Message Dispatch on Pipelined Processors

Karel Driesen
Urs Hölzle
Jan Vitek[1]

Abstract. Object-oriented systems must implement message dispatch efficiently in order not to penalize the object-oriented programming style. We characterize the performance of most previously published dispatch techniques for both statically- and dynamically-typed languages with both single and multiple inheritance. Hardware organization (in particular, branch latency and superscalar instruction issue) significantly impacts dispatch performance. For example, inline caching may outperform C++-style "vtables" on deeply pipelined processors even though it executes *more* instructions per dispatch.
We also show that adding support for dynamic typing or multiple inheritance does not significantly impact dispatch speed for most techniques, especially on superscalar machines. Instruction space overhead (calling sequences) can exceed the space cost of data structures (dispatch tables), so that minimal table size may not imply minimal run-time space usage.

Keywords: message dispatch, implementation, performance, computer architecture

1 Introduction

Message dispatch is a central feature of object-oriented languages. Given a receiver object and a selector (i.e., operation name), message dispatch finds the method implementing the operation for the particular receiver object. Since message dispatch is performed at run-time and is a very frequent operation in object-oriented programs, it must be fast. Therefore, the efficient implementation of message dispatch has been the subject of much previous work. Unfortunately, this research has often presented particular dispatch implementations in isolation, without comparing them to other methods. This paper presents several dispatch techniques in a common framework and compares their cost on modern computer architectures. The study includes most previously published dispatch techniques for both statically- and dynamically-typed languages with both single and multiple inheritance.

Any comparative study of dispatch mechanisms must be a compromise between breadth and depth since it is impossible to explore the entire design space in a single paper. While the present study considers several aspects of dispatch mechanisms (such as speed and space efficiency), the main focus is on run-time dispatch performance. But even when considering only run-time dispatch speed, a myriad of issues must be

[1] Author's addresses: Karel Driesen and Urs Hölzle, Department of Computer Science, University of California, Santa Barbara, CA93106, USA; {karel,urs}@cs.ucsb.edu. Jan Vitek, Object Systems Group, Centre Universitaire d'Informatique, Université de Genève, 24 rue Général Dufour, CH-1211 Genève 4, Switzerland; jvitek@cui.unige.ch.

Accepted at ECOOP'95, to appear in Lecture Notes in Computer Science

addressed. The remainder of this introduction briefly discusses and justifies the issues we address as well as those we don’t.

1.1 Specific measurements vs. analytical models

Previous studies have evaluated the run-time performance of specific dispatch implementations relative to specific systems, languages, and applications; some have not evaluated run-time performance at all. While specific empirical measurements are useful and desirable, they are also limited in scope. Different languages or applications may have different dispatch characteristics, and an implementor who is trying to choose between dispatch techniques may not yet know how the new system relates to the system used for the specific measurements. As discussed below, different processor implementations also change the relative speed of dispatch mechanisms. Any performance comparison using specific measurements will therefore be relative to the particular processors, languages, applications, and run-time systems used.

Therefore, instead of giving concrete measurements, we chose to characterize the performance of each dispatch mechanisms as a function of several configuration parameters that are dependent on the hardware and software environment of the system using the dispatch mechanism. To compare dispatch performance in a new system, an implementor therefore merely needs to measure (or approximate) the values of these performance parameters in that system. By keeping our performance analysis abstract, we hope that this study will be helpful to implementors of a wide range of systems, languages, and applications on a range of hardware platforms.

To help illustrate specific points of trends in the analysis, we also present absolute performance numbers which were obtained by using typical values (taken from previous studies) for the parameters.

1.2 Processor architecture

Dispatch cost is intimately coupled with processor implementation. The same dispatch sequence may have different cost on different processor implementations, even if all of them implement the same architecture (e.g., the SPARC instruction set). In particular, processor pipelining and superscalar execution make it impossible to use the number of instructions in a code sequence as an accurate performance indicator. This paper characterizes the run-time performance of dispatch mechanisms on modern pipelined processors by determining the performance impact of branch latency and superscalar instruction issue. In addition to providing specific numbers for three example architectures, our analysis allows dispatch performance to be computed for a wide range of possible (future) processors. With the rapid change in processor design, it is desirable to characterize performance in a way that makes the dependence on certain processor characteristics explicit, so that performance on a new processor can be estimated accurately as long as the processor’s characteristics are known.

1.3 Influence of dynamic typing

In dynamically-typed languages, a program may try to invoke an operation on some object for which the operation is undefined (“message not understood” error).

Therefore, each message dispatch usually needs to include some form of run-time check to guarantee that such errors are properly caught and reported to the user. Most techniques that support static typing can be extended to handle dynamic typing as well. Our study shows the additional dispatch cost of dynamic typing for all dispatch mechanisms that can support it.

1.4 Single versus multiple inheritance

A system using multiple inheritance (MI) introduces an additional difficulty if compiled code uses hard-coded offsets when addressing instance variables. For example, assume that class *C* inherits directly from classes *A* and *B* (Figure 1). In order to reuse compiled code of class *A*, instances of *C* would have to start with the instance variables of *A* (i.e., *A*'s memory layout must be a prefix of *C*'s layout). But the compiled code in class *B* requires a conflicting memory layout (*B*'s instance variables must come first), and so it seems that compiled code cannot be reused if it directly addresses instance variables of an object.

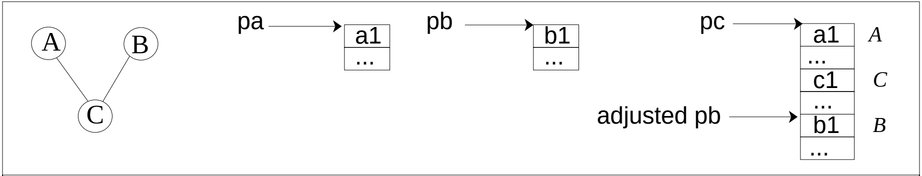


Figure 1. The memory layout of objects of class *A*, *B*, and *C*. Note that the offset of instance variable *b1* is different in instances of *B* versus instances of *C*.

Hard-coded offsets can be retained if the receiver object's address is adjusted just before a *B* method is executed, so that it points to the *B* subobject within *C* [Kro85, ES90].[1] The adjustment can be different for every class that has *B* as a (co-)parent. (If dynamic typing is combined with multiple inheritance it is necessary to keep track of both the unadjusted address and the adjustment.) Strictly speaking, this extra code is not part of method lookup, but if multiple inheritance is allowed, *every* method invocation is preceded by a receiver address adjustment, and thus we chose to include the cost of this adjustment in our study.

1.5 Limitations

This study is limited to single (i.e., receiver-based) dispatch. Since multi-method dispatch techniques (e.g., [KR90, AGS94]) are similar and include single dispatch as an important (very frequent) special case, we hope that the results will nevertheless be useful for implementors or designers of multiple dispatch techniques. Also, we do not consider dynamic inheritance (as used in SELF [CU+91]), i.e., inheritance hierarchies that can change their structure at run-time. Furthermore, we focus on dispatch performance and only briefly discuss other issues such as space overhead and the closed– vs. open–world assumption (see section 6). For space reasons, we consider only the main variant of each technique.

[1] Alternatively, a system could duplicate code (e.g., as in SELF [CUL89]) or access instance variables indirectly (as is done in Eiffel and Sather [MS94]).

A further simplification on the hardware side is that we do not consider every possible processor implementation feature. However, as will be explained in section 5.1, the features we consider represent a very large fraction of past and current processors. Further hardware-related limitations are discussed in section 5.5.

1.6 Overview of the paper

The remainder of this paper is organized as follows. Sections 2 to 4 briefly review the dispatch techniques evaluated. Section 5 presents the results of our performance analysis, and section 6 discusses space costs and other issues.

2 Method lookup techniques

Message lookup is a function of the message name (selector) and the receiver class. If lookup speed was unimportant, lookup could be performed by searching class-specific dispatch tables. When an object receives a message, the object's class is searched for the corresponding method, and if no method is found the lookup proceeds in the superclass(es). Since it searches dispatch tables for methods, this technique is called Dispatch Table Search (DTS). The right-hand side of Figure 2 shows the dispatch tables of the class hierarchy on the left. Each entry in a dispatch table contains the method name and its address. As in all other figures, capital letters (*A*, *B*, *C*) denote classes and lowercase letters denote methods.

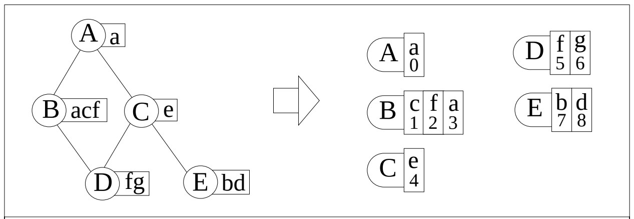


Figure 2. Class hierarchy with corresponding dispatch tables

Since the memory requirements of DTS are minimal (i.e., proportional to the number of methods in the system), DTS is often used as a backup strategy which is invoked when faster methods fail. If desired, DTS can employ hashing to speed up the table search.

All of the techniques discussed in the remainder of this paper improve upon the speed of DTS by precomputing or caching lookup results. The dispatch techniques studied here fall into two categories. *Static techniques* precompute all data structures at compile or link time and do not change those data structures at run-time. Thus, static techniques only use information that can be statically derived from the program's source text. *Dynamic techniques* may precompute some information at compile or link time, but they dynamically update data structures at run-time, i.e., during program execution. Thus, dynamic techniques can exploit run-time information as well as static information. Table 1 lists the techniques we studied.

Each description of a dispatch technique includes pseudo-code illustrating the run-time dispatch. Since our analysis separates out the influence of dynamic typing and multiple

	Acronym	Full Name	Section
static techniques	DTS	Dispatch Table Search (used for illustration only)	2
	STI	Selector Table Indexing (used for illustration only)	4.1
	VTBL	Virtual Function Tables	4.2
	SC	Selector Coloring	4.3
	RD	Row Displacement	4.4
	CT	Compact Tables	4.5
dynam. techn.	LC	Lookup Caching	3.1
	IC	Inline Caching	3.2
	PIC	Polymorphic Inline Caching	3.3

Table 1. Overview of dispatch methods

inheritance/hardwired instance variable offsets, two typographical conventions mark code used to support one of these functions. *Italic code* supports multiple inheritance, and **bold code** supports dynamic typing. Table A-2 in the Appendix contains assembly code sequences.

3 Dynamic techniques

Dynamic techniques speed up message lookup by using various forms of caching at run-time. Therefore, they depend on locality properties of object-oriented programs: caching will speed up programs if the cached information is used often before it is evicted from the cache. This section discusses two kinds of caching: global caching (one large cache per system) and inline caching (one small cache per call site).

3.1 Global lookup caches (LC)

First-generation Smalltalk implementations relied on a global cache to speed up method lookup [GR83, Kra83]. The class of a receiver, combined with the message selector, hashes into an index in a global cache. Each cache entry consists of a class, a selector and a method address. If the current class and selector match the ones found in the entry, the resident method is executed. Otherwise, a dispatch table search finds the correct method, and the new class-selector-method triple replaces the old cache entry (direct-mapped cache). Any hash function can be used; to obtain a lower bound on lookup time, we assume a simple exclusive OR of receiver class and selector.

```
entry = cache[(object->class ^ #selector) & #mask];
if (entry.class == object->class && entry.selector == #selector) {
  adjusted = object + entry.delta;
  entry.func(object, adjusted, arguments); /* cache hit */
} else { /* cache miss: use DTS to find method, and update cache entry e */; }
```

Figure 3. Global lookup cache. Multiple inheritance adds *italicized* code.

To allow hard-coded instance variable offsets in a multiple inheritance context, the receiver is adjusted by #delta.[1] Instance variable accesses in the callee use this adjusted object pointer. Even though LC is considerably faster than dispatch table search (DTS),

[1] To distinguish constants from variables, we precede constants with a hash mark (#) in both C and assembly code.

it still has to compute a hash function for each dispatch. As we shall see, this computation renders LC too slow compared to other techniques. However, LC is a popular fallback method for inline caching.

3.2 Inline caches (IC)

Often, the type of the receiver *at a given call site* rarely varies; if a message is sent to an object of type *X* at a particular call site, it is likely that the next send will also go to an object of type *X*. For example, several studies have shown that the receiver type at a given call site remains constant 95% of the time in Smalltalk code [DS84, Ung87, UP87]. This locality of type usage can be exploited by caching the looked-up method address at the call site. Because the lookup result is cached "in line" at every call site (i.e., no separate lookup cache is accessed in the case of a hit), the technique is called *inline caching* [DS84, UP87].

The previous lookup result is cached by changing the call instruction implementing the send, i.e., by modifying the compiled program on the fly. Initially, the call instruction calls the system's lookup routine. The first time this call is executed, the lookup routine finds the target method. Before branching to the target, the lookup routine changes the call instruction to point to the target method just found (Figure 4). Subsequent executions of the send directly call the target method, completely avoiding any lookup. Of course, the type of the receiver could have changed, and so the prologue of the called method must verify that the receiver's type is correct and call the lookup code if the type test fails.

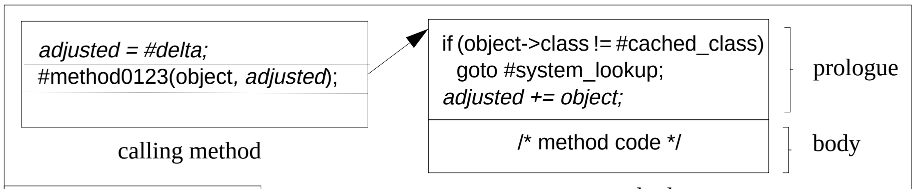


Figure 4. Inline cache

Inline caches are very efficient in the case of a cache hit: in addition to the function call, the only dispatch overhead that remains is the check of the receiver type in the prologue of the target. The above code sequence works for both static and dynamic typing; in the MI case, an inline cache miss updates both the call instruction and the add instruction adjusting the receiver address.

The dispatch cost of inline caching critically depends on the hit ratio. In the worst case (0% hit ratio) it degenerates to the cost of the technique used by the system lookup routine (often, a global lookup cache), plus the extra overhead of the instructions updating the inline cache. Fortunately, hit ratios are usually very good, on the order of 90-99% for typical Smalltalk or SELF code [Ung87, HCU91]. Therefore, many current Smalltalk implementations incorporate inline caches.

3.3 Polymorphic inline caching (PIC)

Inline caches are effective only if the receiver type (and thus the call target) remains relatively constant at a call site. Although inline caching works very well for the majority of sends, it does not speed up a polymorphic call site[1] with several equally likely receiver types because the call target switches back and forth between different methods, thus increasing the inline cache miss ratio. The performance impact of inline cache misses can become severe in highly efficient systems. For example, measurements of the SELF-90 system showed that it spent up to 25% of its time handling inline cache misses [HCU91].

Polymorphic inline caches (PICs) [HCU91] reduce the inline cache miss overhead by caching *several* lookup results for a given polymorphic call site using a dynamically-generated PIC routine. Instead of just switching the inline cache at a miss, the new receiver type is added to the cache by extending the stub routine. For example, after encountering receiver classes A and B, a send of message *m* would look as in Figure 5.

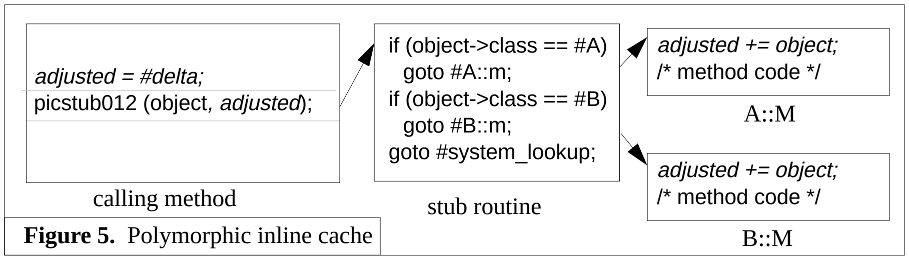


Figure 5. Polymorphic inline cache

A system using PICs treats monomorphic call sites like normal inline caching; only polymorphic call sites are handled differently. Therefore, as long as the PIC's dispatch sequence (a sequence of ifs) is faster than the system lookup routine, PICs will be faster than inline caches. However, if a send is megamorphic (invokes many different methods), it cannot be handled efficiently by PICs. Fortunately, such sends are the exception rather than the rule.

4 Static techniques

Static method lookup techniques precompute their data structures at compile time (or link time) in order to minimize the work done at dispatch time. Typically, the dispatch code retrieves the address of the target function by indexing into a table and performing an indirect jump to that address. Unlike lookup caching (LC), static methods usually don't need to compute a hash function since the table index can be computed at compile time. Also, dispatch time usually is constant[2], i.e., there are no "misses" as in inline caching.

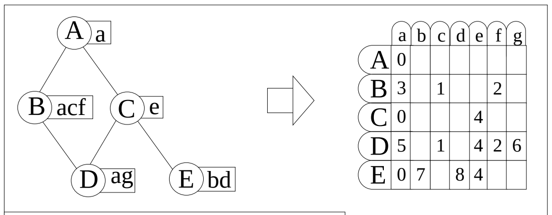
4.1 Selector Table Indexing (STI)

The simplest way of implementing the lookup function is to store it in a two-dimensional table indexed by class and selector codes. Both classes and selectors are

[1] We will use the term "polymorphic" for call sites where polymorphism is *actually* used. Consequently, we will use "monomorphic" for call sites which experience only a single receiver type during a program run, even though they might *potentially* be polymorphic.

[2] Here, "constant" means "constant in number of instructions," not "constant in real time" (due to processor cache effects).

represented by unique, consecutive class or selector codes; if a system has *c* classes and *s* selectors, classes are numbered *0..c-1* and selectors are numbered *0..s-1* (Figure 6). Unfortunately, the resulting dispatch table is very large (*O(c*s)*) and very sparse, since most messages are defined for only a few classes. For example, about 95% of the entries would be empty in a table for a Smalltalk image [Dri93b]. With multiple inheritance, every entry consists of a method code address and a delta (the adjustment to the receiver address). To avoid cluttering the graphics, we do not show the latter in any figure.



	a	b	c	d	e	f	g
A	0						
B	3		1			2	
C	0				4		
D	5		1		4	2	6
E	0	7		8	4		

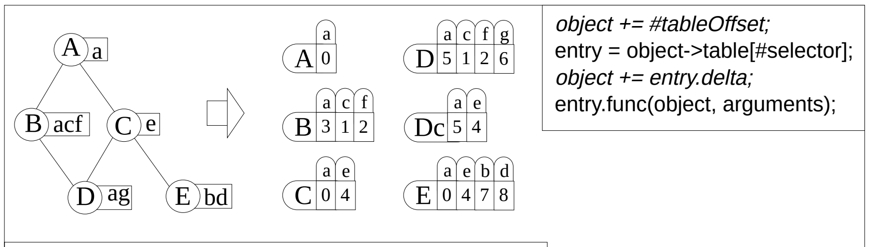
Figure 6. STI dispatch tables

STI works equally well for static and dynamic typing, and its dispatch sequence is fast. However, because of the enormous space cost, no real system uses selector table indexing. All of the static techniques discussed below try to retain the idea of STI (indexing into a table of function pointers) while reducing the space cost by omitting empty entries in the dispatch table.

4.2 Virtual function tables (VTBL)

Virtual function tables were first used in Simula [DM73] and today are the preferred C++ dispatch mechanism [ES90]. Instead of assigning selector codes globally, VTBL assigns codes only within the scope of a class. In the single-inheritance case, selectors are numbered consecutively, starting with the highest selector number used in the superclass. In other words, if a class *C* understands *m* different messages, the class's message selectors are numbered *0..m-1*. Each class receives its own dispatch table (of size *m*), and all subclasses will use the same selector numbers for methods inherited from the superclass. The dispatch process consists of loading the receiver's dispatch table, loading the function address by indexing into the table with the selector number, and jumping to that function.

With multiple inheritance, keeping the selector code correct is more difficult. For the inheritance structure on the left side of Figure 7, functions *c* and *e* will both receive a selector number of 1 since they are the second function defined in their respective class.



	a
A	0

	a	c	f
B	3	1	2

	a	e
C	0	4

	a	c	f	g
D	5	1	2	6

	a	e
Dc	5	4

	a	e	b	d
E	0	4	7	8

```
object += #tableOffset;
entry = object->table[#selector];
object += entry.delta;
entry.func(object, arguments);
```

Figure 7. VTBL dispatch tables and method call code.

D multiply inherits from both *B* and *C*, creating a conflict for the binding of selector number 1. In C++ [ES90], the conflict is resolved by using multiple virtual tables per class. An object of class *D* has two dispatch tables, *D* and *Dc* (see Figure 7).[1] Message sends will use dispatch table *D* if the receiver object is viewed as a *B* or a *D* and table *Dc* if the receiver is viewed as a *C*. As explained in section 1.4, the dispatch code will also adjust the receiver address before calling a method defined in *C*.

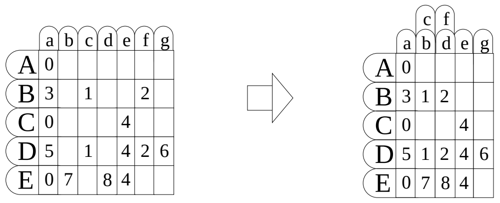
VTBL depends on static typing: without knowing the set of messages sent to an object, the system cannot reuse message numbers in unrelated classes (such as using 0 for the first method defined in a top-level class). Thus, with dynamic typing, VTBL dispatch tables would degenerate to STI tables since any arbitrary message could be sent to an object, forcing selector numbers to be globally unique.

4.3 Selector coloring (SC)

Selector coloring [D+89, AR92] is a compromise between VTBL and STI. SC is similar to STI, but instead of using the selector to index into the table, SC uses the selector's *color*. The color is a number that is unique within every class where the selector is known, and two selectors can share a color if they never co-occur in a class. SC allows more compaction than STI, where selectors never share colors, but less compaction than VTBL, where a selector need not have a single global number (i.e., where the selector *m* can have two different numbers in unrelated classes).

```
entry = object->table[#color];
entry.func(object, entry.delta, #selector, arguments)
```

```
/* method prologue */
if (S != #mySelector) error("Message Not Understood");
adjusted = delta + object;
```



	a	b	c	d	e	f	g
A	0						
B	3		1			2	
C	0				4		
D	5		1		4	2	6
E	0	7		8	4		

		c	f		
	a	b	d	e	g
A	0				
B	3	1	2		
C	0			4	
D	5	1	2	4	6
E	0	7	8	4	

Figure 8. Selector coloring tables, method call code, and method prologue (dynamic typing in **bold**)

Optimally assigning colors to selectors is equivalent to the graph coloring problem[2] which is NP-complete. However, efficient approximation algorithms can often approach or even reach the minimal number of colors (which is at least equal to the maximum number of messages understood by any particular class). The resulting global dispatch table is much smaller than in STI but still relatively sparse. For example, 43% of the entries are empty (i.e., contain "message not understood") for the Smalltalk

[1] Due to limited space, we ignore virtual base classes in this discussion. They introduce an extra overhead of a memory reference and a subtraction [ES90].

[2] The selectors are the nodes of the graph, and two nodes are connected by an arc if the two selectors co-occur in any class.

system [Dri93b]. As shown in Figure 8, coloring allows the sharing of columns of the selector table used in STI.

Compared to VTBL, SC has two potential advantages. First, since selector colors are global, only one dispatch table is needed per class, even in the context of multiple inheritance. Secondly, and for the same reason, SC is applicable to a dynamically-typed environment since any particular selector will have the same table offset (i.e., color) throughout the system and will thus invoke the correct method for any receiver. To guard against incorrect dispatches, the prologue of the target method must verify the message selector, and thus the selector must be passed as an extra argument. Otherwise, an erroneous send (which should result in a "message not understood" error) could invoke a method with a different selector that shares the same color. For example, in Figure 8, message *c* sent to a *E* object would invoke *b* without that check.

4.4 Row displacement (RD)

Row displacement [Dri93a] is another way of compressing STI's dispatch table. It slices the (two-dimensional) STI table into rows and fits the rows into a one-dimensional array so that non-empty entries overlap only with empty ones (Figure 9). Row offsets must be unique (because they are used as class identifiers), so no two rows start at the same index in the master array. The algorithm's goal is to minimize the size of the resulting master array by minimizing the number of empty entries; this problem is similar to parse table minimization for table-driven parsers [DDH84]. [Dri93a] discusses a selector numbering scheme that leaves only 33% of the entries empty for the Smalltalk image. If the table is sliced according to columns (instead of rows), the table can even be filled to 99.7% with an appropriate class numbering scheme [DH95]. Like SC, RD needs only a single table per class even with multiple inheritance, and the technique is applicable to dynamically-typed languages. As in SC, a check is needed in the method prologue, this time to ensure that the method actually is part of the dispatch table of the receiver's class. Therefore, the selector number is passed as an argument to the method.

```
entry = object->table[#selector];
entry.func(object, entry.delta, #selector, arguments);
```

```
/* method prologue */
if (S != #mySelector) error("Message Not Understood");
adjusted = delta + object;
```

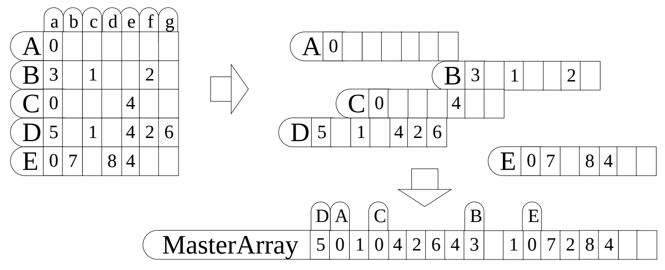


Figure 9. Row displacement tables, method call code, and method prologue

4.5 Compact Selector-Indexed Dispatch Tables (CT)

The third table compaction method [VH94], unlike the two previous methods, generates selector-specific dispatch code sequences. The technique separates selectors into two categories. *Standard selectors* have one main definition and are only overridden in the subclasses (e.g., *a* and *b* in Figure 10). *Conflict selectors* have multiple definitions in unrelated portions of the class hierarchy (e.g., *e* in Figure 10 which is defined in the unrelated classes *C* and *D*). CT uses two dispatch tables, a main table for standard selectors and a conflict table for conflict selectors.

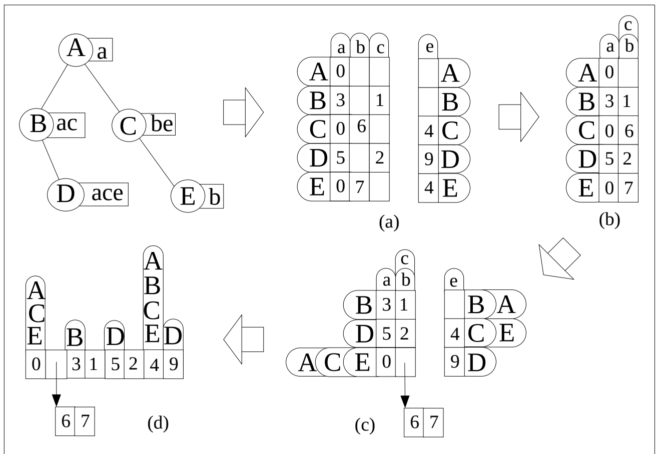
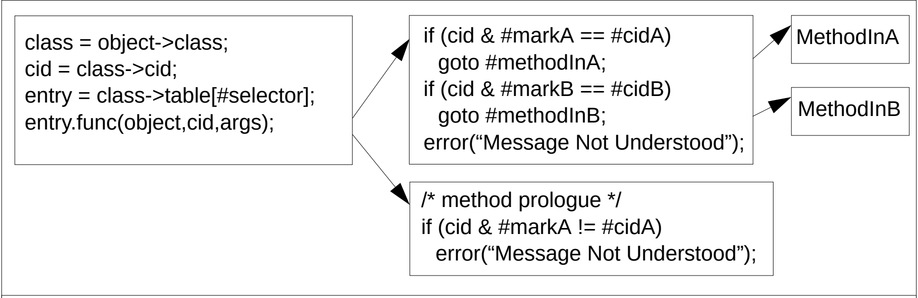


Figure 10. Construction on Compact Tables
(a) main dispatch table and conflict tables for *e*
(b) main dispatch table with *c* and *b* aliased
(c) conflict tables for (*A*, *B*) and (*C*, *E*) shared, main dispatch tables for (*A, C, E*) overloaded
(d) dispatch table fitted on a single master array, main dispatch table for (*A, C, E*) trimmed

Standard selectors can be numbered in a simple top-down traversal of the class hierarchy; two selectors can share a number as long as they are defined in different branches of the hierarchy. Such sharing is impossible for conflict selectors, and so the conflict table remains sparse (Figure 10). But the allocation of both tables can be further optimized. First, tables with identical entries (such as the conflict tables for *C* and *E*) can be shared. Second, tables meeting a certain similarity criterion—a parameter to the algorithm—can be *overloaded*; divergent entries refer to a code stub which selects the appropriate method based on the type (similar to PIC). In Figure 10 (a), the entry for selectors *c* and *b* of tables (*A*, *C*, *E*) is overloaded. The required level of similarity affects the compression rate (stricter requirements decrease the compression rate) as well as dispatch speed (stricter requirements decrease the number of overloaded entries and thus improve dispatch speed). Finally, dispatch tables are trimmed of empty entries and allocated onto one large master array as shown in Figure 10 (b).

Each class needs at least three fields: class identifier (*cid*), main dispatch table, and conflict table. Because of compression, all methods need a subtype test in the method prologue in dynamically-typed languages. For statically-typed languages, only the code stubs of overloaded entries need such a test. Subtype tests are implemented with a simple series of logical operations (a bit-wise AND and a comparison) [Vit95]. Figure 11 shows the code for a call through a CT dispatch table.



```
class = object->class;
cid = class->cid;
entry = class->table[#selector];
entry.func(object,cid,args);
```

```
if (cid & #markA == #cidA)
  goto #methodInA;
if (cid & #markB == #cidB)
  goto #methodInB;
error("Message Not Understood");
```

```
/* method prologue */
if (cid & #markA != #cidA)
  error("Message Not Understood");
```

Figure 11. CT dispatch code. The caller code calls a stub for overloaded entries (upper box). Single implementation entries only require a prologue (lower box) for dynamic typing.

This version of the algorithm (from [VH94]) only handles single inheritance, because of the lack of fast type inclusion test for multiple inheritance.[1]

5 Analysis

5.1 Parameters influencing performance

To evaluate the performance of the dispatch mechanisms, we implemented the dispatch instruction sequence of each technique on a simple RISC-like architecture.[2] Table A-2 in the Appendix lists the resulting instruction sequences. Then, we measured the cost of the dispatch sequences for three hypothetical processor implementations. P92 represents a scalar implementation as it was typical of processor designs in 1992. P95 is a superscalar implementation that can execute up to two integer instructions concurrently, representative of current state-of-the art processor designs. Finally, P97 is an estimate of a 1997 superscalar processor with four-instruction issue width and a deeper pipeline. Table 2 lists the detailed processor characteristics relevant to the study.

In essence, these processors are abstractions of current commercial processors that have been reduced to their most important performance features, namely

- *Superscalar architecture*. The processor can execute several instructions in parallel as long as they are independent. Since access paths to the cache are expensive, all but P97 can execute at most one load or store per cycle.
- *Load latency.* Because of pipelining, the result of a load started in cycle i is not available until cycle $i + L$ (i.e., the processor will stall if the result is used before that time).

[1] This problem is tackled in [VH95]. [Vit95] presents a version of the algorithm which improves dispatch speed and shortens the calling sequence. Unfortunately, we have not been able to analyze and measure this new version yet.

[2] Assuming a RISC architecture simplifies the discussion, but our results are not restricted to RISC architectures since pipelined CISC machines have similar performance characteristics. For example, the Intel Pentium processor closely resembles P95.

- *Branch penalty.* The processor predicts the outcome of a conditional branch; if the prediction is correct, the branch incurs no additional cost. However, if the prediction is incorrect, the processor will stall for *B* cycles while fetching and decoding the instructions following the branch [HP90]. We assume that indirect calls or jumps cannot be predicted and always incur the branch penalty.[1]

	P92	P95	P97
max. integer instructions/ cycle	1	2	4
max. loads or stores / cycle	1	1	2
max. control transfers (branch, call) / cycle	1	1	1
load latency (L)[a]	2	2	2
branch prediction	no	yes	yes
branch miss penalty (B)	1[b]	3	6
examples of equivalent commercial CPUs	[M92, Cy90]	[M94, Gw94]	N/A

Table 2. Processor characteristics

[a] To simplify the analysis, we assumed L > 1; to the best of our knowledge, this assumption holds for all RISC processors introduced since 1990.
[b] No penalty if the branch's delay slot can be filled. (To improve readability, the instruction sequences in Table A-2 are written without delay slots.) On P95/97, delay slots cannot hide branch latency due to multi-cycle branch penalties and superscalar instruction issue, and thus have no performance benefit.

Virtually all processors announced since 1993 exhibit all three characteristics. We also assumed out-of-order execution for the superscalar machines (P95 and P97). To determine the number of cycles per dispatch, we hand-scheduled the dispatch instruction sequences for optimal performance on each processor. In most cases, a single instruction sequence is optimal for all three processors.

Variable	Typical value	Comments
h_{LC}	98%	lookup cache hit ratio ([CPL83] lists 93% for a very small cache size)
$miss_{LC}$	250[a]	LC miss cost (find method in class dictionaries); conservative estimate based on data in [Ung87]
h_{IC}	95%	inline caching hit ratio; from [Ung87] and [HCU91]
$miss_{IC}$	80[a]+L+LC	IC miss cost; from [HCU91]
m	66%	fraction of calls from monomorphic call sites (PIC) [HCU91, CG94]
k	3.54	dynamic number of type tests per PIC stub (from SELF [Höl94])
p	10%	average branch misprediction rate (estimate from [HP90])
M	1%	fraction of calls from highly polymorphic call sites (> 10 receiver types); conservative estimate (in SELF, M < 0.1% [Höl94])
$miss_{PIC}$	150[a]+L+LC	PIC miss cost; based on $miss_{IC}$ (overhead for updating the PIC)
s	99.93%	percentage of single (non-overloaded) entries in CT [VH94]
e	2.25	number of tests per overloaded entry in CT

Table 3. Additional parameters influencing performance

[a] Cycles on P92; 20% less on P95 and 33% less on P97.

The performance of some dispatch techniques depends on additional parameters (listed in Table 3). In order to provide some concrete performance numbers in addition to the formulas, we chose typical values for these parameters (most of them based on

[1] But see section 5.5.

previously published performance studies). However, it should be emphasized that these values merely represent one particular data point. Different systems, applications, or languages may well exhibit different parameter values. Thus, the *numbers* below are specific to an example configuration, but the *formulas* in Tables 4 to 6 are general.

5.2 Overview of dispatch costs

Tables 4 to 6 show dispatch costs as a function of processor parameters (L and B) and algorithmic parameters such as miss ratios, etc. Table A-2 in the Appendix lists the exact assembly instruction sequences used, and Table A-3 contains the raw dispatch times in cycles.

	single inheritance		multiple inheritance	
	static typing	dynamic typing	static typing	dynamic typing
LC	h_{LC}*(9+max(7,2L)+(1-h_{LC})*$miss_{LC}$		h_{LC}*(9+max(10,2L))+(1-h_{LC})*$miss_{LC}$	
VTBL	2+2L	N/A	2+2L	N/A
SC	2+2L	4+2L	2+2L	5+2L
RD	3+L + max(L, 3)	7+L+max(3,L)	2+2L+max(3,L)	7+max(5,2L)
CT	s*(2+3L)+(1-s) *(3+3L+7e)	s*(8+3L)+(1-s) *(3+3L+7e)	N/A	N/A
IC	h_{IC} * (2+max(3,L)) + (1-h_{IC}) * $miss_{IC}$		h_{IC} * (3+max(4,L)) + (1-h_{IC}) * $miss_{IC}$	
PIC	m * (2+max(3,L)) + (1-m) * (2+L+2k) + M * $miss_{PIC}$		m * (3+max(4,L)) + (1-m) * (3+L+2k)+ M * $miss_{PIC}$	

Table 4. P92

	single inheritance		multiple inheritance	
	static typing	dynamic typing	static typing	dynamic typing
LC	h_{LC} * (7+2L+B) +(1-h_{LC}) * $miss_{LC}$		h_{LC} * (8+2L+B) + (1-h_{LC}) * $miss_{LC}$	
VTBL	1+2L+B	N/A	see single inher.	N/A
SC	1+2L+B	3+2L+B	see single inher.	see single inher.
RD	1+2L+B	3+2L+B	2+2L+B	5+2L+B
CT	s*(1+3L+B)+(1-s)* (2+3L+B+e(4 +pB))	s*(5+3L+B)+(1-s)* (2+3L+B+e(4 +pB))	N/A	N/A
IC	h_{IC} * (1+L) + (1-h_{IC}) * $miss_{IC}$		h_{IC} * (2+L) + (1-h_{IC}) * $miss_{IC}$	
PIC	m * (1+L)+(1-m)*(2+L+k(1 +pB)) + M * $miss_{PIC}$		m*(2+L)+(1-m)*(2+L+k(1 +pB)) + M * $miss_{PIC}$	

Table 5. P95

Figure 12 illustrates the cycle cost calculation for VTBL. Data dependencies are indicated with arrows, control dependencies with dashed arrows. Instructions handling multiple inheritance are enclosed by dashed circles. The figure shows the order in which instructions are issued into the processor pipeline.[1] An instruction with a dependency on a load instruction executing in cycle i cannot execute before cycle $i + L$ (where L is the load latency). For example, in P92 instruction 2 cannot execute before cycle L

[1] More precisely, the cycle in which the instruction enters the EX stage (this stage calculates the result in arithmetic operations or the effective address in memory operations and branches). For details on pipeline organization, we refer the reader to [HP90].

	single inheritance		multiple inheritance	
	static typing	dynamic typing	static typing	dynamic typing
LC	h_{LC} * (6+2L+B) +(1-h_{LC}) * $miss_{LC}$			
VTBL	1+2L+B	N/A	see single inher.	N/A
SC	1+2L+B	2+2L+B	see single inher.	see single inher.
RD	1+2L+B	3+2L+B	see single inher.	see single inher.
CT	s*(1+3L+B)+(1-s)* (2+3L+B+e(3 +pB))	s*(4+3L+B)+(1-s)* (2+3L+B+e(3 +pB))	N/A	N/A
IC	h_{IC} * (1+L) + (1-h_{IC}) * $miss_{IC}$			
PIC	m * (1+L) + (1-m) * (1+L+k(1 +pB)) + M * $miss_{PIC}$			

Table 6. P97

because it depends on instruction 1 (Figure 12). Similarly, instruction 5 can execute at *L + L* or *L + 2* (one cycle after the previous instruction), whichever is later. Since we assume *L > 1*, we retain *2L*. The schedule for P92 also shows that instruction 3 (which is part of the multiple inheritance implementation) is *free*: even if it was eliminated, instruction 5 could still not execute before *2L* since it has to wait for the result of instruction 2. Similarly, instruction 4 is free because it executes in the delay slot of the call (instruction 5).[1] As a result, VTBL incurs no overhead for multiple inheritance: both versions of the code execute in *2L + 2* cycles (see Table 4).

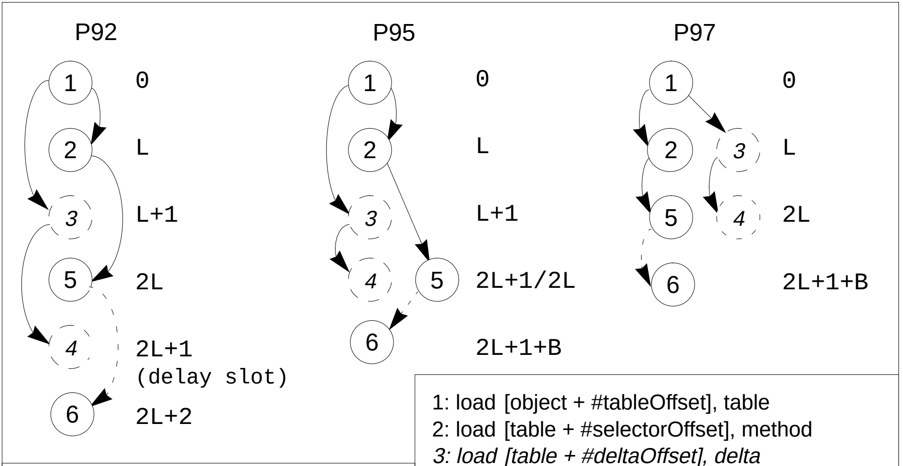


Figure 12. VTBL schedules, cycle counts and assembly code. Dependencies are indicated with arrows.

```
1: load [object + #tableOffset], table
2: load [table + #selectorOffset], method
3: load [table + #deltaOffset], delta
4: add object, delta, object
5: call method
6: <first instruction of target>
```

P95 (middle part of Figure 12) can execute two instructions per cycle (but only one of them can be a memory instruction, see Table 2). Unfortunately, this capability doesn't benefit VTBL much since its schedule is dominated by load latencies and the branch latency *B*. Since VTBL uses an indirect call, the processor does not know its target

[1] Recall that P92 machines had a branch latency B = 1, which can be eliminated using explicit branch delay slots; see [HP90] for details. Since we use a fixed branch penalty for P92, B does not appear as a parameter in Table 4.

address until after the branch executes (in cycle *2L*). At that point, it starts fetching new instructions, but it takes *B* cycles until the first new instruction reaches the EX (execute) stage of the pipeline [HP90], resulting in a total execution time of *2L+B+1*. Finally, P97 can execute up to 4 instructions per cycle, but again this capability is largely unused, except that instructions 2 and 3 (two loads) can execute in parallel. However, the final cycle count is unaffected by this change.

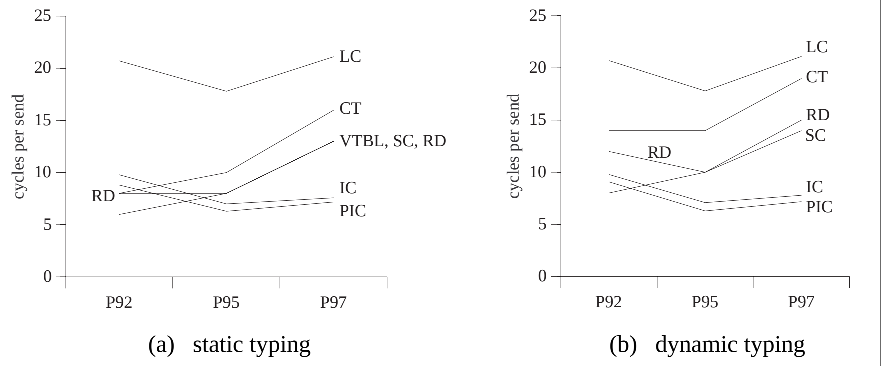


Figure 13. Performance of dispatch mechanisms (single inheritance)

Figure 13a shows the execution time (in processor cycles) of all dispatch implementations on the three processor models, assuming static typing and single inheritance. Not surprisingly, all techniques improve significantly upon lookup caching (LC) since LC has to compute a hash function during dispatch. The performance of the other dispatch mechanisms is fairly similar, especially on P95 which models current hardware. VTBL and SC are identical for all processors; RD and VTBL are identical for all but the P92 processor. Among these techniques, no clear winner emerges since their relative ranking depends on the processor implementation. For example, on P92 VTBL performs best and IC worst, whereas on P97 IC is best and VTBL is worst. (Section 5.4 will examine processor influence in detail.) For dynamic typing, the picture is qualitatively the same (Figure 13b).

5.3 Cost of multiple inheritance and dynamic typing

A closer look at Tables 4 to 6 and Figure 13 shows that supporting dynamic typing is surprisingly cheap for all dispatch methods, especially on superscalar processors like P95 and P97. In several cases (LC, IC, PIC), dynamic typing incurs no overhead at all. For the other techniques, the overhead is still low since the additional instructions can be scheduled to fit in instruction issue slots that would otherwise go unused. Typical overheads are two cycles per dispatch on P95 and one or two cycles on P97. Thus, on superscalar processors dynamic typing does not significantly increase dispatch cost.

The cost of supporting multiple inheritance is even lower. On P97, no technique incurs additional overhead for multiple inheritance, and only LC, RD, and IC incur a one-cycle overhead on P95. (However, recall that we have simplified the discussion of VTBL for

C++ by ignoring virtual base classes. Using virtual base classes can significantly increase dispatch cost in VTBL.)

Since the performance variations between the four scenarios are so small and do not qualitatively change the situation, we will only discuss the case using static typing and single inheritance in the remainder of the paper. The data for the other variations can be obtained from Table A-3 in the Appendix. (Of course, dynamic typing and multiple inheritance can affect other aspects of dispatch implementation; these will be discussed in section 6).

5.4 Influence of processor implementation

According to Figure 13, the cost (in cycles) of many dispatch techniques drops when moving from a scalar processor like P92 to a superscalar implementation like P95. Apparently, all techniques can take advantage of the instruction-level parallelism present in P95. However, when moving to the more aggressively superscalar P97 processor, dispatch cost *rises* for many dispatch techniques instead of falling further as one would expect.[1]

Figure 14a shows that the culprit is the penalty for mispredicted branches. It rises from 3 cycles in P95 to 6 cycles in P97 because the latter processor has a deeper pipeline in order to achieve a higher clock rate and thus better overall performance [HP90]. Except for the inline caching variants (IC and PIC), all techniques have at least one unpredictable branch even in the best case, and thus their cost increases with the cost of a branch misprediction. IC's cost increases only slowly because it has no unpredicted branch in the hit case, so that it suffers from the increased branch miss penalty only in the case of a inline cache miss. PIC's cost also increases slowly since monomorphic calls are handled just as in IC, and even for polymorphic sends its branches remain relatively predictable.

Based on this data, it appears that IC and PIC are attractive dispatch techniques, especially since they handle dynamically-typed languages as efficiently as statically-

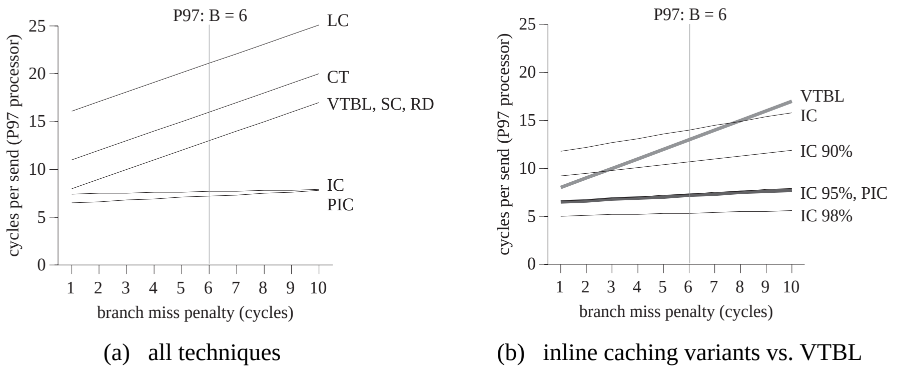


Figure 14. Influence of branch misprediction penalty on dispatch cost in P97

[1] Even though the number of cycles per dispatch increases, dispatch time will decrease since P97 will operate at a higher clock frequency. Thus, while the dispatch cost rises relative to the cost of other operations, its absolute performance still increases.

typed languages. However, one must be careful when generalizing this data since the performance of IC and PIC depends on several parameters. In particular, the dispatch cost of IC and PIC is variable—unlike most of the table-based techniques such as VTBL, the number of instructions per dispatch is not constant. Instead, dispatch cost is a function of program behavior: different programs will see different dispatch costs if their polymorphism characteristics (and thus their inline cache hit ratios) vary. The data presented so far assume a hit ratio of 95% which is typical for Smalltalk programs [Ung87] but may not represent other systems. For example, Calder et al. [CG94] report inline cache hit ratios for C++ programs that vary between 74% and 100%, with an average of 91%. Thus, the performance characteristics of IC and PIC deserve a closer investigation.

Figure 14b compares VTBL with PIC and IC for several inline cache miss ratios. As expected, IC's cost increases with decreasing hit ratio. If the hit ratio is 90% or better, IC is competitive with static techniques such as VTBL as long as the processor's branch miss penalty is high (recall that P97's branch miss penalty is 6 cycles). In other words, if a 91% hit ratio is typical of C++ programs, IC would outperform VTBL for C++ programs running on a P97 processor.

PIC outperforms VTBL independently of the processor's branch penalty, and it outperforms IC with less than a 95% hit ratio. The performance advantage can be significant: for P97's branch miss penalty of 6 cycles, PIC is twice as fast as VTBL. Again, this result is dependent on additional parameters that may vary from system to system. In particular, PIC's performance depends on the percentage of polymorphic call sites, the average number of receiver types tested per dispatch, and the frequency and cost of "megamorphic" calls that have too many receiver types to be handled efficiently by PICs. On the other hand, PIC needs only a single cycle per additional type test on P97, so that its efficiency is relatively independent of these parameters. For example, on P97 PIC is still competitive with VTBL if every send requires 5 type tests on average. As mentioned in section 3.3, the average degree of polymorphism is usually much smaller. Therefore, PIC appears to be an attractive choice on future processors like P97 that have a high branch misprediction cost.

Nevertheless, the worst-case performance of PIC is higher than VTBL, and PIC doesn't handle highly polymorphic code well, so some system designers may prefer to use a method with lower worst-case dispatch cost. One way to achieve low average-case dispatch cost with low worst-case cost is to combine IC with a static technique like VTBL, SC, or RD. In such a system, IC would handle monomorphic call sites, and the static technique would handle polymorphic sites. (Another variant would add PIC for moderately polymorphic call sites.) The combination's efficiency depends on the percentage of call sites that are handled well by IC. Obviously, call sites with only one target fall in this category but so do call sites whose target changes very infrequently (so that the rare IC miss doesn't have a significant performance impact). The scheme's dispatch cost is a linear combination of the two techniques' cost. For example, Calder's data [CG94] suggest that at least 66% of all virtual calls in C++ could be handled without misses by IC, reducing dispatch cost on P97 from 13 cycles for a pure VTBL implementation to 13* 0.34 + 4 * 0.66 = 5.6 cycles for VTBL+IC. In reality, the performance gain might be even higher since calls from call sites incurring very few misses could also be handled by IC. Even though this data is by no means conclusive, the potential gain in dispatch performance suggests that implementors should include such hybrid dispatch schemes in their list of dispatch mechanisms to evaluate.

5.5 Limitations

The above analysis leaves a number of issues unexplored. Three issues are particularly important: cache behavior, application code surrounding the dispatch sequence, and hardware prediction of indirect branches.

We do not consider memory hierarchy effects (cache misses); all results assume that memory references will always hit the first level memory cache. If all dispatch techniques have similar locality of reference, this assumption should not distort the results. However, without thorough benchmarking it remains unsubstantiated.

Application instructions surrounding the dispatch sequence (e.g., instructions for parameter passing) can be scheduled to fit in the "holes" of the dispatch code, lowering the overall execution time, and thus effectively lowering dispatch overhead. Therefore, measuring dispatch cost in isolation (as done in this study) may overestimate the true cost of dispatch techniques. Unfortunately, the effect of co-scheduling application code with dispatch code depends on the nature of the application code and thus is hard to determine. Furthermore, the average basic block length (and thus the number of instructions readily available to be scheduled with the call) is quite small, usually between five and six [HP90]. On superscalar processors (especially on P97) most dispatch sequences have plenty of "holes" to accommodate that number of instructions. Thus, we assume that most techniques would benefit from co-scheduled application code to roughly the same extent.

A branch target buffer (BTB) [HP90] allows hardware to predict indirect calls by storing the target address of the previous call, similar to inline caching. This study assumes that processors do not use BTBs; for most current RISC processors, this assumption holds because BTBs are relatively expensive (since they have to store the full target address, not just a few prediction bits) and because indirect calls are very infrequent in procedural programs.[1] However, future processors like P97 might incorporate BTBs since they will have enough transistors available to accommodate a reasonably-sized BTB; some processors (most notably, Intel's Pentium processor and its successor P6) have small BTBs today. Interestingly, BTBs behave like inline caches—they work well for monomorphic call sites but badly for highly polymorphic call sites. For example, the performance of VTBL on such a processor would be similar to the VTBL+IC scheme discussed above. The impact of BTBs on dispatch performance can be estimated by reducing the value of branch penalty B in the formulas of Tables 5 and 6, but the extent of the reduction depends on the BTB miss ratio (i.e., inline cache miss ratio) of the application.

[1] Procedure returns are the exception, but these can be handled more efficiently by a return address prediction buffer [Gw94].

6 Other considerations

Besides the actual speed of message sends, other considerations influence the choice between dispatch techniques. This section discusses the memory costs of dispatch schemes, and how amenable the schemes are to incremental change.

6.1 Memory cost

The space overhead of method dispatch falls into two categories: program code and dispatch data structures. Code overhead consists of the instructions required at call sites and in method prologues; stub routines (PIC & CT) are counted towards the data structure cost. The analysis below ignores per-instance memory costs (such as keeping a type field in each instance), although such costs can possibly dominate all other costs (e.g., if more than one VTBL pointer is needed for a class with a million instances). The space analysis uses the parameters shown in Table 3. Most parameter values are taken

Variable	Value	Comments
m	8,540	total number of methods; from Smalltalk
c	35,042	total number of call sites; from Smalltalk
M	178,264	total number of valid (receiver class, selector) pairs; from Smalltalk
e	4096	entries in LC lookup cache
O_{DTS}	133%	DTS overhead factor = #total entries / #non-empty entries; from Smalltalk
O_{SC}	175%	single inheritance overhead factor for SC; lower bound, from Smalltalk
O_{RD}	101%	single inheritance overhead factor for RD; from [DH95]
O_{CT}	15%	single inheritance compression rate for CT [VH94]
P_{SC}	no data	multiple inheritance overhead factor for SC[a]
P_{RD}	no data	multiple inheritance overhead factor for RD[b]
P_{VTBL}	no data	multiple inheritance overhead factor for CT[c]
k	3.2	average number of cases in a PIC stub; from SELF [Höl94]
f	7.2%	polymorphic call sites, as a fraction of total; from SELF [Höl94]
e	3.49	average number of functions in an overloaded entry (CT)
n	0.07%	overloaded entries, as fraction of total (CT)

Table 7. Parameters used for space cost analysis

[a] As shown in [AR92], the use of multiple inheritance introduces conflicts between selector colors that are hard to deal with and that substantially increase the overhead.
[b] Tables are harder to fit together because multiple inheritance causes more irregular empty regions to appear.
[c] Every time a class inherits from more than one superclass, overridden method entries are stored together with the appropriate delta's. This overhead depends entirely on the way multiple inheritance is used and is not quantifiable without appropriate code metrics.

from the ParcPlace Visualworks 1.0 Smalltalk system and thus model a fairly large application. For the multiple inheritance overhead we do not give typical values because there are none. The few samples in which multiple inheritance is extensively used in [DH95] show that the overhead varies much more than with single inheritance hierarchies (between 215% and 330% for VTBL), and that it is extremely dependent on how frequently MI is used. Therefore we do not give example space data for multiple inheritance. However, it is obvious in [DH95] that $P_{RD} < P_{VTBL} < P_{SC}$ for samples with extensive MI. Table 8 contains formulas for computing the space cost if the overhead is known.

Our analysis assumes that dispatch sequences are compiled in-line; the cost per call site is taken from the code sequences in Table A-2 in the Appendix.[1] Code for secondary techniques (like LC for IC) is not counted since it only appears once and thus should be negligible. Table 8 shows the space cost computation for all techniques. In the formulas, the symbols *D* and *C* refer to data and code cost; D_{LC}, for instance, refers to the data structure cost of LC in *the same column*.

	single inheritance				multiple inheritance			
	static typing		dynamic typing		static typing		dynamic typing	
	code	data	code	data	code	data	code	data
DTS	2c[a]	$2m*O_{DTS}$	same as SI-ST		same as SI-ST[b]		same as MI-ST	
LC	14c	$3e+D_{DTS}$	same as SI-ST		17c	$4e+D_{DTS}$	same as MI-ST	
IC	3c+2m	D_{LC}	same as SI-ST		5c+3m	D_{LC}	same as MI-ST	
PIC	3c+2m	$3kfc+D_{LC}$	same as SI-ST		5c+3m	$4kfc+D_{LC}$	same as MI-ST	
VTBL	2c	M	N/A		4c	$2M*P_{VTBL}$	N/A	
SC	2c	$M*O_{SC}$	3c+2m	$M*O_{SC}$	4c+m	$2M*P_{SC}$	5c+3m	$2M*P_{SC}$
RD	5c	$M*O_{RD}$	5c+4m	$M*O_{RD}$	7c+m	$2M*P_{RD}$	7c+5m	$2M*P_{RD}$
CT	4c	$M*O_{CT}$ *(1+en)	4c+7m	$M*O_{CT}$ *(1+en)	N/A			

Table 8. Formulas for approximate space cost (in words)

[a] Two instructions (sethi and setlo) to pass the selector to the lookup routine that actually implements the dispatch table search.

[b] Actually, there is a small overhead involved. Every class needs to store the representational offsets of its ancestors. This is much cheaper than storing an offset for every method understood.

Figure 15 shows the space costs for single inheritance versions of the dispatch techniques, using the classes and methods of the ParcPlace Visualworks 1.0 Smalltalk system as an example. Surprisingly, the code space overhead dominates the overall space cost for six of the eight techniques. Most of that overhead consists of the per-call dispatch code sequence. Much of the literature has been concentrated on the size of dispatch tables, treating call code overhead as equivalent among different techniques ([Ung87] is a notable exception). As demonstrated by the above data, minimizing dispatch tables may not reduce the overall space cost if it lengthens the calling sequence, especially in languages with a high density of message sends, like Smalltalk.[2] Code size can be reduced for most techniques by moving some instructions from the caller to the callee, but only at the expense of a slower dispatch. (LC's code size requirements could be dramatically reduced by doing the lookup out-of-line.)

The size of the immediate field in an instruction significantly impacts the code cost of SC, RD, and CT. This study assumes a 13-bit signed immediate field, limiting the range

[1] We choose not to include the call instruction in each dispatch sequence in the space cost since this instruction is required for direct function calls as well. To include the call instructions, just add c to each entry in Table 8.

[2] Increased code size can also impair execution performance by causing more instruction cache misses.

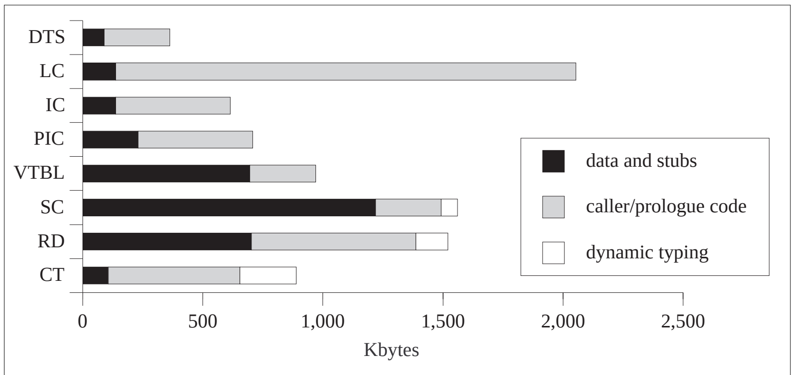


Figure 15. Space overhead for the Smalltalk system

of immediates to -4096..4095.[1] The Smalltalk system measured had 5087 selectors, and thus the selector number fits into an immediate. SC needs only one instruction to load the selector code into a register (see Table A-2), but RD takes two instructions for the same action because the selector offset needs two more bits (both zero) to address a word-aligned method. The same phenomenon increases the method prologue overhead in both RD and CT.[2] In RD, the reduction in data structure size relative to SC is almost offset by a corresponding increase in code size. The data in Figure 15 are thus relative to the processor architecture. For example, for an architecture with larger immediates (or for smaller applications), CT's space advantage over VTBL would double. Of course, the data also depends on application characteristics such as the proportion of call sites versus number of classes, selectors, and methods.

Given these admonitions, IC and PIC apparently combine excellent average speed with low space overhead. The bounded lookup time of SC and RD is paid for with twice as much memory; VTBL is about one third smaller than those two. CT's small data structure size is offset by its code cost.

VTBL, RD, and SC require significantly more data space than DTS because they duplicate information. Each class stores all the messages it *understands*, instead of all the messages it *defines*. For example, in the Smalltalk system a class inherits 20 methods for each one it defines [Dri93b], so the number of entries stored in the class' dispatch table increases by a factor of 20.

Dynamic typing makes a relatively small difference in space cost. Dynamic techniques have no extra overhead because each dispatch already contains a run-time check to test for the cache hit. Static techniques[3] perform the run-time type check in the method prologue, so the overhead grows linearly with the number of defined methods, which is much smaller than the number of call sites.

[1] The size of immediates varies from architecture to architecture: for example, SPARC has 13 bits, Alpha 8 bits, and MIPS 16 bits.

[2] Here the crucial quantity is the number of bits necessary to represent a *cid* (16 bits for the Smalltalk example). For the same reason, CT's dynamic typing cost is higher.

[3] Excluding VTBL, which only works for statically-typed languages.

6.2 Other aspects

The choice of a dispatch technique is influenced by considerations other than space cost and execution speed. A detailed discussion of these factors is beyond the scope of this paper, so we will only briefly mention some of them.

- *Responsiveness*. Some static techniques are less suitable for interactive programming environments that demand a high level or responsiveness (i.e., immediate feedback to programming changes). Most static techniques need to recompute some data structures after a programming change. For example, introducing a new method name forces VTBL to rearrange the dispatch table of the affected class and all its subclasses. SC, RD and CT even have to completely recompute their tables from scratch (except in rare cases, e.g., if the new method fits into an empty slot in the dispatch table). For the Smalltalk system, this recomputation can take hours for SC, tens of seconds for RD, and seconds for CT.
- *Support for run-time extensibility.* Many static techniques (e.g., SC, RD, CT) presume knowledge of the entire program, i.e., knowledge of all classes and methods. With dynamic link libraries, the complete set of classes and methods is not known until run time since the classes contained in dynamic libraries are unknown until they are loaded at program start-up time or later during the execution of the program. Thus, these techniques have to recompute the dispatch data structures each time a new library is dynamically loaded.
- *Sharing code pages*. Some operating systems (e.g., Unix) allow processes executing the same program to share the memory pages containing the program’s code. With shared code pages, overall memory usage is reduced since only one copy of the program (or shared library) need be in memory even if it is used concurrently by many different users. For code to be shared, most operating systems require the code pages to be read-only, thus disallowing techniques that modify program code on-the-fly (e.g., IC and PIC).

Many of the dispatch techniques discussed here can be modified to address problems such as those outlined above. For example, static techniques can be made more incremental by introducing extra levels of indirection at run-time (e.g., by loading the selector number rather than embedding it in the code as a constant), usually at a loss in dispatch performance. For this study, only the simplest and fastest version of each technique was considered, but any variant can be analyzed using the same evaluation methodology.

7 Related work

Rose [Ros88] analyzes dispatch performance for a number of table-based techniques, assuming a RISC architecture and a scalar processor. The analysis included both dispatch and tag checking code sequences. The study considers some architecture-related performance aspects such as the limited range of immediates in instructions. Other studies have analyzed the performance of one or two dispatch sequences. For example, Ungar [Ung87] analyzes the performance of IC, LC, and no caching on SOAR, a RISC-processor designed to run Smalltalk. Driesen [Dri93b] analyzes algorithmic issues of a number of dispatch techniques for dynamically-typed languages, but without taking processor architecture into account. Hölzle et al. [HCU91] compare IC and PIC for the SELF system running on a scalar SPARC processor. Milton and Schmidt [MS94] compare the performance of VTBL-like techniques for Sather. None of these studies takes superscalar processors into account.

Calder et al. [CG94] discuss branch misprediction penalties for indirect function calls in C++. Their measurements of several C++ programs indicate that inline caching might be effective for many C++ programs (although measurements by Garrett et al. [G+94] are somewhat less optimistic). Calder et al. propose to improve performance with "if-conversion," an inline cache with a statically determined target. For each call site the address of the most frequently called function is determined from execution profiles.

We have considered single dispatch only; multiple dispatch techniques are discussed in [KR90] and [AGS94]. However, singly-dispatched calls are so frequent even in systems offering multiple dispatch that implementations usually special-case these calls. Ingalls [Ing86] shows how to implement multiple dispatch with a sequence of single dispatch, but such implementations may not be optimal [AGS94].

Dispatch overhead can also be reduced by eliminating dispatches (rather than just making them fast). For example, the SELF-93 system inlines 95% of all dispatches [Höl94] with compiler optimizations such as customization [CUL89] and type feedback [HU94]. Similarly, concrete type inference [OPS92, VHU92, APS93, PC94, AH95] or link-time optimizations [App88, Fer95] can determine the concrete receiver types of calls, possibly eliminating dynamic dispatch for many sends.

8 Conclusions

We have evaluated the dispatch cost of a range of dispatch mechanisms, taking into account the performance characteristics of modern pipelined superscalar microprocessors. On such processors, objectively evaluating performance is difficult since the cost of each instruction depends on surrounding instructions and the cost of branches depends on dynamic branch prediction. In particular, some instructions may be "free" because they can be executed in parallel with other instructions, and unpredictable conditional branches as well as indirect branches are expensive (and likely to become more expensive in the future). On superscalar architectures, counting instructions to estimate performance is highly misleading. We have studied dispatch performance on three processor models designed to represent the past (1992), present (1995), and future (1997) state of the art in processor implementation.

We have analyzed the run-time performance of dispatch mechanisms as a function of processor characteristics such as branch latency and superscalar instruction issue, and as a function of system parameters such as the average degree of polymorphism in application code. The resulting formulas allow dispatch performance to be computed for a wide range of possible (future) processors and systems. In addition, we also present formulas for computing the space cost of the various dispatch techniques. Our study has produced several results:

- The relative performance of dispatch mechanisms varies strongly with processor implementation. Whereas some mechanisms become relatively more expensive (in terms of cycles per dispatch) on more aggressively superscalar processors, others become less expensive. No single dispatch mechanism performs best on all three processor models.

- Mechanisms employing indirect branches (i.e., all table-based techniques) may not perform well on current and future hardware since indirect branches incur multi-cycle pipeline stalls, unless a branch target buffer is present. Inline caching variants pipeline very well and do not incur such stalls. On deeply pipelined superscalar processors like the P97, inline caching techniques may substantially outperform even the most efficient table-based techniques.
- Hybrid techniques combining inline caching with a table-based method may offer both excellent average dispatch cost as well as a low worst-case dispatch cost.
- On superscalar processors, the additional cost of supporting dynamic typing or multiple inheritance is small (often zero) because the few additional instructions usually fit into otherwise unused instruction issue slots.
- Instructions (in particular, per-call code) can contribute significantly to the overall space requirements of message dispatch. In our example system, many techniques spend more memory on dispatch code sequences than on dispatch data structures. Thus, minimizing dispatch table size may not always be the most effective way to minimize the overall space cost, and may in some cases even increase the overall space cost.

Even though selecting the best dispatch mechanism for a particular system is still difficult since it involves many factors, the data presented here should allow dispatch speed and space costs to be accurately estimated for a wide range of systems. Therefore, we hope that this study will be helpful to system implementors who need to choose the dispatch mechanism best suited to their needs.

Acknowledgments. The authors would like to thank Lars Bak, Kris De Volder, Brian T. Lewis, Ole-Lehrmann Madsen, Patrick Steyaert, and Mario Wolczko for their comments on earlier versions of this draft.

9 References

[APS93] Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. Type Inference of SELF: Analysis of Objects with Dynamic and Multiple Inheritance. In *ECOOP '93 Conference Proceedings*, p. 247-267. Kaiserslautern, Germany, July 1993.

[AH95] Ole Agesen and Urs Hölzle. *Type Feedback vs. Concrete Type Inference: A Comparison of Optimization Techniques for Object-Oriented Languages*. Technical Report TRCS 95-04, Department of Computer Science, UCSB, March 1995.

[AGS94] Eric Amiel, Olivier Gruber, and Eric Simon. Optimizing Multi-Method Dispatch Using Compressed Dispatch Tables. In *OOPSLA '94 Conference Proceedings*, pp. 244-258, October 1994. Published as SIGPLAN Notices 29(10), October 1994.

[AR92] P. André and J.-C. Royer. Optimizing Method Search with Lookup Caches and Incremental Coloring. *OOPSLA '92 Conference Proceedings*, Vancouver, Canada, October 1992. Published as SIGPLAN Notices 27(10), October 1992.

[App88] Apple Computer, Inc. *Object Pascal User's Manual*. Cupertino, 1988.

[CG94] Brad Calder and Dirk Grunwald. Reducing Indirect Function Call Overhead in C++ Programs. In *21st Annual ACM Symposium on Principles of Programming Languages*, p. 397-408, January 1994.

[CUL89] Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *OOPSLA '89 Conference Proceedings*, p. 49-70, New Orleans, LA, October 1989. Published as SIGPLAN Notices 24(10), October 1989.

[CU+91] Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle. Parents are Shared Parts: Inheritance and Encapsulation in SELF. *Lisp and Symbolic Computation* 4(3), Kluwer Academic Publishers, June 1991.

[CPL83] T. Conroy and E. Pelegri-Llopart. An Assessment of Method-Lookup Caches for Smalltalk-80 Implementations. In [GR83].

[Cy90] Cypress Semiconductors. *CY7C601 SPARC processor*, 1990.

[DDH84] P. Dencker, K. Dürre, and J. Heuft. Optimization of Parser Tables for Portable Compilers. *TOPLAS* 6(4):546-572, 1984.

[DH95] Karel Driesen, Urs Hölzle. Minimizing Row Displacement Dispatch Tables. Technical Report TRCS 95-05, Department of Computer Science, UCSB, March 1995.

[DM73] O.-J. Dahl and B. Myrhaug. *Simula Implementation Guide*. Publication S47, Norwegian Computing Center, Oslo, March 1973.

[DS84] L. Peter Deutsch and Alan Schiffman. Efficient Implementation of the Smalltalk-80 System. *Proceedings of the 11th Symposium on the Principles of Programming Languages*, Salt Lake City, UT, 1984.

[D+89] R. Dixon, T. McKee, P. Schweitzer, and M. Vaughan. A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance. *OOPSLA '89 Conference Proceedings*, pp. 211-214, New Orleans, LA, October 1989. Published as SIGPLAN Notices 24(10), October 1989.

[Dri93a] Karel Driesen. Selector Table Indexing and Sparse Arrays. *OOPSLA '93 Conference Proceedings*, p. 259-270, Washington, D.C., 1993. Published as SIGPLAN Notices 28(10), September 1993.

[Dri93b] Karel Driesen. *Method Lookup Strategies in Dynamically-Typed Object-Oriented Programming Languages*. Master's Thesis, Vrije Universiteit Brussel, 1993.

[Dus89] P. Dussud. TICLOS: An implementation of CLOS for the Explorer Family. *OOPSLA '89 Conference Proceedings*, pp. 215-220, New Orleans, LA, October 1989. Published as SIGPLAN Notices 24(10), October 1989.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1990.

[Fer95] Mary F. Fernandez. Simple and effective link-time optimization of Modula-3 programs. To appear in *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, 1995.

[G+94] Charles D. Garrett, Jeffrey Dean, David Grove, and Craig Chambers. *Measurement and Application of Dynamic Receiver Class Distributions*. Technical Report CSE-TR-94-03-05, University of Washington, February 1994.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Second Edition, Addison-Wesley, Reading, MA, 1985.

[Gw94] Linley Gwennap. Digital leads the pack with 21164. *Microprocessor Report* 8(12), September 12, 1994.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc. 1990.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches. In *ECOOP '91 Conference Proceedings*, Geneva, 1991. Published as Springer Verlag Lecture Notes in Computer Science 512, Springer Verlag, Berlin, 1991.

[HU94] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *PLDI '94 Conference Proceedings*, pp. 326-335, Orlando, FL, June 1994. Published as SIGPLAN Notices 29(6), June 1994.

[Höl94] Urs Hölzle. *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming*. Ph.D. Thesis, Technical Report STAN-CS-TR-94-1520, Department of Computer Science, Stanford University, 1994.

[Ing86] Daniel H. Ingalls. A simple technique for handling multiple polymorphism. *OOPSLA '86 Conference Proceedings*, p. 347-349, Portland, OR., 1986. Published as SIGPLAN Notices 21(11), November 1986.

[KR90] Gregor Kiczales and Louis Rodriguez. Efficient Method Dispatch in PCL. *Proc. ACM Conference on Lisp and Functional Programming*, 1990. Also in [Pae93].

[Kra83] Glenn Krasner. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, Reading, MA, 1983.

[Kro85] Stein Krogdahl. Multiple inheritance in Simula-like languages. *BIT* 25, p. 318-326, 1985.

[MS94] S. Milton and Heinz W. Schmidt. *Dynamic Dispatch in Object-Oriented Languages*. Technical Report TR-CS-94-02, The Australian National University, Canberra, January 1994.

[M92] MIPS Inc. *R4000* Technical Brief, 1992.

[M94] MIPS Inc. *R10000* Technical Brief, September 1994.

[OPS92] Nicholas Oxhøy, Jens Palsberg, and Michael I. Schwartzbach. Making Type Inference Practical. In *ECOOP '92 Conference Proceedings*, p. 329-349, Utrecht, The Netherlands, June/July 1992

[Pae93] Andreas Paepcke (ed.). *Object-Oriented Programming: The CLOS Perspective*, MIT Press, 1993.

[PC94] John Plevyak and Andrew A. Chien. Precise concrete type inference for object-oriented languages. In *OOPSLA '94 Conference Proceedings*, pp. 324-340, October 1994. Published as SIGPLAN Notices 29(10), October 1994.

[PW90] William Pugh and Grant Weddell. Two-Directional Record Layout for Multiple Inheritance. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, p. 85-91, White Plains, NY, June, 1990. Published as SIGPLAN Notices 25(6), June 1990.

[Ros88] John Rose. Fast Dispatch Mechanisms for Stock Hardware. *OOPSLA'88 Conference Proceedings*, p. 27-35, San Diego, CA, November 1988. Published as SIGPLAN Notices 23(11), November 1988.

[UP83] David Ungar and David Patterson. Berkeley Smalltalk: Who Knows Where the Time Goes? In [GR83].

[Ung87] David Ungar. *The Design and Evaluation of a High-Performance Smalltalk System*. MIT Press, Cambridge, MA, 1987.

[UP87] David Ungar and David Patterson. What Price Smalltalk? In *IEEE Computer* 20(1), January 1987.

[VH94] Jan Vitek and R. N. Horspool. Taming Message Passing: Efficient Method Look-Up for Dynamically-Typed Languages. In *ECOOP '94 Conference Proceedings*, Bologna, Italy, 1994.

[VH95] Jan Vitek and R.N. Horspool, *Fast Constant-Time Type Inclusion Tests*, Unpublished manuscript, 1995.

[Vit95] Jan Vitek. *Compact Dispatch Tables for Dynamically-Typed Object-Oriented Languages*. M.S. Thesis, University of Victoria, B.C., forthcoming.

[VHU92] Jan Vitek, R. N. Horspool, and J. Uhl. Compile-time analysis of object-oriented programs. *Proc. CC'92, 4th International Conference on Compiler Construction*, pp. 236-250, Paderborn, Germany, Springer-Verlag, 1992.

Appendix A. Detailed data

R1		a register (any argument without #)
#immediate		an immediate value (prefix #)
load	[R1+#imm], R2	load the word in memory location R1+#imm to register R2
store	R1,[R2+#imm]	store the word in register R1 into memory location R2+#imm
setlo	#imm, R1	set the least significant part of register R1 to #imm, the rest to 0
sethi	#imm, R1	set the most significant part of R1 to #imm[a]
xor	R1, R2, R3	bit-wise xor on register R1 and R2[b]. Result is put in R3
and	R1, R2, R3	bit-wise and on register R1 and R2. Result is put in R3
add	R1, R2, R3	add register R1 to R2 and put result in R3
call	R1	jump to address in R1 (can also be immediate), saves return address
comp	R1, R2	compare value in register R1 with R2 (R2 can be immediate)
bne	#imm	if last compare is not equal, jump to #imm[c]
jump	R1	jump to address in R1 (can also be immediate)

Table A-1. Abstract instruction set for Table A-2

[a] sethi always occurs after a setlo in our code. We use the same string to indicate the upper and lower part of #imm, to indicate which instructions depend on the bit length of a particular value.
[b] Operands of arithmetic and logic instructions can be immediates, if the bit length permits.
[c] PC-relative. We do not go into such details, unless they affect code size and/or speed.

SC: Selector Coloring	load [object + #tableOffset], table *store adjusted, [stackptr + #deltaOffset]* load [table + #colorOffset], method *load [table + #deltaOffset], delta* **setlo #selector, selector** call method **comp selector, #methSelector** *add delta, object, adjusted* **bne #messageNotUnderstood**	IC: Inline Caching	load [object + #classOffset], actualClass setlo #class, predictedClass *store adjusted, [stackptr + #deltaOffset]* *add object, #delta, adjusted* call #method[a] comp actualClass, predictedClass bne #inlineCacheMiss
RD: Row Displacement	load [object + #tableOffset], table setlo #selector, selector sethi #selector, selector[b] add table, selector, table load [table], method *store adjusted, [stackptr + #deltaOffset]* *load [table + #deltaOffset], delta* call method **sethi #methSelector, thisSelector** **setlo #methSelector, thisSelector** **comp selector, thisSelector** *add delta, object, adjusted* **bne #messageNotUnderstood**	PIC: Polymorphic Inline Caching	load [object + #classOffset], actualClass comp actualclass, #predictedClass1 *store adjusted, [stackptr + #deltaOffset]* *add object, #delta1, adjusted* call #PIC_nnn[a] bne #2nd jump #method1 2nd: *add object, #delta2, adjusted* comp actualClass, #predictedClass2 bne #3rd jump #method2 3rd: ... last: jump #PICmiss

Table A-2. Instruction sequences

LC: Lookup Caching	load [object + #classOffset], class setlo #cacheAddr, cache sethi #cacheAddr, cache setlo #selectorCode, selector xor class, selector, index and index, #mask, index add cache, index, cache load [cache], cacheClass *store adjusted, [stackptr + #deltaOffset]*[c] load [cache + 4], cacheSelector load [cache + 8], cacheTarget *load [cache + 12], delta* comp class, cacheClass bne #miss comp selector, cacheSelector bne #miss *add delta, object, adjusted* call cacheTarget
VTBL	load [object + #tableOffset], table load [table + #selectorOffset], method *load [table + #deltaOffset], delta* *add object, delta, object* call method (works only for statically-typed languages)
CT: Compact selector-indexed Tables	load [object + #classOffset], class load [class + #tableOffset], table load [class + #cidOffset], cid load [table + #selector], method call method single-implementation (method prologue): **setlo #mask, mask** **sethi #mask, mask** **and mask, cid, cid** **setlo #thisClass, thisClass** **sethi #thisClass, thisClass** **comp thisClass, cid** **bne #messageNotUnderstood** overloaded: setlo #mask1, mask1 sethi #mask1, mask1 and mask1, cid, temp setlo #cid1, cid1 sethi #cid1, cid1 comp cid1, temp bne #2nd jump #method1 2nd: setlo #mask2, mask2 sethi #mask2, mask2 and mask2, cid, temp setlo #cid2, cid2 sethi #cid2, cid2 comp cid2, temp bne #3nd jump #method2 3rd: ... last: jump #messageNotUnderstood

Table A-2. Instruction sequences

[a] The message selector is placed right after this call instruction. This doesn't have any effect on speed, but does increase the code size by one word.

[b] selector offset is unlikely to fit in immediate field of instructions in large applications (see text).

[c] In all code sequences the adjusted receiver address is saved before a call (except in VTBL where it can be reconstructed because every class/superclass combination has its own table with deltas). Actually, this store is only necessary if instance variables can be accessed after the call. Thus, it may be optimized away by the compiler. We do not show the corresponding load, because it is likely to be hidden in the code following the call (on a superscalar processor).

	P92				P95				P97			
	single inh.		multiple inh.		single inh.		multiple inh.		single inh.		multiple inh.	
	static	dyn.	static	dyn.	static	dyn.	static	dyn.	static	dyn.	static	dyn.
LC	20.7		23.6		17.8		18.8		21.1			
VTBL	6.0	N/A	6.0	N/A	8.0	N/A	8.0	N/A	13.0	N/A	13.0	N/A
SC	6.0	8.0	6.0	9.0	8.0	10.0	8.0	10.0	13.0	14.0	13.0	14.0
RD	8.0	12.0	10.0	12.0	8.0	10.0	9.0	12.0	13.0	15.0	13.0	15.0
CT	8.0	14.0	N/A		10.0	14.0	N/A		16.0	19.0	N/A	
IC	9.8		11.9		7.1		8.1		7.8			
PIC	8.8		10.5		6.3				7.2			

Table A-3. Dispatch timings (in cycles)

	single inheritance					
	static typing			dynamic typing		
	code	data	sum	code	data	sum
DTS	274	89	363	same as SI-ST		363
LC	1,916	137	2,053	same as SI-ST		2,053
IC	477	137	614	same as SI-ST		614
PIC	477	231	708	same as SI-ST		708
VTBL	274	696	970	N/A		
SC	274	1,219	1,493	341	1,219	1,560
RD	684	703	1,387	817	703	1,520
CT	548	107	655	782	107	889

Table A-4. Approximate space cost for dispatch in Smalltalk image (in Kbytes)