# Fast Dispatch Mechanisms for Stock Hardware

John R. Rose
Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043

(415) 336-1071
jrose@sun.com

## 1. Introduction

Common Lisp [Steele] can be characterized as an "almost object oriented" language, in that it supports a fixed set of generic functions (e.g., ELT) and a type structure which is open-ended enough to allow for extension of those functions. Alas, this extension cannot be done by application programmers, but is reserved for implementors. It is a common need for language designers and other researchers to extend generic operations; they are usually reduced to unsatisfactory package-system kludges. For example, both [Hillis] and [Sabot] define new sequence types while adapting Common Lisp to massively parallel hardware.

The proposed Common Lisp Object System [ANSI] (called "CLOS" below) provides a way to specify the extension of generic functions, but explicitly declines to include the means for extending functions (like ELT) which are generic over "built-in" classes. This is due to the fact that such functions must be assumed to use "special purpose" dispatch mechanisms, which typically amount to case analysis over a small, fixed number of argument types. The motivation for using such case analysis instead of a general, extensible dispatch mechanism is that the code (and the type representation) may be hand-tuned for maximum efficiency.

The present paper attempts to show how such case analysis may often be more efficiently done with extensible mechanisms, on standard hardware. Thus, there is little reason, as regards performance, for hard-coding the dispatch of Common Lisp generics such as the sequence functions. Moreover, the techniques presented here apply to the implementation of new generic functions, such as are defined with CLOS. Thus, a plausible implementation strategy for a Common Lisp with CLOS would be to use a single dispatch mechanism for both primitive and CLOS-defined generics.



The dispatch mechanisms presented here are usable for any language which must perform runtime type-checking and/or method selection, when the number of data types and operations is not too large (say, no more than a few thousand of each). This means our reasoning applies to fully object-oriented languages, as well as Lisp. In fact, many of these mechanisms are already in use in existing systems, as we'll see below. The main contribution of the present paper is the construction of a systematic design space which contains a range of fast mechanisms. These mechanisms are characterized by the extraction of a *tag* and a *key*, and the subsequent determination of a linear table and an index into it.

### 1.1. Disclaimers and Specializations

Note that dispatch mechanisms address the problem of *runtime* type checking. We do not inquire into performance of programs which bind types to expressions *before* runtime. For example, a fully-declared Lisp program (or type-inferred ML program) would not do much runtime dispatching, and so would not be greatly affected by dispatch speed. But in practice, runtime dispatching is common. Some programs cannot be statically typed. Some are too open-ended, perhaps due to an object oriented organization. Many programs use some sort of discriminated union type. Finally, most programs are not fully typed while under development (unless the language requires full type declarations). All such programs need *safety checks* in their execution, since type errors can arise at runtime. In most Common Lisp implementations, adding declarations to improve performance is difficult and error-prone, and full safety precludes reasonable execution speed. It is our hope that efficient runtime type-checking can reduce the need for declarations, and allow full runtime safety checks to be the norm.

We make our arguments using the SPARC [SUN] for concreteness, because it is so easy to count SPARC instruction cycles. Nevertheless, the same reasoning applies to any stock hardware architecture. Dependence on machine-specific features is noted where appropriate.

We also neglect the impact of virtual memory management on the choice of a dispatch mechanism. For systems with limited physical memory, a slow dispatch mechanism with a compact dispatch table representation

may have a higher overall performance. Our space could be enlarged to include compact as well as fast dispatch mechanisms, but it would be difficult to compare points within this space, since storage management performance is a poorly understood function of hardware, replacement strategy, and locality of reference. Therefore, we make the simplifying assumption that a system using our mechanisms has enough memory and locality to retain the advantages of fast dispatch. In practice, the speediest, most space-hungry dispatch mechanism may require too much virtual memory, and time must be traded off for space. It is the author's belief that such tradeoffs can often take place within the confines of the design space described in this paper.

### 1.2. The Rest of the Paper

In the remainder of this paper, we first survey the generic functions of Common Lisp. Next, a design space of high-performance dispatch algorithms is described, and choices within that space are related to issues of performance, CPU architecture, and data format. Finally, we address certain hard design and implementation problems which relate to our dispatch scheme. For example, we examine the CLOS CHANGE-CLASS primitive, which, like Smalltalk's BECOME message, modifies an object in place.

## 2. Genericity in Common Lisp

Many functions in Common Lisp exhibit genericity. Arithmetic primitives are all generic across the rich set of NUMBER subtypes: INTEGER, FLOAT, RATIONAL, COMPLEX, and more. The comparison operations EQUAL and EQL have several subcases, depending on the type of the arguments. There is a large group of "sequence functions" (e.g., ELT, MEMBER, UNION, REMOVE) which operate interchangeably on lists and vectors. The array accessor AREF needs to distinguish among several different array types (e.g., bit vector, string, matrix). Some functions will accept either an operand object or the name of one; e.g., INTERN accepts either a package or a package name. Certain specialized types like streams and hash tables almost certainly use generic functions, although the standard does not mandate subtypes to implement their variant behaviors.

Measurements of the instruction mix of Lisp programs show that a relatively small number of these operations are important because of their dynamic frequency. Small integer arithmetic is the most important. Function linkage and memory access (notably CAR and CDR) are also very significant. These common operations are also extremely sensitive to overhead because of their speed: Small integer addition, which takes one CPU cycle, can slow down by one or two orders of magnitude due to clumsy tag checking.

It may not be obvious at first blush, but the Lisp function calling sequence is a generic operation. The caller specifies any fixed number of arguments, and perhaps a list holding more (in the case of APPLY), while the callee must ensure that there are neither too many nor too few of them. Optional arguments must be filled in when necessary. Likewise, the callee may yield any number of values, which the caller must be prepared to accept. All this costs cycles. The simplest (and most common) case is when the callee expects a fixed, small number of arguments, the caller supplies the proper number, and the callee yields one value. Typically, the caller transmits the argument count, encoded as an extra argument, which is then immediately decoded by the callee. Later, the callee returns its value, with a value count of 1 in a separate register; the callee is typically free to ignore the value count.

Because the argument count flag is set only to be immediately decoded, we have an opportunity to optimize out those steps by propagating that information statically. Although they are in different pieces of code, the following trick works fine: Give the callee different entry points, depending on what the caller is supplying as argument, and expecting as return values. The organization of such entry points is very naturally done on a per-type, rather a per-function basis. That is, we would have a type for "function taking exactly two arguments", and the type would support the operation "call with two arguments", but handle "call with one argument" by signaling an error. Such mechanisms are described below.

A final source of genericity in any Common Lisp implementation is argument checking. For example, the CLOSE function may not be applied to anything other than a stream. Therefore, part of the dispatch CLOSE performs will probably distinguish streams from non-streams; the latter arguments in effect cause execution of a "non-stream" method which signals an error. (Common Lisp does not mandate the signaling of an error in many of these cases, but it is a desirable behavior.)

Hard numbers to support these assertions can be found in [Steenkiste87] and [Steenkiste86], which provide profiles for a RISC machine Lisp implementation. These papers also have a good overview of common tag encoding and dispatch techniques.

The important fact is that many important Common Lisp primitives need to perform at least some case analysis on their operands. Our contention is that anywhere "some" case analysis is performed is a logical place to introduce general case analysis; that is, extensibility.

## 3. Fast Dispatch on Stock Hardware

Initially, we consider the case of "classical" dispatch, where a single operand is combined with a generic function, and the code executed depends only on the

operand's type and the message. We formalize the process using the following equation:

method = dispatch(tag, key)

The *tag* is the object's contribution to the dispatch, while the *key* depends on the generic function. In Smalltalk [Goldberg], the tag is the operand's class object, and the key is a symbol, while the dispatch function performs a traversal of the class object's hierarchy seeking a method. In Lisp Machine Flavors [Moon], the tag is (essentially) the base address of a hash table, and the key is an arbitrary Lisp object whose address bits are used to index into the hash table.

Our aim is to make the computation of the tag and the key, and their combination by the dispatch function, as fast as possible on stock hardware. To this end, we immediately specialize to dispatch functions which are lookups in a small linear table of some sort. (If hashing is involved, we charge the cycles of hash code computation to the extraction of the key and/or the tag.) The code for accessing a small table consists of the computation of the table's base address, the addition of the index (possibly scaled), and an indirection:

```
# Find table base and index, somehow:
mov table_base, %base
mov table_index, %index
# And to the lookup:
ld [%base+%index], %result
```

There are many small variants of this code; one to note in particular is the multi-way branch, in which the table holds offsets into the code stream. Note that the heart of this code is an add followed by an indirection. On most architectures, the time for this is very close to the time required for one step of case analysis, a compare followed by a branch:

```
cmp %obj_tag, desired_tag
bne wrong_tag
```

Branches have much the same performance impact on caches and pipelines that loads do, so these two codes are very similar in speed. (Note: For certain very fast operations like addition and CAR, a smaller instruction sequence may be possible. We will return to this point.)

The major constraint imposed by our table lookup code is that the table be small, no more than a few thousand bytes. The index must not be much longer than 10 bits, or the tables will begin to crowd the available (32-bit) address space. However, there is one interesting degree of freedom left: We are free to make either the key or the tag take the role of index. That is, either keys or tags must be short, but we have our choice. This is the first major choice in the design space we are exploring: short tags vs. short keys.

Most classical methods for implementing Lisp make tags short, so that their bits can be packed into the same word as the the object's data pointer. Such tags are sometimes called *immediate*, because they are always present in the CPU register file. Immediate tags can be lengthened, but only if the object reference format itself is lengthened. On stock hardware, the next reasonable size after 32 bits is twice that size: 64 bits. Many machines (even SPARC) have special instructions for manipulating such quantities, such as register pair moves, or double word loads and stores.

Some Lisp implementations put tag information into page-tables, or as an initial word accessed through the data pointer. Such non-immediate tags are called *indirect*. Indirect tags can easily be extended to 32 bits. The choice of immediate vs. indirect tag is a second choice in our design space. It helps determine the tag-extraction time.

There is a well-known and highly efficient object-oriented language which uses short keys and long tags. It is C++ [Stroustrup]. In that language, dynamically dispatched ("virtual") functions are implemented by loading an indirect long tag from the object in question; to this table base is added a key which is assigned (by the compiler) to the desired virtual function. The address of the method to invoke is found in the table. This dispatch mechanism is reasonably fast. The major disadvantage lies in the inflexible allocation of keys to generic functions. For example, it is difficult to incrementally assign new generic functions fresh keys. We will address those problems below in a way which applies equally to C++ and CLOS.

So far, we have shown two degrees of freedom in our design space: short tags vs. short keys, and immediate vs. indirect tags. The encoding of keys can also be classified as immediate or indirect. If keys are associated with pieces of code (generic function entry points), immediate keys are constants loaded directly from the code stream, while indirect keys are loaded from a static variable associated with the generic function.

There are more delicate tradeoffs to make in the case of tags than keys, because data structures must be both compact and efficiently decodable. A given data type will probably be replicated many times, and so the compactness of its representation will affect memory usage more significantly than the compactness of a piece of code, which may not be replicated many times. Often, the object code in a system is small in size relative to system data structures.

In a classical message-passing system, there are no generic functions, and the key is extracted instead from a message object, typically a symbol. In this case, key extraction is performed on a data structure, rather than from a code stream, and so tradeoffs between compactness and decodability must be made, as with tag extraction.

The next decision in designing our dispatch concerns the number of indirections which must be performed to obtain the table base, once the key and tag are

in hand. No indirections is most desirable and least flexible. In the case of a long key, an indirect table is obtained by loading from a static variable local to the function; the key is the immediate constant which addresses that variable. Otherwise, the key immediately addresses the table. Since the SPARC has no 32-bit immediate operand format, an appropriate alternative is a PC-relative address: The 32-bit key is built from an offset added to the program count, and the table (or its indirect address) is found in the instruction stream.

Once the key and tag have yielded an index and table, and the latter have been added, a final decision must be made: What is to be done with the data found there? Again, it is the choice of a flexible, indirect data structure vs. a fast immediate one. We can either load the 32-bit address found in the table, and jump to that address, or we can jump into the table itself. This is the choice between *fat* and *thin* dispatch tables. Thin tables have a regular, easy-to-manage structure, and pack in the most entries. Fat tables allow for the highest performance, because they can contain very short methods which execute with no further transfers of control. Note that the C++ dispatch mechanism described above uses thin tables, and so could be sped up by switching to fat tables.

Interestingly, on the SPARC, fat tables and thin tables have about the same performance, in the case where the table specifies an immediate transfer of control to another location. That is, a fat table contains a CALL instruction (followed by another "delayed" instruction) whose address is just the contents of the corresponding thin table slot. Jumping through the thin table requires a load and a call; the fat table requires two calls, and both instruction sequences require 4 units of time. This is probably not a peculiarity of SPARC, but rather a basic similarity between calls and indirections. (The SPARC *is* peculiar in that the delayed instruction in the fat table code has the potential to perform method-specific work.)

### 3.1. The Generic Algorithm

In summary, our dispatch algorithm has these steps:

- **Extract tag.** For indirect tags, this is a load. For short immediate tags, a field mask or extraction is needed. For long immediate tags, the tag is already in a register; the cost was incurred as an extra cycle when the reference was loaded.
- **Get table base.** For immediate tables, the address is already in a register or the instruction stream (perhaps it needs displacement). For indirect tables, a load must be performed.
- **Get table entry.** This is an addition. The offset may need to be scaled, depending on where it came from; this takes an extra cycle or two. (If the offset comes from a short immediate tag, it can be pre-shifted simply by defining the tag field as omitting the lower two bits.)
- **Get jump address.** For fat tables, this is a no-op. For thin tables, a load is required.
- **Transfer control.** This is a branch.

The degrees of freedom are (1) short tags vs. short keys, (2) table index construction (e.g., tag extraction), (3) indirect vs. immediate table base, and (4) fat vs. thin tables.

## 4. Looking at the Tradeoffs

Let us look at what these choices mean to execution time, program size, and system extensibility.

First of all, note that full extensibility is achieved only with long keys. Suppose we have a short tag of 8 bits, and the Common Lisp kernel uses 1/2 of those codes; we are left with 128 definable classes before that narrow space is exhausted. At that point, some sort of reallocation of tag codes could be performed, but it would require a traversal of the entire Lisp world (equivalent to a full GC), or some other re-tagging protocol devised. The 128-class limit would be reached quickly in "real" systems, too. The Symbolics Lisp Machine predefines nearly 3000 flavors, of which about 60% actually need tags because they have been instantiated.

On the other hand, the corresponding problem with short keys is much less severe. There are quite possibly even more generic functions than there are classes, so the key code space is no less crowded, but it is much easier to reallocate key codes than tag codes. This is because keys are tied to generic functions and not data objects, A database could be maintained by the compiler of all points where the short keys occur (as immediate constants to addition instructions); since code objects are opaque in Common Lisp, the management of the keys would be undetectable to the Lisp system.

The allocation of keys in this setting is quite similar to the problem of storage allocation in a demand-paged virtual memory system. The goal is to ensure that dynamically important generic functions always have usable keys. Some distinguished "null" key would cause all object types to invoke a method which asked the allocator for a fresh key for the current generic function. Some key codes, like those for ELT or FUNCALL, would never suffer replacement, but like tag codes in a classical Lisp system, would remain forever hardwired.

The database of key usage points can be simplified or eliminated, at a performance cost. To do this, load keys indirectly from storage locations; each generic function would store its key in one place, and all copies of its code would access the key there. This would greatly ease administration key values, but the extra indirection

would also slow the dispatch mechanism.

Note that the choice between long tags and short tags has implications for system robustness: If a damaged object reference is created, and if we are using long tags, its tag may well yield a totally inappropriate dispatch table, and using it will crash the system mightily. There are perhaps fewer things that can go wrong with short tags. On the other hand, even a short tag, if it is random enough, can cause havoc by accessing a fat table at an unexpected instruction. If the hardware performs alignment checks on memory accesses, then there is a tradeoff between robustness and speed, because a dispatch mechanism which performs more indirections, although slowed by them, is more likely to cause a bus error when handed bad data. This is a reason to prefer indirect, thin tables.

### 4.1. Example: The Fastest

With respect to raw dispatch speed, we have seen that the best possible dispatch protocol looks something like this:

```
# Extract long tag:
#mov %obj_tag, %tag
# Get immediate table base:
#mov %tag, %base
# Get fat table entry & jump address, transfer control:
jmpl %base, key, %o7
nop
# (Previous 2 lines do a displaced indirect call.)
```

In other words, it takes two cycles to get to method-specific code using this technique. (Note: The two "mov" instructions are commented out in the expectation that compile-time analysis can allocate both operands of each move to the same register, and hence render the move unnecessary.) This is extremely fast, no slower than an ordinary function call. Note the use of short keys; on the SPARC their size is 13 bits. The trade-off is a heavier use of the CPU register file: Most quantities occupy register pairs. Suppose that six of the eight SPARC argument registers are devoted to argument transmission (%o7 and %o6 are return address and stack pointer, usually). This means that in the 64-bit world, functions taking more than 3 arguments spill them out of the register file, whereas in the normal case an ample 6-argument limit is maintained. ([Steenkiste86] reports that the average Lisp procedure in a set of benchmarks had an argument count of about two.)

There is a subtle inflexibility in this scheme: Because the table is immediate, and encoded in the object tag, all objects "know" where their type's dispatch table is located. This means that, after the first object of class K is constructed, class K's dispatch table must never be relocated. This would, for example, prevent a clever system from saving space by deallocating dispatch tables of inactive classes, and patching them all with a special "wakeup" dispatch table.

### 4.2. Example: The Classic

Here is a more classical dispatch mechanism, which reverses nearly all of the decisions of the previous example (tags are still immediate):

```
# Extract short immediate tag:
and %obj, 63, %tag
# Get indirect table base (PC-relative):
ld [table], %base
# Get table entry, get jump address:
ld [%base + %tag], %meth
# Transfer control:
jmpl %meth, 0, %o7
nop
# Runtime system stores table pointer for generics:
table:
word 0
```

This sequence takes 7 cycles before method-specific code is reached. One or two cycles could be shaved off by using an immediate table in the code itself. Since small tags are used, it is nearly impossible extend the type system. However, the set of tags is reasonably large--64 tags--and could be made even larger. The fact that the 6 lower bits of address must be "free" does not preclude the creation of small objects. For example, fixnums (i.e., small integers[1]) or cons cells could be given *all* tags with a particular pattern in the lower two bits. Likewise, an eight-word type could be allocated all tags with a particular patten in the lower 5 bits, and the system would take care to create such objects properly aligned. The dispatch table technique has enough flexibility to assign multiple tags to a single type.

It is a well-known trick that tags in lower-order bits on a pointer can be removed by displaced addressing in the course of the loading through it. For example, if a the list tags are all "01" in the low-order bits, then CAR and CDR read through the pointer with displacements of -1 and 3. Moreover, if the memory bus signals errors on misaligned addresses (as is the case with SPARC), then the bus logic will serve to decode tags! Note the effect of these observations on the dispatch algorithm above: If the tag is misaligned relative to the table base it is added to, then the attempted load of the method address from the table will get a bus error. The requirement is that the sum of the table address and the tag value must be zero in the low two bits. This implies that the 64 tag codes are divided into four disjoint sets, and that a generic function may handle types within only one of those sets. (In

[1] In Lisp parlance, a *fixnum* is an integer small enough to be represented in a single machine word. It may or may not include tag bits. When fixnum arithmetic overflows, Lisp produces *bignums*, which are infinite precision integers stored on the heap. This scheme is shared by other languages, notably Smalltalk, and efficiently supports arithmetic on an unlimited range of integer values.

practice, one set would be given over to fixnums, and one to cons cells.)

### 4.3. Example: Flavors

The Lisp Machine Flavors implementation [Moon] makes the following choices: short keys, indirect tag, indirect table bases, thin tables. The short key is hashed at runtime from the generic function's name by masking away high-order pointer bits. Each table has its own mask (and therefore its own power-of-two size); this method tends to compress flavor tables. (Tables range in size from 16 entries to 1024; if they were a uniform 1024 they would occupy five million words instead of the present 0.8 million.)

Here is a SPARC rendering of the flavors dispatch mechanism, minus some machinery having to do with instance variables and variable-sized hash tables:

```
# Extract long indirect tag:
ld  [%obj], %tag
# Get indirect table base (PC-relative):
ld  [%tag + 8], %base
# Get table entry, get jump address:
and  %message, 4092, %key
ld  [%base + %key], %meth
# Transfer control:
jmpl  %meth, 0, %o7
nop
```

This sequence takes 9 cycles before method-specific code is reached. Note that we select the key mask so that the key value is properly aligned for accessing a table of words. The C++ virtual function mechanism is quite similar, except that the table is not indirect; therefore, the above sequence saves a two-cycle instruction.

## 5. Hard Problems

We have presented a plausible set of dispatch mechanisms, the fastest of which is as fast as function calling. Since dispatching on the type of a single object, in order to perform a generic operation, is an extremely common operation, it therefore behooves us to make this operation fast, as few machine instructions if possible. Using linear tables and dynamically managed table indices meets this goal of speed, but such tense techniques constrains the system implementor in important ways. Indeed, there are operations both much faster and much slower than function calling which affect the overall performance of a system: Arithmetic and slot access (e.g., addition and CAR) should be faster than a function call, and the layout of storage affects both fast operations like the loading of registers and slower ones like automatic storage management.

In this section, we examine some specific implementation issues which are affected by our choice of dispatch mechanism. These are the exploitation of hardware supported tag checks, the changing of representations at runtime, the support of multiple dispatch, and the customization of short tags.

### 5.1. Interaction with Machine-Specific Tag Decoding

Can a dispatch mechanism actually be used for the very fastest of the crucial operations, such as small integer addition or slot access (CAR, CDR)? The answer is usually "no". These operations are performed very quickly without type checking, in one or two machine instructions. Any of our dispatch mechanisms would add an unacceptable overhead, at least doubling the execution time.

At this point, hardware designers frequently offer help. As noted above, there may be architectural support (intentional or otherwise) for particular encodings of in-line short tags. Many memory busses check the low two or three address bits; RISC architectures may support tagged integer arithmetic (SPARC does), in which checking for non-fixnum operands is done in the same cycle as the actual work. There is no way a software dispatch mechanism can compete with this kind of speed. Here is a comparison of the code sequences for (SETQ Y (+ X 1)). The hardware support in this case comes from the SPARC's TADD (tagged add) instruction, which sets the overflow flag if it sees an unexpected pattern in the lower two bits of either of its operands. (We could make a similar comparison for a slot access like (SETQ Y (CAR X)); in that case the hardware support would be the memory bus alignment checking logic.) First, here is the code which uses hardware support:

```
# Perform addition (fixnum 1 is binary 00000100):
tadd  %x, 4, %y
# Check for tag exception:
bvc  done
nop
# Got an exception.
------ Do generic processing of non-fixnums...
mov  %o0, %y
done:
```

For the normal fixnum case, this code takes three cycles to complete. (There is a version which takes one cycle, but uses a more expensive trap on non-fixnum arguments.) Here is how the same operation might use a dispatch table:

```
# Assume that %x and %x_tag are properly placed:
#mov  %x, %obj
#mov %x_tag, %obj_tag
# Get fat table entry & jump address, transfer control:
jmpl  %obj_tag, plus_fixnum_key, %o7
# Set up the second operand:
mov  1, %arg2
------
# Inside dispatch table, we merely add and return:
jmp  %o7+8
```

```
# We use the delayed instruction here:
add %obj, %arg2, %obj
------
# Perform the assignment:
mov %obj, %y
mov %obj_tag, %y_tag
```

This code takes 6 cycles to complete, which is only half of the performance. Note that we assumed X was already properly placed for the method invocation. This is not an unreasonable supposition, especially if we allow fundamental generic operations like plus to support multiple entry points, one for each of several expected argument/register bindings. Note also that the hardware support exacts a penalty of two bits in the fixnum format, while the 64-bit solution allows for a full 32-bit format.

Well, if we can't beat them, we should join them. We can integrate the 32-bit tag scheme with the native tag format simply by using both schemes, as follows. The 32-bit tag completely encodes the type of the object, but the low-order two bits of the non-tag word redundantly specify a type of fixnum or cons cell, as the case may be. When the fast hardware assisted operation fails (or signals an error), the dispatch mechanism can be fallen back on. Under this scenario, even CAR and CDR can be extended to new types, if only the memory bus errors can be handled smoothly enough. Even if that is not the case, objects could be defined which look like cons cells in their first two words, but which have arbitrarily strange tag words.

If we are forced to give up two bits of fixnum precision for hardware-defined tag checking, we can recover the precision by using bits in the 32-bit tag. That is, allocate four copies of the dispatch table for fixnums, and let each copy serve to stand for one value of the missing set of bits. (Perhaps those bits can be introduced into the addresses of those tables, by clever allocation.) Indeed, there is no need to stop at recovering the original 32 bits. For example, unsigned 32-bit numbers can be represented in a 33-bit signed format; this would smooth the interface with languages like C, which support such unsigned types.

The CAR and addition operations may be too fast to be implemented with software dispatch, but function calling is not in this category. A function call and return takes at least four cycles on the SPARC, and negotiating optional arguments and return values takes even longer. These tasks are ideal for handling with our dispatch mechanisms. As noted above, argument checking should be implemented by multiple generic operations, one per calling pattern. Such a technique should actually speed up Common Lisp function calling.

### 5.2. Changing Representations and Types

Sometimes we need to change the representation of an object without changing its identity. This occurs with CLOS through the CHANGE-CLASS primitive, which performs an in-place modification of an object's type. The operation of CHANGE-CLASS is incremental, in that it attempts to preserve object state across the change. It can remove or add instance variables, but instance variables common to the old and new types are left unchanged. Also, the object can request notification of class changes, so that invariants can be maintained.

In Smalltalk, the BECOME message performs a representation change. This change is non-incremental; the object's new identity is taken from another object of arbitrary class. (BECOME is in fact symmetric; the two objects exchange representations.) Both CHANGE-CLASS and BECOME are tricky to understand, use, and implement, but they are occasionally appropriate. CLOS uses CHANGE-CLASS to support a useful notion of incremental class redefinition.

In general, when a representation change occurs, the actual bits of the object's storage will be rearranged, which implies that some of the method code used by the object must be changed to respect the new storage layout. (This change of method code amounts to a change type, whether or not the object's type seems to change at the language level.) In terms of our dispatch mechanisms, a change of method code must be effected by modification of that object's dispatch table.

This change of dispatch table can be brought about by one of two means: Either modify the object's tag bits, or modify the meaning of that tag. If we are using indirect tag bits (that is, a tag is stored in the object itself) it is easy to update the tag so that it refers to the new type.

However, if we are using immediate tag bits (that is, tags bits within the object reference) then any tag bit changes would have to be performed on every reference to the changed object. Finding all such references is a task comparable in expense to garbage collection. In this case, it is easier to leave the tag bits alone, and modify their meaning. If we are using long immediate tags (64-bit references), this means that the object's dispatch table must be modified to invoke the appropriate code for the changed object. This produces a back-compatibility problem, since that table is probably shared with other objects of the original type. This can be solved by marking the changed object so as to distinguish it from its unchanged brethren. The mark should also specify the object's new type. (This mark amounts to an auxiliary indirect tag.) Finally, the table must be filled with code which first checks for that mark, and upon seeing one, then performs a second dispatch based on the object's new type. (That is, we have forced a dispatch on an indirect tag.) Note that all objects of the old type incur a performance penalty for the mark check. We can say that the dispatch table held in common by the old type and the changed object is a "tainted" table. Such tables should not be used in the creation of new objects, and

they should be dissociated from existing objects when possible. (This is possible within the framework of a copying garbage collector.)

In the case of class redefinition, the tainted table is the dispatch table for the superseded class definition. In this case, it is further modified so that any dispatch through it causes an automatic CHANGE-CLASS of the recipient object to the new representation, when necessary. If it is eventually possible to prove that no objects of the old representation remain, the tainted table can be changed into a copy of the dispatch table of the new class definition. If this is an expected outcome, the table can be shared by both the old and new class definitions.

If CHANGE-CLASS is used in less constrained ways, the dispatch tables of participating classes might tend to stay tainted over time. The result of this would be that those classes would effectively use an indirect tag, and consequently show a slower dispatch speed.

There is a second problem which occurs when representation changes are supported: The changed object may need to increase in size. This usually means it must be reallocated to a new place in the heap, and the old location somehow marked to *forward* references to the new place. Some architectures, such as the Lisp Machine, feature specially-marked *forwarding pointers* to support this relocation. The location of the old object is filled with forwarding pointers, and then accesses are redirected to the new location by microcode. For stock hardware, the problem is harder to solve.

First note that if the first problem (of changing tags) is solved, the second one (of changing size) can be dealt with. For every type T which has an instance requiring relocation, build a new type INDIR(T), which is a "boxed" version of T, containing just one instance variable, a pointer to T. All operations on an INDIR(T) object are forwarded to the unboxed T object. Such a boxed value contains just one object reference, and so is certainly small enough to fit in the old heap location. When an object must be relocated, its type is changed to INDIR(T), where T is nominally the desired new type. (As noted above, a copying garbage collector could perform unboxing if desirable.)

### 5.3. Multiple Dispatch

Some object-oriented systems, like CLOS, allow methods to be selected according to the type of more than one operand. This is in contrast with the classical object-oriented framework, where there is always a syntactically distinguished argument, the message recipient, whose type determines the method invoked. When a method specifies the type of more than one argument, it is called a *multimethod*, and we say that the generic operation it implements performs *multiple dispatch*. In terms of our design space, multiple dispatch requires the combination of several tags and a key to produce the method to run. That is,

method = dispatch(tag1, tag2, ..., key)

This cannot be directly implemented in our scheme. Instead, a dispatch on multiple tags may be reduced to a chain of dispatches, on one tag at a time, like this:

key2 = dispatch(tag1, key)
key3 = dispatch(tag2, key2)
...
method = dispatch(tag*N*, key*N*)

Here, the intermediate stages of dispatch are represented by new keys, which are arbitrary values created by the multimethod dispatch software. (An example use of this technique may be found in [Ingalls].)

Perhaps the most well-known application of multiple dispatch is generic arithmetic. Consider the addition primitive. In Common Lisp, addition operates on a wide variety of numeric types. For example, addition must accept either fixnums or bignums. If an addition is performed on two integers, and both operands are fixnums, we can use a fast machine instruction to perform the addition (with an overflow check thrown in). Otherwise, one of the integers is a bignum, and multiple-precision arithmetic must be performed. This can be expressed in terms of multimethods by saying that there is a simple method for (FIXNUM FIXNUM) arguments, and a more general and expensive method for (INTEGER INTEGER) arguments. (Note that INTEGER is a union type for FIXNUM and BIGNUM.) A multiple dispatch mechanism is required decide which method to invoke for any two integer arguments. (In general, other numeric types occur, but this does not add new complications to the multiple dispatch problem.)

### 5.4. Modestly Extensible Metaclasses

Many Lisp systems today use small tags, typically less than eight bits, which puts strict limitations on the encoding of new data types via these tags. There is an interesting possibility for handling spare tag codes in a small-tag Common Lisp implementation. The key is is making available a mechanism for defining new types under the meta-class BUILTIN-CLASS; the handful of available tag codes would still serve researchers who only want one more type, such as [Sabot] and [Hillis].

## 6. Summary

We have described a family of dispatch algorithms, some of which approach the speed of plain subroutine linkage. All depend on small tables to combine a "tag" and a "key". It appears that 32-bit tags have definite advantages of flexibility and even speed over classical small tags. Methods are outlined for managing small keys which are inherently more tractable than corresponding problems with managing small type tags. It is our belief that further thought on these issues will suggest interesting almost-standard 64-bit hardware for

symbolic and object-oriented programming.

## 7. References


[ANSI] Daniel Bobrow, David Moon, et al., "Common Lisp Object System Specification", ANSI X3J13 Document 87-002, American National Standards Institute, Washington, DC, 1988.

[Brooks] Rodney A. Brooks et al, "Design of An Optimizing, Dynamically Retargetable Compiler", *Proc. 1986 ACM Conf. on Lisp and Functional Programming*, pp. 67-85. ACM, Cambridge, 1986.

[Goldberg] Adele Goldberg and David Robson, *Smalltalk-80: The Language and Its implementation*, Addison-Wesley, Reading, MA, 714 pp., 1983.

[Hillis] Guy L. Steele, Jr., and W. Daniel Hillis, "Connection Machine LISP: Fine-Grained Parallel Symbolic Processing", *Proc. 1986 ACM Conf. on Lisp and Functional Programming*, pp. 279-297. ACM, Cambridge, 1986.

[Ingalls] Dan Ingalls, "A Simple Technique for Handling Multiple Polymorphism", Proc. OOPSLA, *SIGPLAN Notices, 21(11)*, pp. 347-349, 1986.

[Moon] David Moon, "Object-Oriented Programming with Flavors," Proc. OOPSLA, *SIGPLAN Notices, 21(11)*, pp. 1-8, 1986.

[Sabot] Gary W. Sabot, *An Architecture-Independent Model for Parallel Programming*, (Ph.D. Thesis), Harvard Univ. Center for Research in Computing Technology TR-06-88, February, 1988.

[Steele] Guy Steele, *Common Lisp: The Language*, Digital Equipment Corp., 465 pp., 1984.

[Steenkiste86] Peter Steenkiste and John Hennessy, "Lisp on a Reduced-Instruction-Set-Processor", *Proc. 1986 ACM Conf. on Lisp and Functional Programming*, pp. 192-201. ACM, Cambridge, 1986.

[Steenkiste87] Peter Steenkiste and John Hennessy, "Tags and Type Checking in LISP: Hardware and Software Approaches", *Proc. 2nd International Conf. on Architectural Support for Programming Languages and Operating Systems* (Palo Alto, CA, Oct. 5-8, 1987), pp. 50-59.

[Stroustrup] Bjarne Stroustrup, "Data Abstraction in C", *AT&T Bell Laboratories Technical Journal*, vol. 68, no. 8, October 1984.

[Sun] Sun Microsystems, *The SPARC Architecture Manual* (part no. 800-1399-07), Mountain View, CA, August 8, 1987.