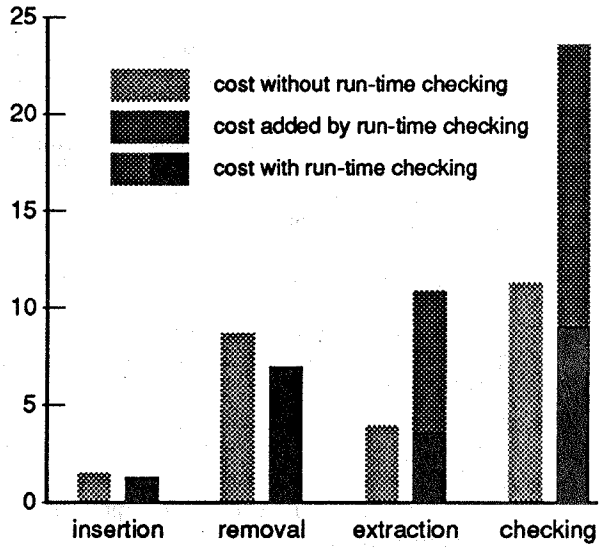

Figure 1: Percentage of time spent on all tag handling operations

3.3. Tag extraction

On MIPS-X, it is necessary to extract the tag of a data item before it can be compared with a known tag value to check the type of the item. Tag extraction can be done in a single cycle with a logical shift that places the tag in the low order part of the word. Figure 1 shows that 4% of the execution time is spent on tag extraction when no run-time checking is done. These tag extraction operations are part of type tests that are explicitly specified in the source program.

The dark histogram in Figure 1 shows the cost of tag extraction in programs with full run-time checking. The black part of the graph corresponds to the operations that were already present in the programs without run-time checking (light grey histogram), and

the dark grey corresponds to the operations that were added as part of the run-time checking; both components are expressed as a percentage of the execution time with run-time checking. Adding full run-time checking sharply increases the extraction cost and checking list operations is responsible for 80% of this increase. When we add run-time checking, the increase in the number of tag extraction operations, and thus in the number of tag checking operations, is about equal to the number of tag removal operations. This is what one would expect: with full run-time checking the type of each data item is checked before the item is used and using an item requires the removal of its tag.

3.4. Tag checking

Figure 1 shows that programs without run-time checking, spend 11% of their time on tag checking. The cost of tag checking includes the cost of extracting the tag, one cycle for a comparison, and possibly one or two cycles for unused branch delay slots. With full run-time checking almost 24% of the execution time is used for tag checking. Both with and without run-time testing, 95% of the tag checking operations are of the simple type (tag extraction followed by a comparison with a constant); the remaining tag checking operations are related to testing for integers and numbers.

3.5. Summary

In this section we saw that the total cost of tag handling is between 22% and 32% (Figure 1), depending on how much run-time checking is done. This cost is fairly constant across all programs - the standard deviations are 5.6% and 7.5% respectively - although the programs are widely different. In the following sections we will look at different tag implementations that reduce the cost of tag handling: in Section 4 we discuss software tag implementations that speed up integer testing, and in Sections 5 and 6 we look at schemes that can be used to reduce the cost of tag removal and tag checking. The hardware schemes proposed must be evaluated not only for improvements in instruction count, but also for potential negative impact on the processor's cycle time.

4. Software optimization of generic arithmetic and integer testing

If the tag is kept in the low order part of the word, integers should have tag value 0 for fast arithmetic, and integer testing is the same as for other data types. But when the tag is kept in the most significant part of the word, integers should get a tag that is the sign extension of their sign bit, and type checking for integers is more expensive, because positive and negative integers have different tags. This section first describes tag checking for integers, and then demonstrates how a special tag encoding can reduce the number of test-for-integer operations that are required for integer-biased generic arithmetic.

4.1. Support for integer testing

Testing for an integer, if the tag is in the most significant part of the word, can be implemented in a number of ways (assume 5 tag bits):

1. Extract the tag, then check for a positive integer; if that fails, check for a negative integer.
2. Sign extend the least significant 28 bits; the item is an integer if the result is equal to the original item.
3. Assuming that an arithmetic shift left gives a trap on overflow, doing an arithmetic shift left over 4 bits will trap if the item is not an integer.

The last method allows integer testing in a single cycle, but unfortunately, most architectures do not have a true arithmetic shift left. Recovering from a trap is also expensive, so this implementation would probably only be acceptable if a non-integer operand represents an error condition. Neither the first nor the second method require special hardware. It depends on the sign of the number which one is the fastest. The second method was used for the measurements in this paper, and it always costs 3 cycles. The first method is faster for positive numbers, and slower for negative numbers.

4.2. Reducing the cost of generic arithmetic

With integer-biased generic arithmetic (Section 2.2) the cost of generic arithmetic, averaged over the ten programs, is only 2%, but for computation intensive programs, this cost can be substantially higher. In *rat*, which is the most computation intensive program in our set, 8% of the execution time is spent on generic arithmetic. In this section we describe how the overhead of integer testing, which dominates the cost of generic arithmetic for simple integer operations, can be reduced by using a special tag encoding.

A generic integer add takes 10 cycles: 9 cycles for type and overflow checking, and 1 for adding. The reason for this high cost is that 3 type checking operations are required: two for the operands, and one to check the result for overflow. If we assign tag values in such a way that the sum of two non-integer tag values can never result in an integer tag value, then we can add the numbers, and do all the type and overflow checking with one single type checking operation on the result. This reduces the cost of a generic add of two integers to 4 cycles. For some other arithmetic operations the speedup would be smaller because more than one type test is necessary. With this special encoding of tags, the time spent on generic arithmetic drops to 1.6%, or a gain of 0.4% over the scheme with a

straightforward tag encoding; for *rat*, the speedup is about 2%.

The requirement on tag values can be met by using an extra tag bit; in the case of PSL, 6 tag bits instead of 5. With 6 tag bits it is possible to assign tag bits in such a way that the sum of two non-integer tag values, with possibly a carry in, can never result in a integer tag value without giving overflow. Another possibility is to keep 5 tag bits, and to reduce the number of tag values that are required by putting some typing information with the data.

This tag implementation has the disadvantage that it requires an extra tag bit. Not only does this reduce the address size by one bit, but it also means that this scheme cannot be used with tag implementations that allow only 2 or 3 tag bits (see Section 5.2). Since these tag implementations have a higher payoff, at least for our set of programs, we will not use the encoding described in this section. But if enough tag bits are available, and if generic arithmetic is important, then the special tag encoding deserves consideration.

5. Support for tag removal

In Section 3.2 we found that our set of LISP programs spend around 9% of their time masking out the tag of a data item in order to be able to use the data part. In this section we first discuss the need for tag removal. Then we show what can be gained if tag removal is not necessary, and finally we describe a number of tag implementations that eliminate the need for tag removal.

5.1. The need for tag removal

The data part of most LISP objects contains a pointer to the data, so it will always be used as an address. Two important exceptions are integers and symbols. We saw earlier that no tag removal is necessary for integers, and symbols are either compared with other symbols, without removing the tag, or they are used as an index in a symbol table, in which case the data part of the item is again used for the purpose of addressing a memory location. On a processor that drops the top 5 bits of 32 bit addresses before accessing memory, it is not necessary to mask the tag of an item explicitly when the data part of the item is used to access memory, which is, as we just argued, usually the case.

We changed the compiler so that no masking of the tag is done for items that are used as addresses, and we changed the simulator, so that it only uses the bottom 27 bits of an address when accessing memory. Figure 2 shows, for programs with no run-time checking, the decrease in instruction frequencies resulting from this optimization. When we compare the 'and' entry in Figure 2 with the 'removal' entry in Figure 1, we see that almost all masking operations have been removed. Part of the gain is undone by an increase in move instructions which is a consequence of the requirement that all load instructions have to be idempotent (repeatable). The increase in wasted cycles (no-ops and squashed instructions) results from the fact that fewer ALU instructions are available to fill delay slots after branches, loads and stores. Not having to mask the tag speeds up our programs 5.7% on average.

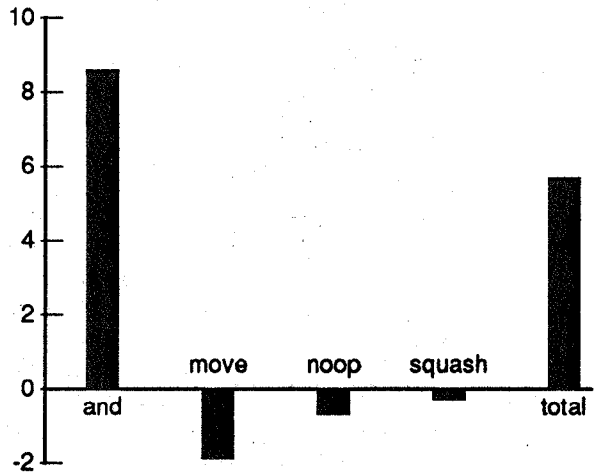

Figure 2: Reduction in instruction frequencies due to the elimination of tag removal

5.2. Implementation

Removal of the tag when an object is used as an address could be accomplished in either hardware or software. A hardware solution is available on machines where the address length is shorter than the word length and the tag can be placed in the upper bits of the word. Several architectures with this property exist (68000 [20], IBM/370 [23]), but most architectures that are designed today have a full 32 bit address space. LISP machines typically treat the tag and the data part of a data item as separate entities, so it is natural that the tag is automatically dropped when memory is accessed [1, 14].

Given a general-purpose processor like MIPS-X it would be possible to add special hardware that would blank out the 5 most significant bits of each address, before it is put on the address bus. This hardware could be controlled by a bit in the processor status word or special load and store instructions could be added to the instruction set.

It is possible to avoid the need for tag masking by making only changes in the software. On MIPS-X, most tag removals for addresses could be eliminated by using the two low-order bits of a word as a tag. MIPS-X uses byte addresses, but all memory accesses are

word aligned, hence, the bottom two bits of an address are dropped before the (word-addressed) memory is accessed. With two tag bits, three combinations are used to encode the most frequently used data types, leaving one combination as an escape, and thus eliminating tag masking for most memory accesses. On architectures that look at the two bottom bits of the address, this approach can still be used, but the compiler has to adjust the offset that is used to access the object so that the tag is eliminated, as is done in [15].

It is possible to avoid tag removal for more LISP types by using the *three bottom bits* for tag encoding. Even and odd integers get the tag values 000 and 100, so that integer arithmetic and indexing in word vectors will be fast, four tag values can be used for frequently used data types, and the values 011 and 111 (two bottom bits 1) are reserved as an escape. For data types with 3 bit tags, data objects will always be aligned on even or odd word boundaries. This is not a problem since list cells always require two words, and other types such as vectors and structures often come in larger blocks, so wasting a word to ensure the alignment is relatively cheap. Aligning list cells on double word boundaries might also be beneficial for caches with block sizes larger than two. The Lucid Common Lisp compiler uses the three bottom bits as tag bits for some architectures.

Not having to strip the tag of an item before accessing memory can save almost 6% in execution time, and several simple implementations exist. The software schemes that place the tag in the bottom two or three bits are very attractive: they avoid tag removal for almost all memory accesses without requiring special hardware, and they have the added advantage that the address space is not limited, which is important for large LISP systems.

6. Support for tag extraction and tag checking

Our programs spend between 11% and 24% of their time on tag checking (Figure 1), depending on how much run-time checking is done, so tag checking is an attractive candidate for optimization. Tag checking operations, or type dispatching, are required in a number of situations:

1. *run-time error checking*, for example type checking as part of a *car* operation,
2. *generic operations*, for example generic arithmetic, and
3. *checking operations specified at the source level*, for example the function *atom*.

Ignoring efforts to eliminate type and tag checking at compile-time using type deduction, the cost of tag checking can be reduced in two ways. First, by eliminating the need for tag extraction, thus reducing the cost of tag checking operations in all of the above categories. Second, by eliminating some tag checking operations completely in some of the categories. Because tag checking is a very simple operation, there is no room for software optimization, and both approaches require hardware changes. Both strategies are discussed in this section.

6.1. Eliminating tag extraction

We saw in Section 3.4 that tag extraction followed by a single eq/neq comparison is the most common form of type checking. The tag extraction operation can be avoided with a special conditional eq/neq branch that only tests the part of the word that contains the tag. This would eliminate almost 4% of the instructions, if no run-time checking is done, and about 10% of the instructions with full run-time checking. Some architectures, for example the VAX-11, have instructions that compare bit-fields directly, and that allow tag checking without explicit tag extraction. However, these instructions must be faster than a sequence of simpler instructions if they are to yield a performance improvement.

The special conditional branches can be implemented in a number of ways. First, it can be hardwired into the processor what bits will be used in the comparison. This is not very flexible, because the architecture, and not the software, determines where the tag bits should be placed in the word, but such an implementation is acceptable for LISP machines [14, 6, 21]. Second, the special conditional branch can take a mask as a third argument. This solution is more flexible, but it is expensive to implement because it introduces an instruction with 3 sources, complicating the data path and shortening the branch offset. Third, the bits can be specified by a mask that is set under program control. This can either be implemented by having a special mask register, or by using a specific general purpose register, in which case we would still need 3 reads from the register file. Note that an eq/neq comparison on part of a word, specified by a mask, is easy to implement and is also very fast.

6.2. Hardware tag checking

The cost of tag checking can be substantially reduced by providing special hardware that does some tag checking operations in parallel with other operations. It is not really possible to eliminate tag checking operations that are specified in the source code (category three of the beginning of this section), but LISP machines such as the Symbolics 3600 [14], TI Explorer [6] and SPUR [21] provide hardware support for tag checking operations that result from error checking on primitive operations, and from generic operations (categories one and two).

6.2.1. Error checking

Software tag checking used for error testing on primitive LISP operations can be eliminated by having special memory instructions that check the tag of the address during address calculation, and that trap if the tag does not have the expected value; the expected tag could be specified in a register, as an immediate, or in the opcode. The hardware test is limited to a simple tag check; for operations such as list accessing, this is sufficient, but for vector operations, range checking would still have to be done in software.

Hardware for parallel error checking is very simple if the tag location and the tag values, can be built into the hardware. For example, if the expected data type is encoded in the opcode, a PLA with the opcode and the operand tag as inputs, and a single output indicating success or failure, is all that is needed. The hardware becomes a lot more complicated if the tag implementation has to be left to the software.

For MIPS-X specifically, two difficulties would arise when adding parallel tag checking. First, the path to load/store data from/to the cache is critical; adding gates to that path to implement parallel checking and trapping, would slow down the processor. Second, the MIPS-X design tries to detect the arrival of an exception (e.g. traps and interrupts) as early as possible, to keep the exception handling out of the critical path of the processor. Adding an instruction that could explicitly trap, might reduce the time that is available to handle exceptions, thus slowing down the processor.

Run-time error checking on list operations accounts for between 0% and 12% of the execution time in our programs. Adding hardware to do this test in parallel with the address calculation would eliminate this cost, plus an extra 0%-4% because no tag removal would be required for memory accesses to lists. Extending the hardware to allow parallel type checking for other data types (vectors and structures), could give a speedup similar to the speedup for lists, depending on what data types are used in the program.

Note that the MIPS-X architecture naturally allows some overlap between an operation and its corresponding tag checking operation. With a *squashed delayed branch*, two instructions are executed while the branch condition is calculated and while the next instruction is fetched, and the effect of both instructions is cancelled if the the branch does not go [5]. An operation and its tag check will happen concurrently, if the branch condition is 'tag equal to expected tag' and if the operation is moved in a delayed slot of the branch.

6.2.2. Generic operations

Generic operations are operations that can handle data of different types. Examples are generic arithmetic and the function *equal*. Generic operations can be implemented in software by testing sequentially for the different possible data types, by dispatching on the type of the operand(s) (basically a case statement), or by a combination of the two, as described in see Section 2.2 for integer-biased generic arithmetic. It is possible to provides hardware support for generic operations at various levels. A first possibility is to test the type of the operands, while executing the operation, assuming that the operands are of the most common type. If the test fails, a trap is generated, and the operation is aborted; less common data types can then be handled in software. The implementation of arithmetic operations on SPUR [21] follows this strategy. This approach is very fast for the most common data type, but the treatment of other data types can be slow, depending on how fast traps are handled, and on how often the 'less common' case occurs. A floating point program that uses integer-biased generic arithmetic could well be slower than an integer-biased software implementation, because of the trap overhead. Trap handling can be simplified, at the expense of extra hardware, by the use of shadow registers that cache the operands [22].

By also providing hardware support for dispatching on the type of operands, it is possible to further speed up generic operations [14, 6]. The VLSI chip used in the TI Explorer II [2], for example, has a special on-chip memory for dispatch tables that can be used by the micro-code. Generic arithmetic operations are implemented by testing for the most common integer case while starting the integer operation, and if the test fails, the micro-code dispatches on the type of the operands. This approach is similar to the software implementation of generic arithmetic on general-purpose processors, but it will be faster, if the extra hardware is free, because it allows more parallelism.

The hardware described in this section would reduce the cost of generic arithmetic to 1.3%, down from 2%; all operands are integers, so a type dispatch is never needed. If a type dispatch is needed, the performance of the different tag implementations will vary strongly. When a type dispatch is needed for every arithmetic operation, that is, the inline test always fails, then with the MIPS-X software tag implementation, the overhead of the type dispatch would increase the average execution time by 2.7%. We expect that this number will be lower on a processor like the Symbolics, but it will be higher if less common data types cause a trap, as in SPUR.

Compile-time analysis can be used to reduce the cost of using the wrong bias. If compile-time analysis indicates that the operands are probably not integers, the compiler can generate code that invokes the general dispatch routine, or a (software) routine with a different bias. As the compiler is more and more successful at deriving data types, the 'less common' case will become more and more an exception.

7. Summary: what can hardware buy?

Table 2 shows what fraction of the cycles (after pipeline scheduling) would be eliminated in the ten PSL programs, with the various software and hardware tag implementations discussed in this paper. The programs were executed as described in Section 3, and the speedup is relative to the execution time with the straightforward tag implementation of Section 2.1. The two columns give results for programs with, and without run-time checking, and each column gives the speedup relative to the execution time for that column. The exact speedup will depend on the quality of the compiler, and on whether the compiler has been tuned for speed or for safety. Remember that adding full run-time checking in software slows down our programs by 25% on average.

The first three rows summarize the results for the tag implementations of Sections 5 and 6.1:

- row one corresponds to a software tag implementation in which the tag is kept in the bottom two or three bits of the word so no explicit tag removal is necessary before accessing memory; load and store instructions that ignore the tag bits in the address give the same speedup.
- row two gives the speedup if the processor has a special conditional branch that checks the tag without extracting it.
- row three gives the result if the implementations of rows one and two are combined.

These three implementations require either no hardware changes, or very simple hardware changes.

The fourth row shows the reduction in cycles if we had special hardware that traps if an arithmetic operation has non-integer operands, or if overflow occurs. The speedup is small because our programs are not computation intensive; note that the tag implementation of Section 4.2 would further reduce the gain of hardware support for generic arithmetic over a software implementation, but that tag implementation is not compatible with the implementation of row 1. The fifth row gives the speedup if tag checking on list operations is done with extra hardware in parallel with the address calculation (Section 6.2). In the sixth row, we assumed that parallel tag checking is not only possible for lists, but also for vectors and structures. These hardware additions are more substantial because they influence the control of MIPS-X.

The seventh row corresponds to a processor that has:

1. loads and stores that ignore the tag (row 1),
2. special instructions to check the tag without extracting it (row 2),
3. hardware for generic arithmetic (row 4),
4. loads and stores that do parallel error checking for all data types (row 6).

This is to the maximum amount of hardware support that can be added to MIPS-X without requiring a total reorganization of the processor. It would eliminate between 9% and 22% of the cycles. This savings should be compared with the software implementation of row one (6%-5%), and with row three (9%-14%), which only requires very limited hardware changes.

			no run-time checking	run-time checking
-1-	avoid tag masking (software)		5.7%	4.6%
-2-	avoid tag extraction		3.6%	9.3%
-3-	avoid masking and extraction		9.3%	13.9%
-4-	support generic arithmetic		0%	0.7%
-5-	avoid tag checking on list ops	check	0%	12.1%
		mask	0%	4.2%
		total	0%	16.3%
-6-	avoid error tag checking (lists+vectors)	check	0%	13.6%
		mask	0%	4.6%
		total	0%	18.2%
-7-	avoid masking		5.7%	4.6%
	avoid extraction		3.6%	2.9%
	avoid all error tag checking		0.0%	14.6%
		total	9.3%	22.1%

Table 2: Speedup in percent for different degrees of hardware support

More complex hardware support, such as microcode support for type dispatching, is possible. However, it would require a total reorganization of the MIPS-X architecture, and our results indicate that the payoff would be much less than the negative impact of the added hardware on cycle time and other performance measures.

SPUR [21] provides hardware support for tag handling corresponding to row seven, except that SPUR does not allow parallel checking on memory accesses other than list accesses (row 5 instead of row 6). As a result, the SPUR tag hardware would eliminate between 9% and 21% of the cycles in our programs, although the gain drops to between 4% and 16% of the cycles, if the software tag implementation of row one is used on MIPS-X.

8. Conclusion

In this paper we first looked at the cost of the various tag handling operations, if a straightforward tag implementation is used. We found that tag checking, which includes the cost of extracting the tag, is the most expensive operation, certainly if full checking at run-time is required (11%-24% of execution time). Tag removal, which is done before using the data, uses about 8% of the execution time, and tag insertion uses 1.5%.

Then, we looked at how the cost of tag handling can be reduced with different software and hardware tag implementations. The cost of tag removal can be reduced in software by putting the tag in the bottom 2 or 3 bits of the data word; this results in a speedup of about 5%. Speeding up tag checking requires special hardware. One possibility is to eliminate the need for tag extraction before a tag check. This is easy to implement, and it gives a speedup of 4% to 9%, depending on how many tag checking operations can be eliminated by the compiler. The two optimizations combined eliminate between 9% and 14% of the cycles. Another possibility is to have special hardware that does tag checking in parallel with memory access operations. This, together with the previous features, gives a speedup of between 9% and 22%, but it requires more complicated hardware, and the tag implementation has to be built into the architecture.

Appendix

The following set of 10 LISP programs were used to collect data:

- *inter*: a simple interpreter for a subset of LISP is used to calculate the Fibbonachi number 10, and to sort a list of numbers; adapted from "Lisp in Lisp" [25].
- *deduce*: a deductive information retriever for a database organized as a discrimination tree; adapted from [4].
- *dedgc*: the same program as *deduce*, but a copying garbage collector is invoked. The program spends about 50% of its time in the garbage collector.
- *rat*: a rational function evaluator that comes with the PSL system.
- *comp*: the first pass of the front-end of the PSL compiler.
- *opt*: the optimizer that was added to the compiler. It uses lists, and vectors.
- *frl*: a simple inventory system using the *frame representation language*.
- *boyer*: the boyer benchmark; a rewrite-rule-based simplifier combined with a dumb tautology-checker; benchmark published by Gabriel [7].
- *brow*: a short version of the browse benchmark; creates and browses through an AI-like database of units; benchmark published by Gabriel [7].
- *trav*: a short version of the traverse benchmark; creates and traverses a tree structure; uses structures which are implemented as vectors; benchmark published by Gabriel [7].

Table 3 gives the number of procedures, the number of lines of source code (without comments), and the number of MIPS-X machine instructions after compilation, for each program. Each program includes besides the user program, the LISP system modules, or parts of modules, that are used by the program.

	number of procedures	lines source code	words object code
inter	64	710	1533
deduce	100	900	3419
dedgc	116	1100	4112
rat	148	1900	6315
comp	220	2400	9466
opt	226	3500	11121
frl	198	2500	11802
boyer	84	1200	1793
brow	91	1000	2296
trav	78	810	1673

Table 3: Information on the 10 test programs

Acknowledgments

We thank the members of the MIPS-X group for their help and suggestions. Mark Horowitz provided helpful information about the MIPS-X implementation. The PSL system was developed at the University of Utah.

References

1. Bawden, A., Greenblatt, R., Holloway, J., Knight, T., Moon, D., and Weinreb, D. LISP Machine Progress Report. Memo No 444, MIT Artificial Intelligence Laboratory, August, 1977.

2. Bosshart, P., Hewes, C., Chang, M., and Chau, K. A 553K-Transistor LISP Processor Chip. Digest 1987 International Solid-State Circuits Conference, IEEE, New York, February, 1987, pp. 202-203.

3. Brooks, R., Posner, D., McDonald, J., White, J., Benson, E., and Gabriel, R. Design of An Optimizing, Dynamically Retargetable Compiler for Common Lisp. Proceedings of the 1986 Conference on LISP and Functional Programming, ACM, Boston, August, 1986, pp. 67-85.

4. Charniak, E., Riesbeck, C. K., and McDermott, D. V.. *Artificial Intelligence Programming.* Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1980.

5. Chow, P., and Horowitz, M. Architectural Tradeoffs in the Design of MIPS-X. Proceedings of the 14th Annual International Symposium on Computer Architecture, ACM, June, 1987, pp. .

6. Corley, C.J. and Statz, J.A. "LISP workstation brings AI power to a user's desk". *Computer Design 24*, 1 (January 1985).

7. Gabriel, R.P.. *Computer Systems Series.* Volume : *Performance and evaluation of LISP systems.* The MIT Press, 1985.

8. Griss, M.L. and Hearn, A.C. "A Portable LISP Compiler". *Software - Practice and Experience 11*, 6 (June 1981), 541-605.

9. Griss, M.L., Benson, E., and Maguire, G.Q. PSL: A Portable LISP System. Proceedings of the 1982 Symposium on LISP and Functional Programming, Pittsburgh, August, 1982, pp. 88-97.

10. Horowitz, M., Hennessy, J., Chow, P., Gulak, P., Acken, J., Agarwal, A., Chu, C.Y., McFarling, S., Przybylski, S., Richardson, S., Salz, A., Simoni, R., Stark, D., Steenkiste, P., Tjiang, S., and Wing, M. A 32b Microprocessor with On-Chip 2K Byte Instruction Cache. Digest 1987 International Solid-State Circuits Conference, IEEE, New York, February, 1987, pp. 30-31.

11. Kranz, D., Kelsey, R., Rees, R., Hudak, P., Philbin, J, and Adams, N. ORBIT: An Optimizing Compiler for Scheme. Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, ACM, Palo Alto, June, 1986, pp. 219-233.

12. Milner, R. "A Theory of Type Polymorphism in Programming". *Journal of Computer and System Science 17*, 3 (December 1978), 348-375.

13. Moon, D.A. *Maclisp Reference Manual.* MIT, Laboratory of Computer Science, 1983.

14. Moon, D.A. Architecture of the Symbolics 3600. Proceedings of the 12th Annual International Symposium on Computer Architecture, ACM, Boston, June, 1985, pp. 76-83. Also in SIGARCH Newsletter 13(3).

15. Rees, J., and Adams, N. T: A Dialect of Lisp or, or LAMBDA: The Ultimate Software Tool. Proceedings of the 1982 Symposium on LISP and Functional Programming, Pittsburgh, August, 1982, pp. 114-122.

16. Steele, G.L. Jr. Fast Arithmetic in MacLISP. Proceedings of the 1977 MACSYMA Users' Conference, July, 1977.

17. Steele, G. L. Jr.. *Common Lisp - The Language.* Digital Press, 1984.

18. Steenkiste, P., and Hennessy, J. LISP on a Reduced-Instruction-Set-Processor. Proceedings of the 1986 Conference on LISP and Functional Programming, ACM, Boston, August, 1986, pp. 192-201.

19. Steenkiste, P. *LISP on a Reduced-Instruction-Set Processor: Characterization and Optimization.* Ph.D. Th., Stanford University, March 1987.

20. Stritter, E., and Gunter, T. "A Microprocessor Architecture for a Changing World: The Motorola 68000". *IEEE Computer 12*, 2 (February 1979), 43-52.

21. Taylor, G.S., Hillfinger, P.N. Larus, J., et al. Evaluation of the SPUR Lisp Architecture. Proceedings of the 13th Annual International Symposium on Computer Architecture, ACM, Tokyo, June, 1986, pp. 444-452.

22. Ungar, D. *The Design and Evaluation of A High Performance Smalltalk System.* Ph.D. Th., UC Berkeley, March 1986. Technical Report UCB/CSD 86/287.

23. White, J. "LISP/370: A Short Technical Description of the Implementation". *SIGSAM 12*, 4 (November 1978), 23-27.

24. White, J. Reconfigurable, Retargetable Bignums. Proceedings of the 1986 Conference on LISP and Functional Programming, ACM, Boston, August, 1986, pp. 174-191.

25. Winston, P. and Horn, B.. *Lisp.* Addison-Wesley Publishing Company, 1981.