Efficient Method Dispatch in PCL

Gregor Kiczales[1] Luis Rodriguez[2]

September 7, 1992

[1]gregor@parc.xerox.com
[2]lhr@altdorf.ai.mit.edu

14.1 Introduction

Efficient implementation of CLOS is critical to its success as a standard. Some excellent work on Lisp Machines [3, 7] has demonstrated clearly that CLOS can be implemented efficiently using special-purpose hardware.

We describe a mechanism for implementing CLOS method dispatch efficiently on stock hardware, in the current generation of Common Lisp implementations. This mechanism is implemented in the newest version of PCL, a portable implementation of CLOS, and runs in more than ten Common Lisps.[1]

PCL is designed to support development, not delivery of CLOS programs. The goal is to provide not just high run-time performance, but also rapid interactive response during development of CLOS programs.

This work is based on a careful analysis of the behavior of existing CLOS programs. The method dispatch mechanism differs from previously published work in three important ways. First, the use of a new hashing algorithm improves memoization table density and distribution. Second, the selection of memoization table format based on the dynamic history of each generic function makes it possible to store information in the memoization tables more efficiently and do the run-time method dispatch more quickly. Third, lazy updating techniques are used to speed interactive programming environment response without undue degradation of program execution.

The portable nature of PCL is itself the cause of a number of efficiency problems. While these are interesting, they are not the subject we of our discussion. Instead, we focus on the CLOS implementation architecture that is the heart of PCL. That architecture, which would be easy to implement in any *single* Common Lisp, is the lasting value of this work. The trials and tribulations of implementing this architecture in ten different Common Lisps with no sources is best left to fireside chats.

We begin by presenting a very simple run-time method dispatcher. Its simplicity allows it to be very fast, particularly when the distribution in the memoization table is good. We then show how to ensure nearly perfect distribution in every memoization table without slowing down the run-time method dispatch. After presenting our analysis of the behavior of CLOS programs, we use that analysis to develop certain special cases of memoization table representation. These special cases save space in the table and increase the speed of the run-time method dispatch. We also show how lazy updating techniques can be used to extend this mechanism to support interactive program development.

14.2 Instance Implementation

In PCL, instances of user-defined classes[2] are represented as shown in figure 14.1. An instance is a two element structure. The first element is a *class wrapper* and the second element is a *slot vector*. The slot vector is a simple vector that contains the values of the slots of the instance. The class wrapper is itself another two element structure. Its first element is a fixnum called the *hash seed*. This is used for hashing in the memoization tables.

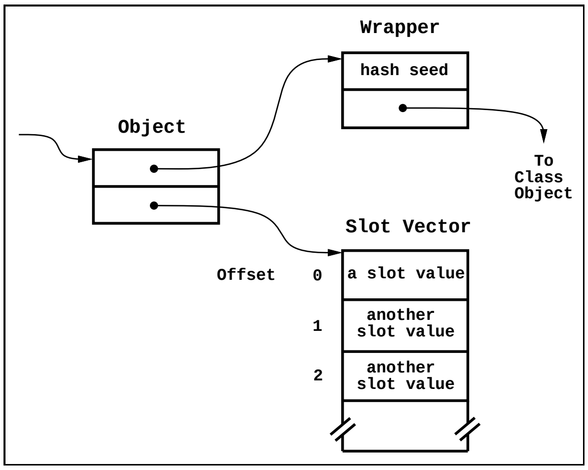

Figure 14.1: PCL's representation of objects.

Its second element is the class object itself. Except for the situations discussed in the section on updating, all instances of a class point to the same wrapper.

14.3 The Basic Run-Time Method Dispatch Mechanism

For simplicity in presentation, the next five sections assume generic functions with only one specialized argument. The section on multi-methods discusses the expansion of the method dispatch mechanism to include specialization on more than one argument.

When a generic function is called, we need to determine and call the appropriate effective method.[3] This determination is based on the class of the argument to the generic function and the set of methods defined on the generic function. In general, this can be a time-consuming process. Our goal is to improve the performance of CLOS programs by memoizing this method determination whenever possible.

In PCL, memoization works by associating a memoization table with each generic function. The table maps class wrappers to the appropriate effective method function for that class.

The basic run-time method dispatch mechanism is shown in figure 14.2. Each line of a memoization table can contain a class wrapper and the starting program counter (PC) of the appropriate compiled effective method function for that class.[4] An empty line is indicated by nil in the wrapper field; a non-empty line is called a *table entry*.

When the generic function is called, the method dispatch run-time attempts to find a table entry for the argument's wrapper. The hashing function is simple: the hash seed of the argument's wrapper is reduced modulo the size of the generic function's memoization table to yield the line for the initial probe. If the wrapper stored at that line is eq to the argument's

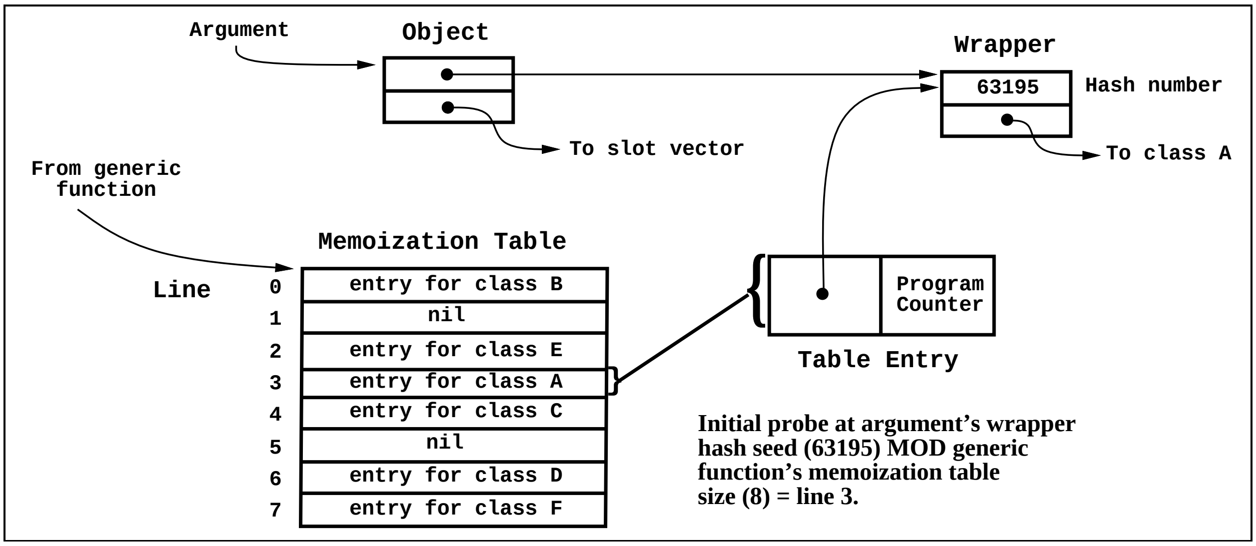

Figure 14.2: PCL's basic run-time method dispatch mechanism.

wrapper, this is the correct table entry and the method dispatch can jump directly to the corresponding PC. This is called an *initial probe hit* because the first line inspected contained the correct table entry.

Memoization tables can be implemented either as simple vectors or as untyped blocks of words. To improve the performance of run-time method dispatch, PCL always allocates memoization tables whose size is a power of 2, thereby enabling primitive logical instructions to be used for the hashing calculations.

If the initial probe does not contain the correct value the rest of the table is scanned sequentially, starting at the next line, and terminating back at the initial probe line. If no entry is found with the correct wrapper it is a *memoization table miss*; the memoization table filler must be called to compute the effective method and store the appropriate values in the table. This simple dispatch algorithm is chosen for its efficiency at getting to the initial and next one or two lines.

If we can enssure that most table entries are in their initial probe line, or at most one or two lines away, the simplicity of this dispatcher will be a performance advantage. In the next section we discuss refinements which make it possible to ensure such placement of the entries.

14.4 Adding New Entries to the Memoization Tables

For a given memoization table, we can define the *average probe depth* as the average distance each of the tables entries is from its initial probe line. Two entries which share the same initial probe line are said to *collide*. We also define the *memoization table density* as

type	*dynamic analysis number of calls*		*static analysis number of generic functions*	
	absolute	percentage	absolute	percentage
Reader-Method-Only	459671	69	257	28
n-Effective-Methods	113408	17	223	24
1-Effective-Method	47258	7	266	29
Multiple-Specialized-Arguments	23739	4	42	4
Writer-Method-Only	21804	3	140	15

Table 14.1: Dynamic data collected from CLOS applications.

that fraction of table lines that contain valid entries.

Our task is to find a way to place the entries in the table so as to minimize collisions and thereby reduce the average probe depth. We should do this without resorting to the obvious solution of making the tables arbitrarily large. In fact, working set considerations are so important that we would like to keep memoization tables as small as possible.

Our first step is to look at the allocation of wrapper hash seeds. Since these directly produce the initial probe line, improving the distribution of wrapper hash seeds should improve the distribution of initial probe lines. Other work [8] has shown that using a global mechanism to adjust the hash seeds can yield excellent memoization table distribution. In PCL, the requirements of rapid updating during program development led us to look for an alternative to such global mechanisms. Instead, the hash seed of a wrapper is allocated at the time the wrapper is created and is not adjusted later. By allocating these numbers randomly, we can achieve improved distribution and avoid cases where a given program reliably gets bad hashing behavior.

Even so, as the number of generic functions increases, the probability that there will be collisions among the entries of a given memoization table increases. We must further improve the hashing distribution.

If we increase the size class wrappers slightly, we can add more hash seeds to each wrapper. If n is the number of hash seeds stored in each wrapper, we can think of each generic function selecting some number x less than n and using the xth hash seed from each wrapper.[5] Currently we store 8 hash seeds in each wrapper, resulting in very low average probe depths.

The additional hash seeds increase the probability that a generic function will be able to have a low average probe depth in its memoization table. If one set of seeds doesn't produce a good distribution, the generic function can select one of the other sets instead. In effect, we are increasing the size of class wrappers in order to decrease the size of generic function memoization tables. This tradeoff is attractive since typical systems seem to have between three and five times as many generic functions as classes.

14.5 Run-Time Data

To investigate further optimization of the method dispatch mechanism, data was gathered from several large CLOS applications.[6] The data was gathered by logging information about every generic function call during runs of the applications. For each generic function call, the classes of its specialized arguments were recorded. The data is summarized in table 14.1.

Analysis of the data reveals several natural categories of generic functions. Each line in the table corresponds to one such category. The line labeled *reader-method-only* includes those generic functions which, in the course of running the application, only invoked automatically generated reader methods.[7] Similarly *writer-method-only* includes those generic functions which only invoked automatically generated writer methods. A *1-effective-method* generic function has run only one effective method throughout its history. An *n-effective-methods* generic function dispatched to more than one effective method. A *multiple-specialized-arguments* generic function is one that specializes on more than one argument — a new feature of CLOS which is not yet widely used.

It is important to remember that these categories are not derived from a static analysis of the methods on the generic function. Rather, they derive from the effective methods that the generic function actually calls during the execution of the applications. For example, the reader-method-only category would include a generic function with a default method that is never called and a reader method that is called. A generic function that has a number of possible effective methods, but only ever runs one of them, is in the 1-effective-method category.

The data shows that, by far, most generic functions calls were to reader-method-only generic functions. N-effective-methods generic functions were also called frequently. The remaining categories were called less frequently.

Note that some of the regularities in the data may be attributed to the fact that people are accustomed to programming in Loops and Flavors. Moreover, a large part of the analyzed code was automatically converted from Flavors. As the CLOS programming style matures, we would expect to see changes in the data, particularly an increase in the multi-method case. A process similar to the one presented in the next section can be followed to further adapt this memoization mechanism as new styles develop.

14.6 Specialized Run-Time Dispatch Procedures

From the data we can see that many generic functions only ever call one effective method. For such a generic function, every entry in the memoization table would have a different class wrapper but the same effective method start PC. This leads us to address the inefficiency of storing the same start PC repeatedly.

Rather than using a memoization table for which each entry has one key (a class wrapper) and one value (a start PC), 1-effective generic functions can use a memoization table in which the entries have only keys (class wrappers). The value common to each entry can be stored

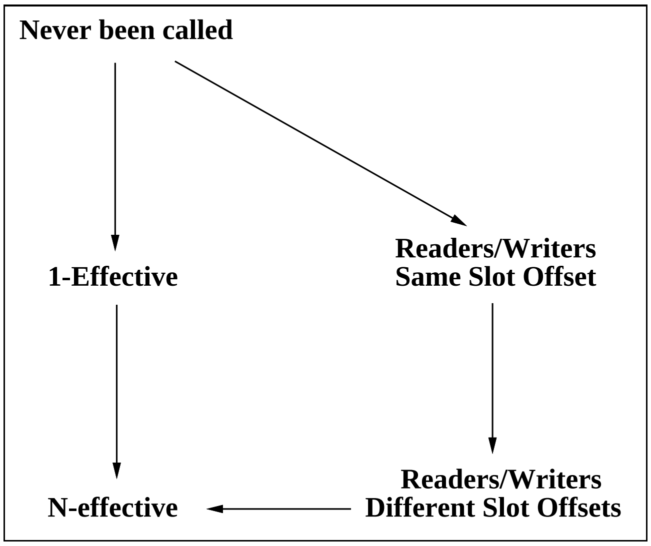

Figure 14.3: State transition diagram for PCL's single argument method dispatchers.

separately, thereby avoiding duplication in the table. In such a table the existence of a wrapper in the table serves to confirm that the one common effective method function is appropriate for that class of argument.

The data also shows that reader and writer generic functions represent an overwhelming majority of the dynamic generic function calls. This motivates us to consider further optimization of these generic functions.

The critical point is that the code for automatically generated reader methods (for simplicity we present only the reader case) is known to the implementation. It is possible to "pull" the code for the method back into the method dispatcher of the generic functions. This is done by storing the offset of the slot being accessed in the memoization table instead of the start PC of the method.

Having made the optimization to a special reader method dispatch, yet another optimization is possible, similar to the one which produces the 1-effective method dispatch. If all of the slot offsets stored in the table are the same, the shared value is stored separately instead.

Additional data, not shown in table 14.1, has shown that for many reader-method-only generic functions, the generic function is called with instances of only one or two distinct classes. For these generic functions, we can choose a method dispatch procedure which does no hashing but which simply checks the one or two class wrappers in question. For one or two wrappers, this kind of dispatch is faster than doing hash lookup and getting an initial probe hit.

The memoization table format and corresponding method dispatcher to use must be chosen dynamically based on which effective methods the generic function has run. This is easily accomplished by having the memoization table filler, in addition to taking care of expanding and filling the tables also take care of changing the table format and changing the method dispatcher when needed.

Whenever a new method is added to a generic function, we reset the kind of dispatcher and table it is using by setting it to a special initial state that will choose the correct table format based on the first call to the generic function. At each subsequent table fill, the table format will change if need be. The state transition diagram of dispatchers is shown in figure 14.3.

14.7 Updating

During program development, the programmer repeatedly changes definitions in the program. A CLOS implementation designed to support program development must provide rapid response to such changes. If the programmer changes the definition of a class (perhaps by editing and re-evaluating the corresponding defclass form), the response must be rapid, certainly no more than one or two seconds. This can be problematic because some changes can have wide-ranging effects.

In PCL, these kinds of redefinitions are handled using lazy updating techniques. The goal of these techniques is to improve interactive response by delaying, as much as possible, the work required to perform a redefinition. The work done at the time of redefinition is called *immediate updating*. The work done at later times is called *deferred updating*. The time to perform the immediate updating is the delay the programmer experiences when changing a definition. The goal is to reduce this time by deferring as much of the update as possible.

The use of lazy updating techniques is suggested by two important observations about the nature of interactive program development. The first is that changes often come in bunches. The programmer will often make more than one change to a program before attempting to test any of them. Also, the program is often run only briefly following a change; whereupon another change is then made and tested. In such situations, it can be wasteful to perform all of the updating associated with any one change immediately because the updating associated with the next change might have substantial overlap. Lazy updating can be more efficient because it prevents these redundant updates.

The second observation is that even in cases where lazy updating doesn't prevent redundant work it can still appear faster to the interactive user. This happens in development/debugging situations where what the programmer notices most is the response to initial redefinition. Degradation in performance which follows — while the deferred updating happens — is barely noticeable in such situations.[8]

An additional asset of the lazy updating techniques is their simplicity of implementation. Earlier work [3, 7] has shown that it is possible to handle these updates using complex mechanisms of redundant cross-referenced tables. Such code is difficult to write and debug. By contrast, the code required to implement the lazy updating mechanism in PCL is short and simple.

Changes to generic function or method definitions have localized effect and so are easy to handle. The only updating required is to correct affected entries in the memoization table of the generic function. The immediate redefinition of a generic function simply discards the entire memoization table and resets the method dispatch procedure to the *never been called* state shown in figure 14.3. The deferred updating happens during subsequent calls to the generic function when the memoization table is rebuilt by the normal table miss mechanism.

Changes to the definition of a class can have far-ranging effects and so can be much more difficult to handle. Given such a redefinition, the class and all of its subclasses are *affected classes*. The method applicability of any generic function with methods applicable to an affected class can change. If the slot inheritance of an affected class changes, all extant instances of the class must be updated before the next time they are used.

During immediate redefinition of each affected class, the only work done is to mark its wrapper as being invalid. This mark causes subsequent memoization table lookups which use the wrapper to miss. The miss mechanism will detect that deferred updating is required and perform it. The deferred updating will include some combination of the following three actions: update the actual class object, update memoization table entries, and update the instance.

The class object is updated the first time an invalid wrapper is encountered (or if a new instance is created). This involves computing the new inheritance of superclasses and slots and storing it in the class. A new class wrapper is also created for the class. This wrapper is used for all newly created instances, and as old instances are updated their wrapper field is changed to point to this wrapper. The presence of this new class wrapper is used to tell that the class object itself has been updated.

The first time a lookup in a given memoization table encounters a given invalid wrapper, the corresponding memoization table entry must be updated. This is done by deleting the entry with the invalid wrapper, and creating a new entry with the new wrapper and the correct value.

Each time an instance with an invalid wrapper is found, that instance is updated. If the instance will require more slots a new slot vector is created and used. The instance is marked as being updated by changing the wrapper field to point to the new wrapper. This happens as shown in figure 14.4.

In this way, the deferred update corresponding to a class redefinition is spread out over subsequent generic function calls involving instances of the affected classes. Over time, all the affected memoization tables and instances will be updated. A memoization table miss only occurs when some deferred update is required; otherwise the normal fast memoization table dispatch occurs.

This mechanism hingees on having each memoization table dispatch procedure check to see whether the wrapper involved is marked as invalid. If this check is time consuming, this deferred updating mechanism will degrade the performance of all method dispatches. By requiring each memoization table to have one empty entry, it is possible to arrange for the check for wrapper validity not to slow down the normal case of method dispatch.

Marking the wrapper as invalid is done by setting all hash seed fields of the wrapper to

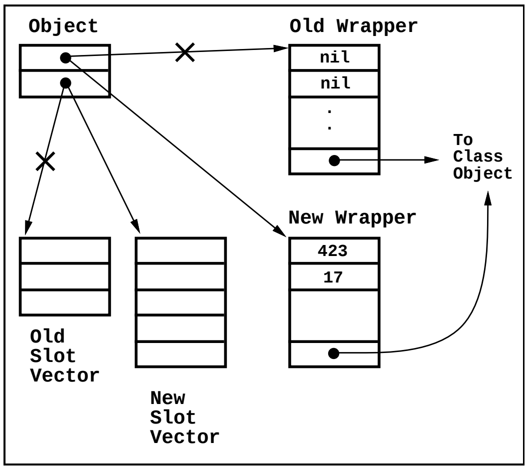

Figure 14.4: Updating an Instance

zero, a value which normal, valid wrappers never have. Additionally, the zero line in any memoization table, which corresponds to the initial probe line for a hash seed of zero, is always kept empty. This ensures that the initial probe with an invalid wrapper will never find the wrapper, which allows all method dispatch procedures to delay checking whether the wrapper is invalid until the second probe. Even when the check is actually performed it is fast; it simply involves testing whether the hash seed is zero.

14.8 Multi-Methods

Unlike other object-oriented languages, CLOS includes a mechanism for allowing method dispatch to be based on the class of more than one argument.

The implementation of method dispatch when multiple arguments are specialized is a simple extension of the single argument case. Each entry in the memoization table is expanded to include multiple wrappers, one for each specialized argument. The initial probe line is computed using a combination of the hash seeds from the wrapper of each specialized argument.

Computing the initial probe line is done by simply adding the hash seeds of each wrapper before masking them. Adding was chosen because, of the primitive logical instructions, it produces the best distribution of hash seeds.

In method dispatch with multiple specialized arguments checking for wrapper validity must be done by checking whether the hash seed of any of the wrappers is zero.

14.9 Multiple Processes

Implementing CLOS in the presence of multi-tasking introduces a number of complex issues. The one issue that has been addressed in the PCL work is to ensure that one task does not modify a memoization table entry while another task is reading it.

Unfortunately, the primitive process scheduling mechanisms provided as part of most Common Lisp implementations are not efficient enough to address these problems directly. In particular, the usual mechanism for implementing atomic operations (without-interrupts) is too inefficient to be used in the memoization table lookup operation.

In PCL, an alternative mechanism which requires without-interrupts to be used only during the non-performance-critical table filling operations, is used. This mechanism works by modifying the table consistency invariants slightly: filling of memoization tables must be an atomic operation; memoization table lookups are not atomic, but must ensure that the table is not filled during the lookup.

Checking that the table is not filled during a lookup is implemented using a number called the *table lock count* which is stored at the head of each memoization table. Each atomic filling operation increments this lock count (wrapping around every 2^{32} bits). Each table lookup operation reads the lock count before beginning the table lookup, and again when the lookup is complete to check that nothing has changed. If the table has changed during the lookup, the lookup is restarted, otherwise the entry read from the table is used.

This mechanism ensures memoization table consistency, but is not entirely satisfactory in its run-time performance. We would like to develop a technique in which memoization table lookup bears no overhead for ensuring table consistency. In the future we hope to explore the use of a mechanism in which memoization tables are immutable. This appears tractable because it is possible to have the immutability requirement only apply to existing entries. Entries could be added to a table. Only when shuffling the entries in a table would a new table be required.

14.10 Performance of this Mechanism

Several factors make it difficult to properly evaluate the performance of the PCL architecture[5, 6]. Because of its portable nature, direct comparisons between PCL and other implementations for various μ-benchmarks are misleading. Moreover, as has been shown repeatedly, the performance of a system on μ-benchmarks is not always indicative of its performance running real programs. As an example, when running large programs, working set considerations can make memoization table density a more important factor than raw method dispatch performance.

The best way to evaluate CLOS implementations is probably to compare the performance of several large applications running in them. We do not include such a comparison here; we hope to see these as more CLOS implementations come available.

Instead, we resort to two other kinds of performance information. The first comes from a simple set of μ-benchmarks which measure the performance of PCL method dispatch. The

test	*Franz 3.1.4*		*Lucid 3.0.1*	
	μsecs	ratio	μsecs	ratio
FUN	1.67	1	1.80	1
ACCESSOR1	3.50	2.0	7.90	4.4
ACCESSOR2	3.59	2.1	7.65	4.3
ACCESSOR3	4.67	2.8	12.77	7.1
G1	5.33	3.2	10.70	5.9
G2	5.17	3.1	10.25	5.7

Table 14.2: Performance of PCL method dispatch on the μ-benchmarks. All measurements are taken on a SUN 4/330. The Franz times reflect the fact that the Franz port of PCL includes a direct interface to the compiler back-end which allows a more direct and efficient implementation of the method dispatch.

second measures the cache density and average probe depth obtained using the PCL method dispatch architecture.

The generic function call performance benchmarks in the appendix are used to measure the performance of the special method dispatch procedures presented earlier. These benchmarks are nonetheless interesting for any CLOS implementation because the special method dispatch procedures were derived from an analysis of CLOS programs. These results are summarized in table 14.2. The numbers in the ratio column compare the time required to call generic functions to the time required to call an ordinary function.

These results also give some indication of the difference between the architectural PCL and the PCL that it is possible to express in Common Lisp. The Franz port of PCL includes a direct interface to the compiler back-end; accounting for the performance difference between the two Lisps. The development of this back-end for Lucid Lisp is not yet complete.

Average table density and probe depth results are shown in table 14.3. These measurements were made using CLIM, a large user-interface system. The results show that the average probe depths are between 1 and 2, indicating that most table entries are either in their initial probe or very next line. Given that PCL always uses table sizes that are powers of 2, the table densities are quite high. We plan to explore the possibility of using other table sizes, particularly for larger tables where doubling the size of a table when only one new line is needed is unreasonable.

14.11 Acknowledgements

We would like to thank Jim Bennet, Chris Burdorf, David Capshaw, Neil Goldman, Brett Kessler, Stan Lanning, Greg Meredith, Jeff Morrill, Rob Pettengill, Ramana Rao, Chris Sweeney and Rich Piazza for their time and partience in gathering statistics. We would like to thank Danny Bobrow, Mike Dixon, Patrick Dussud, David Moon, and Jim des Rivières

dispatcher type	*average probe depth*	*average table density*
Readers Method Only	1.6	.77
n Effective Methods	1.7	.67
Writers Method Only	1.5	.70
1 Effective Method	1.4	.72
Multiple Specialized Arguments	1.9	.70

Table 14.3: Average probe depth and table densities for each type of method dispatcher.

for their comments on earlier drafts of this chapter.

Notes

1. We are referring to Rainy Day PCL, released February 16th 1990. This version of PCL actually fails to conform to the CLOS specification in some minor ways, but these are of no consequence in our discussion.
2. We use the term *user-defined class* to refer to standard classes.
3. In CLOS, the method combination mechanism combines the set of methods applicable to a instances of a class into an *effective method*. The techniques used in PCL to produce the compiled effective method function are not discussed here.
4. This is an excellent example of the difference between the architectural PCL described here and what it is easy to implement portably. The optimization of storing the PC (rather than the compiled function) and jumping to it directly (rather than doing a funcall or apply) is an important one. It can reuse the stack frame, and by having it be a PC to a special entry point can elide redundant argument setup and checking. The PCL architecture supports this optimization. Unfortunately, no current port of PCL yet supports this.
5. Note that for a memoization table of size s, each 32 bits of wrapper hash seed is itself good for *floor(32/log(s))* wrapper hash seeds. PCL does not currently take advantage of this, but only for silly implementation reasons.
6. The applications include a window system, an expert system shell, and a large database system. The database system was originally written in Flavors, and was automatically converted to CLOS.
7. These are methods generated by the :accessor or :reader slot options to defclass; they simply read the value of one of the object's slots.
8. This argument must be made carefully. If the deferred updating were implemented in a way that paged against the programmer's real activity it would be annoying — this was the bane of early background garbage collectors. The technique used in PCL does not suffer from this problem because it updates only those instances and memoization tables

encountered in the course of running the program.

Bibliography

[1] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp Object System Specification. X3J13 Document 88-002R. (Also published in *Sigplan Notices*, Volume 23, Special Issue September 1988, and in Guy Steel: Common Lisp, The Language, 2nd ed., Digital Press, 1990)

[2] Craig Chambers, David Ungar, Elgin Lee. An Efficient Implementation of SELF, a Dynamically-Typed Object Oriented Language Based on Prototypes *OOPSLA Conference Proceedings*, October 1989.

[3] Patrick Dussud. TICLOS: An Implementation of CLOS for the Explorer Family. *OOPSLA Conference Proceedings*, October 1989.

[4] Adele Goldberg and David Robson. Smalltalk-80: The Language and its Implementation. *Addison Wesley*, 1983.

[5] Gregor J. Kiczales. CLOS Speed. Appears in Common Lisp Object System. Kyouritsu Publishing Company, November 1988

[6] Gregor J. Kiczales. Evaluating the Performance of CLOS Programs. Electronic message to CommonLoops and Common-Lisp mailing lists, Fall 1989.

[7] David A. Moon. Object-Oriented Programming with Flavors. *OOPSLA Conference Proceedings*, October 1986.

[8] David A. Moon. Personal communication.

Benchmark Code

```
;;;
;;; Some simple micro-benchmarks for CLOS.
;;;


(in-package 'pcl)

(proclaim '(optimize (speed 1) (safety 3)))

(defun fun (x) 0)

(defclass c1 ()
     ((x :initform 0
         :accessor accessor1
         :accessor accessor2
         :accessor accessor3)))

(defclass c2 (c1)
     ())

(defclass c3 (c1)
     ())

(defmethod g1 ((f c1)) 0)

(defmethod g2 ((f c1)) 0)
(defmethod g2 ((b c2)) 0)

(defvar *outer-times* 3)
(defvar *inner-times* 100000)

(defmacro test (&body body)
  `(let ((i1 (make-instance 'c1))
         (i2 (make-instance 'c2))
         (i3 (make-instance 'c3)))
     (dotimes (i *outer-times*)
       (time (dotimes (j *inner-times*) ,@body)))))

(defun fun-test       () (test (fun i1)))
(defun accessor1-test () (test (accessor1 i1)))
```

```
(defun accessor2-test () (test (accessor2 i2)
                               (accessor2 i2)))
(defun accessor3-test () (test (accessor3 i1)
                               (accessor3 i2)
                               (accessor3 i3)))
(defun g1-test        () (test (g1 i1)))
(defun g2-test        () (test (g2 i2)
                               (g2 i2)))
```