Fast generic dispatch for Common Lisp

Robert Strandh
University of Bordeaux
351, Cours de la Libération
Talence, France
robert.strandh@u-bordeaux1.fr

ABSTRACT

We describe a technique for generic dispatch that is adapted to modern computers where accessing memory is potentially quite expensive. Instead of the traditional hashing scheme used by PCL [6], we assign a *unique number* to each class and the dispatch consists of comparisons of the number assigned to an instance to a certain number of (usually small) constant integers. While our implementation (SICL) is not yet in a state where we are able to get exact performance figures, a conservative simulation suggests that our technique is significantly faster than the one used in SBCL, which uses PCL, and indeed than the technique used by most high-performance Common Lisp implementations. Furthermore, existing work [7] using a similar technique in the context of static languages suggests that perfomance can improve significantly compared to table-based techniques.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Code generation, Optimization, Run-time environments*

1. INTRODUCTION

Generic dispatch is an extremely important part of any Common Lisp implementation, because it constitutes the essence of the invocation of all generic functions, including accessors. It is therefore of utmost importance that generic function dispatch be as efficient as possible.

The efficiency of the generic dispatch technique may have consequences on the programming style, as programmers may be dissuaded from using generic functions on standard instances for reasons of performance, and instead opt in favor of ordinary functions on other types of data structures such as structure instances, arrays, or lists, even though standard instances have better behavior in the context of interactive and incremental development.

Conversely, with a high-performance generic dispatch technique, better data structures can be used, including in the implementation itself, with less special-purpose code and therefore improved maintainability as a result.

While it may seem like techniques for efficient generic dispatch exist, and indeed are part of most high-performance Common Lisp implementations, many of those techniques and implementations date back a few decades, and the parameters that determine efficiency have changed radically as a result of the increasing gap between processor speed and memory-access time.

2. PREVIOUS WORK

Most work on generic dispatch has been done in the context of more mainstream programming languages such as C++ or Java. There are three aspects of Common Lisp that complicate the situation with respect to such languages:

1. Multiple inheritance.
2. Multiple dispatch.
3. Interactivity.

Multiple inheritance[1] makes it possible for a slot to have a different *position* in the slot vector in instances of different classes. Slot accessors must take this possibility into account.

Multiple dispatch makes it more difficult to use table-based techniques with entries for each class, because the size of the table grows exponentially in the number of specialized parameters. Table-compression techniques help overcome some of these problems, at the cost of a more complicated, and thus more expensive, dispatch algorithm.

Many existing techniques are based on the complete program being available in order to compute dispatch tables. The interactive nature of Common Lisp makes it more difficult to use such techniques, because every table may have to be recomputed whenever a small modification is made to a class. Though, some techniques may amortize this cost by *invalidating* some tables and recomputing them when required.

The fact that Common Lisp is interactive also makes it possible for the existence of *obsolete instances*, i.e., instances

[1] C++ also has multiple inheritance of course.

where the slots no longer correspond to the definition of the class, so they have to be *updated* before they are inspected.

2.1 PCL

In PCL[2] [6] a standard object is represented as a two-word header object where the first words is a pointer to a *class wrapper* and the second word is a pointer to the *slot vector* of the instance. The class wrapper is also a two-word structure that contains a *hash seed* and a pointer to the class object.

Each generic function contains a *memoization table*. Each entry of the table contains a class wrapper and the entry point for the effective method to be called for instances of the corresponding class. The memoization table uses a simple hashing mechanism, so that the hash seed of the class wrapper of the argument is reduced modulo the size of the memoization table in order to find the corresponding entry. The class wrapper in the entry is compared using `eq` to the class wrapper of the argument, and if they are the same, the corresponding effective method is called. When there is no hit, it could be that there is a hash collision, or it could be that no entry exists in the table for the class of the argument. Thus, if there is no initial hit, the table is searched sequentially until an entry is found or all the entries have been examined.

In the best case then, the following operations are required for a simple slot reader generic function:

1. Access the class wrapper of the argument; a memory access.
2. Access the hash seed of the class wrapper; a memory access.
3. Access the size of the memoization table; a memory access.
4. Reduce the hash seed of the class wrapper modulo the size of the memoization table; a simple masking operation if the size of the table is a power of 2.
5. Access the memoization table of the generic function; a memory access.
6. Access the class wrapper in the memoization table entry; a memory access.
7. Compare the class wrapper in the memoization table entry to the class wrapper of the argument; a simple register comparison.
8. Access the entry point of the effective method in the memoization table entry; a memory access.
9. Jump to the entry point of the effective method; an unconditional jump.
10. The effective method accesses the slot vector of the instance; a memory access.
11. The slot containing the desired object is read and returned; a memory access.

[2]PCL stands for Portable Common Loops.

As we can see, there are 8 memory accesses involved.

The authors also mention an optimization for slot readers and slot writers in the case where such a generic function is called with only a few different classes. In that case, they suggest replacing the table lookup with a simple test for the different cases. However, since class wrappers are heap-allocated objects, a copying garbage collector may move them around. For that reason, class wrappers can not be inline constants in the code, and must be stored in the generic function. If such an optimization is implemented, the mechanism is reduced to the following steps:

1. Access the class wrapper of the argument; a memory access.
2. Access one or more class wrappers stored in the generic function; at least one memory access.
3. Compare the class wrapper of the argument to the class wrapper(s) stored in the generic function; fast register operation.
4. Access the slot vector of the instance; a memory access.
5. The slot containing the desired object is read and returned; a memory access.

The minimum number of memory access is reduced to 4 if the generic function is called with instances of a single class.

The technique used by PCL automatically catches obsolete instances. When a class is modified, the current wrapper is eliminated from all existing memoization tables, forcing the lookup to fail, and thus triggering the update of the obsolete instance.

While not mentioned in the published work, PCL also allows for a discriminating function that tests argument types using `typep`. While in general quite expensive, `typep` can be efficient when a built-in type known at compile time is tested for. If the number of cases is small and the `typep` test can be determined to be inexpensive, then this kind of discriminating function could potentially achieve performance similar to our technique.

2.2 Work by Zendra et al

Perhaps the work that is most similar to ours is that of Zendra et al [7]. Like the present work, they are interested in performance improvements to dispatch on modern architectures by eliminating table lookups. Unlike the present work, the context of their work is a static language (Eiffel) which simplifies many aspect of the optimization. For one thing, they use global type inference to optimize away dispatch entirely when not needed, and they can inline the dispatch mechanism at each call site because there can be no changes to the class hierarchy or to the applicable methods at runtime. Like the present work, they also use inline code performing a binary search in order to determine which applicable method to invoke.

2.3 Other work

Dreisen et al [2] give an overview of different dispatch techniques in the context of modern processor architectures. They are specifically interested in the influence of *processor pipelining* and *superscalar execution* on the performance of various dispatch techniques. In addition, they take into account the influence of dynamic typing on the dispatch cost. In particular, they mention the fact that, with multiple inheritance, the location of a slot may vary according to the exact runtime type of an object. Their study is limited to the case of single dispatch. Nevertheless, they treat both static techniques and dynamic techniques, including call-site caching. Most of the techniques used are based on some kind of table lookup, including the dynamic techniques containing a table (which can be small) used as a cache. They conclude that table-based methods are expensive on modern hardware, and that inline caching is likely to perform the best on modern processors.

The work of Zibin and Gil [8] also discusses table-based techniques. Their paper addresses multiple inheritance and multiple dispatch, but concentrates on the efficiency of the algorithm for building the dispatch table and the size of the resulting dispatch table.

In their 2012 paper, Hariskrishnan and Kumar [4] also sacrifice efficiency of the dispatch algorithm in favor of space efficiency by removing the lookup table altogether. They propose the alternative name *constant-time* technique for *table-based* technique, and *non-constant-time* for techniques that do not use table lookup. Their technique is applicable only to *single inheritance* systems.

The technical report of Bachrach and Burke [1] is concerned with the language Dylan which is similar to Common Lisp in many respects, although in terms of generic dispatch, it allows for specializers other than classes and singletons, and features such as *sealing* of classes and methods allow for further optimizations of generic dispatch. Their approach is similar to ours, in that they construct a *decision tree* for each generic function. It is different from ours in that the decision tree is not implemented as inline code, but instead as a data structure consisting of *engine nodes*, requiring the dispatch code to make several memory accesses. On the other hand, their technique is more flexible in that it allows for each call site to exploit the decision tree and optimize according to locally available information, sometimes resulting in particular call sites not requiring any dispatch code at all. Unfortunately, this technical report is in an unfinished state, making it hard to determine the work and the results behind it.

Dujardin et al [3] give a fast algorithm for creating compressed tables for multiple dispatch. While the dispatch is still constant time after compression, as with other table-compression techniques, theirs adds overhead to the dispatch itself. The unique number of the type of each argument must be used to index a per-argument table in order to obtain indices in the compressed table. And of course the element in the compressed table must then be accessed (requiring index arithmetic and a memory access) before the relevant method can be invoked. Merging the per-argument tables using standard table-compression techniques add yet more comparisons and memory accesses.

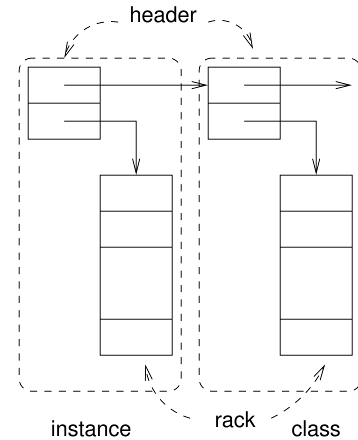

Figure 1: Representation of a general instance.

Like Bachrach and Burke, Hölze and Ungar [5] optimize dispatch by specializing the dispatch algorithm for each call site. They use run-time type information as feedback to the compiler which can then often create more efficient dispatch code simply because for a given call site it is common that only a small subset of all possible argument types are actually used. We have not addressed this possibility, because it would require a way to recompile the caller, or at least a single call site whenever a change to the callee or to the class hierarchy is made.

3. OUR TECHNIQUE

A SICL object is either an *immediate object* or a *heap object*. A heap object is either a `cons` cell or a *general instance*.[3] General instances and cons cells have distinct unique *tags*. Every general instance is represented by its *header*. The header contains two pointers, one to the *class* of the instance and one to the *rack* of the instance. The pointer to the class is a tagged pointer to another general instance. The pointer to the rack is a raw machine pointer. This representation is shown in Figure 1.

Each class is assigned a *unique number* starting at 0. The number is assigned when the class is finalized, and a new number is assigned whenever a class is finalized as a result of changes to the class or any of its superclasses. Currently, class numbers are never reused. This way of allocating class numbers is advantageous because it often results in a subtree of classes occupying a *dense interval* of class numbers, the importance of which is discussed below. On a 64-bit architecture with 63-bit or 62-bit fixnums, the unique number will always be a fixnum. On 32-bit architectures this is very likely also true, and it is certainly true if class numbers are reused.[4] If class numbers are not reused, on a 32-bit plat-

[3]Since heap allocated objects are either `cons` cells or general instances, it follows that general instances are used to represent not only *standard objects*, but also structure instances, instances of built-in classes such as `symbol` and `package`, arrays, bignums, complex numbers, etc.

[4]The garbage collector would have to take care of recycling

form one might have to use two words in order to be on the safe side.

The first element of the rack of every general instance is called the *stamp*. The stamp is the unique number of the class as it was when the instance was created or updated as a result of changes to its class. An instance is *obsolete* if and only if its stamp is not the same as the unique number of its class.

Together, the unique number of the class and the stamp of the instance play a role similar to that of the *class wrapper* of PCL. The unique number of the class corresponds to the most recent wrapper for the class and the stamp corresponds to a wrapper that may have become obsolete as a result of updates to the class.

Our dispatch technique works by comparing the stamp of each specialized argument to a set of non-negative constant integers in a way similar to a binary search. The result of these comparisons identifies the argument as being an instance of a particular class, or of one of a set of classes for which the same effective method is valid. As an example, let us take a generic function that specializes on a single argument, and that has already been called with classes numbered 1, 4, 5, 6, 8, and 10, where classes numbered 4, 5, and 6 result in the same effective method being invoked. The discriminating function of this generic function would then look like this:

```
(tagbody
    (if (< stamp 7)
        (if (< stamp 4)
            (if (= stamp 1)
                (go m1)
                (go miss))
            (go m2))
        (if (= stamp 8)
            (go m3)
            (if (= stamp 10)
                (go m4)
                (go miss))))
  m1
    ;; invoke method for class numbered 1
     ...
    (go out)
  m2
    ;; invoke method for classes numbered 4, 5, 6
     ...
    (go out)
  m3
    ;; invoke method for class numbered 8
     ...
    (go out)
  m4
    ;; invoke method for class numbered 10
     ...
    (go out)
  miss
    ;; handle miss
    ...
  out)
```

class numbers.

This discriminating function is generated from the *call history* of the generic function. The call history is a simple list of entries, where each entry contains a list of classes for each specialized parameter and a corresponding effective method to call.

As can be seen from the example above, in this case, a maximum of three tests will be performed for six different classes. The number of tests required is logarithmic in the number of entries of the call history.

The advantage of this technique is that on modern processors, comparing integers is very fast, whereas table lookups require memory accesses which are significantly more costly in general.

In terms of different types of operations, our technique requires the following steps for a simple slot reader:

1. Access the rack of the argument; a memory access.
2. Access the stamp of the rack; a memory access.
3. Compare the stamps to a set of constants; fast register operations.
4. The slot containing the desired object is read and returned; a memory access.

As we can see, only 3 memory accesses are required. The number of comparisons is not constant, of course, but it is very small for the vast majority of generic functions, and quite reasonable (compared to the cost of a memory access) even when a very large number cases need to be handled.

Like the technique used by PCL, our technique automatically detects obsolete instances. When a class is updated, every generic function that dispatches on this class[5] is determined, and the *call history* of each such generic function is searched for entries using the class. These entries are removed and then the discriminating function is either recomputed from the call history, or is set to a function that, when invoked, recomputes the discriminating function from the call history. The result is that, if an obsolete instance is used in dispatch, its stamp will not be valid for dispatch, so the discriminating function will fail to recognize the instance, forcing an update of the instance, and a new dispatch attempt.

In contrast to the technique used by PCL, our technique makes it more costly to update classes. In PCL, invalidating a class wrapper is a simple matter of setting the hash seed to 0. Our technique requires that the call history be updated and the discriminating function to be recomputed for every generic function with a method that specializes on the class being updated. While this cost is unbounded, it is acceptable in practice.

[5]The function `specializer-direct-generic-functions` returns a list of generic functions that have a method using the class as a specializer.

While not directly related to dispatch performance, an important consequence of the split between the header and the rack for every general instance is that generic functions can be updated without requiring locking, by using a simple version of software transactional memory. When a generic function needs to be updated, for instance as a result of one of the classes it specializes on being updated, the rack and the call history can be copied, and then modifications can be made to the copy, and finally, the copied rack can replace the original rack by the use of a `compare-and-swap` instruction. Should the instruction fail, the attempted update is restarted.

4. PERFORMANCE OF OUR TECHNIQUE

Unfortunately, our system (called SICL) is not yet in sufficiently finalized to allow us to make any tests of performance. However, we constructed a few simulations that give us some indications of the performance of our technique compared to the technique used by PCL.

In our first test, we decided to measure the time it takes for the generic dispatch of a simple slot reader.

First, we created a class with a single slot and a reader for that slot like this:

```
(defclass c () ((%x :initarg :x :reader x)))
```

Next, we defined an instance of this class:

```
(defparameter *i* (make-instance 'c :x 1))
```

Then, we created a function containing a loop where the slot reader is called in each iteration:

```
(defun f ()
  (declare (optimize (safety 0) (speed 3) (debug 0)))
  (loop with i = *i*
        repeat 10000
        do (loop repeat 100000
                 do (x i))))}
```

In order to minimize overhead due to looping, we set the `optimize` flags as shown.

The loop calls the reader 10^9 times. Since 10^9 may not be a fixnum on some 32-bit platforms, we use a nested loop. The table below shows the result of this test on a small selection of platforms.

Impl	OS	Proc	Clock	Time	Cyc
SBCL 1.1.13	MacOS	x86-64	1.8GHz	11	20
SBCL 1.1.16	MacOS	x86-64	3.3GHz	4.7	16
SBCL 1.1.18	Linux	x86-64	1.6GHz	12	19
CMU 20d	Linux	x86-64	3GHz	158	474
Allegro 9.0	MacOS	x86-64	2.2GHz	10	22
LispWorks 6.1.1	Windows	x86-64	3GHz	11	33
Clozure 1.9	MacOS	x86-64	3.3GHz	21	69
ABCL 1.2.1	MacOS	x86-64	1.8GHz	183	329
ABCL 1.0.1	Linux	x86-64	3GHz	152	456

The time includes not only calling the slot reader, but also the loop iteration overhead. Furthermore, calling the slot reader involves not only the generic dispatch, but also checking the argument count and some other function-call overhead. For now, we ignore all this overhead.

In the table, we have included only implementations reputed to be "high-performance", though some interpreted implementations such as CLISP and ECL compare favorably to the slower implementations that we tested. An important aspect that is not included in the table is whether the implementation is thread safe or not. Thread safety may have a negative impact on performance, so implementations that are not thread safe may look better in comparison. Unfortunately, we do not know which implementations are thread safe among the ones in the table, except that we know that LispWorks 6.1.1 *is* thread safe.

Note that the test above was designed to be as advantageous as possible to table-based techniques in the following ways:

- The only instance involved will rapidly reside in the cache, so the additional cost of memory references is not measured.
- There is a single class involved, allowing the implementation to select a better strategy for the discriminating function, as the paper on PCL suggests.

To get some indication of the performance of our technique, we need to simulate the layout of a SICL general instance. We do that by defining the header as a Common Lisp `struct` and by using a *simple vector* for the rack. The definition of the header looks like this:

```
(defstruct s class rack)
```

We create a simulated general instance as follows:

```
(defparameter *j*
  (let ((rack (make-array 2 :initial-contents '(10 1))))
    (make-s :class nil :rack rack)))
```

Our simulated slot reader is defined like this:

```
(defun y (instance)
  (declare (optimize (safety 0) (speed 3) (debug 0)))
  (let* ((rack (s-rack instance))
         (stamp (svref rack 0)))
    (declare (type fixnum stamp))
    (if (= stamp 10)
        (svref rack 1)
        (error "1"))))

(proclaim '(notinline y))
```

Finally, we define a function containing a loop that calls our simulated slot reader:

```
(defun g ()
  (declare (optimize (safety 0) (speed 3) (debug 0)))
  (loop with j = *j*
        repeat 1000000000
        do (y j)))
```

On our computer (x86-64 running at 1.6GHz) executing this function takes less than 3 seconds, which represents a significant improvement. Comparing to the table above, 3 seconds represents 5 clock cycles.

The comparison above is somewhat biased in our favor, because we can not be sure that a single test will suffice in order to determine the correct method to invoke. For that reason, we devised the following test:

```
(defun yy (instance)
  (declare (optimize (safety 0) (speed 3) (debug 0)))
  (let* ((rack (s-rack instance))
         (stamp (svref rack 0)))
    (declare (type fixnum stamp))
    (cond ((> stamp 1280) (error "1"))
          ((> stamp 640)  (error "2"))
          ((> stamp 320)  (error "3"))
          ((> stamp 160)  (error "4"))
          ((> stamp 80)   (error "5"))
          ((> stamp 40)   (error "6"))
          ((> stamp 20)   (error "7"))
          ((> stamp 10)   (error "8"))
          (t (svref rack 1)))))

(proclaim '(notinline yy))

(defun gg ()
  (declare (optimize (safety 0) (speed 3) (debug 0)))
  (loop with j = *j*
        repeat 1000000000
        do (yy j)))
        maximize (yy (car l))))
```

This test simulates a situation where the generic dispatch needs 8 tests to determine what method to call, which can be thought of as a generic function with 256 methods, or alternatively a single method but where the call history contains 256 classes with sparse unique numbers.

On our computer executing this function takes less than 4.5 seconds, which seems to suggest that the number of comparisons has only a modest impact on the performance. In fact, the numbers suggest that less that one additional clock cycle per additional comparison is required.[6]

The following test is meant to show the impact of a very large number of classes, so that constants are no longer very small:

```
(defun yyy (instance)
  (declare (optimize (safety 0) (speed 3) (debug 0)))
  (let* ((rack (s-rack instance))
         (stamp (svref rack 0)))
    (declare (type fixnum stamp))
    (cond ((> stamp 12800000) (error "1"))
          ((> stamp 6400000)  (error "2"))
          ((> stamp 3200000)  (error "3"))
          ((> stamp 1600000)  (error "4"))
          ((> stamp 800000)   (error "5"))
          ((> stamp 400000)   (error "6"))
          ((> stamp 200000)   (error "7"))
          ((> stamp 100000)   (error "8"))
          (t (svref rack 1)))))

(proclaim '(notinline yyy))

(defun ggg ()
  (declare (optimize (safety 0) (speed 3) (debug 0)))
  (loop with j = *j*
        repeat 1000000000
        do (yyy j)))
```

On our computer executing this function takes exactly the same time to execute as the previous one, which seems to suggest that the size of the constants has very little impact on performance, at least on our platform.

The tests include loop overhead that should be subtracted from the timing results, but this overhead is probably very small, so we can ignore it. Furthermore, disassembling the simulated slot readers show that the code is very close to what we expect the SICL compiler to emit, so there is very little overhead there as well.

Notice, however, that the results all include the overhead of a function call. This function call can not be avoided, which suggests that our results reflect the real observable improvement. However, in order to appreciate the performance improvement of dispatch mechanism itself, the overhead of the function call should not be included, which suggests that the net improvement is significantly higher than a factor 3.

5. CONCLUSIONS AND FUTURE WORK

We have presented a fast technique for generic dispatch in Common Lisp. Clearly, our tests do not represent any scientifically convincing argument that our technique is faster than existing techniques. Rather, the presentation of the technique itself should be considered essence of the paper, and the performance simulations should only be viewed as indications that our technique is worth pursuing as the bases of the generic dispatch mechanism in SICL.

Having said that, we can still speculate about the impact of our technique, should the results be confirmed in a more realistic setting.

With our technique, the amount of work to be done in a simple slot reader or slot writer is no greater than the work needed for a non-generic version, such as `symbol-name` or `package-nicknames` in a typical Common Lisp implementation. Our technique therefore makes it feasible to make such readers and writers generic, and this is exactly what we do in SICL. We use the same CLOS mechanisms (i.e., class initialization, class finalization, etc.) for built-in classes as those used for standard classes, even though built-in classes can not be redefined. By using the same mechanism, we remove a number of special cases and we are able to simplify the overall structure of the system.

It is even possible to go one step further. The amount of work required in our dispatch mechanism is no greater than the work needed in a binary addition function that tests for the exact type of its arguments. Hence it is entirely feasible to make such a function generic, allowing the user to add methods for other kinds of objects such as polynomials or other mathematical objects. For reasonable performance, it would still be required to capture special cases such as fixnum or floating-point addition and inline them, but the default function could very well be an ordinary generic function without any significant loss of performance.

Although it is easy to switch to a different mechanism when the number of methods turns out to be very large, or more generally, when the call history contains a large number of elements, preliminary tests indicate that it might not be necessary to do so. The canonical example of a generic function where a different mechanism might be considered is that of `print-object`. However, our preliminary tests show that the additional cost of a comparison is very small (a single processor cycle) so that even if the call history is very large, the cost of these comparisons is modest.

[6]In this case, however, the *branch prediction* circuits of the processor always guess the right thing

For a scientifically significant comparison of our technique to existing techniques, we first need to get SICL into a sufficiently finished state that more elaborate tests can be designed. In particular, we then need to test situations where cache misses might significantly impact performance. Furthermore, we need to gather information about techniques used in existing high-performance implementations, which may not be easy since several of the existing high-performance implementations are commercial and closed source.

6. ACKNOWLEDGMENTS

We would like to thank Pascal Bourguignon, Stas Boukarev, Dave Fox, and Hans Hübner for their help with the tests on platforms not at our disposal. We would also like to thank Christophe Rhodes for reading an early draft of the paper and for suggesting improvements, and to the developers at LispWorks for pointing out the potential influence of thread-safety on performance.

7. REFERENCES

[1] J. Bachrach and G. Burke. Partial dispatch: Optimizing dynamically-dispatched multimethod calls with compile-time types and runtime feedback. Technical report, 2000.

[2] K. Driesen, U. Hölzle, and J. Vitek. Message dispatch on pipelined processors. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 253–282, London, UK, UK, 1995. Springer-Verlag.

[3] E. Dujardin, E. Amiel, and E. Simon. Fast algorithms for compressed multimethod dispatch table generation. *ACM Trans. Program. Lang. Syst.*, 20(1):116–165, Jan. 1998.

[4] S. Harikrishnan and R. Kumar. Space efficient non-constant time multi-method dispatch in object oriented systems. *SIGSOFT Softw. Eng. Notes*, 37(2):1–6, Apr. 2012.

[5] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 326–336, New York, NY, USA, 1994. ACM.

[6] G. Kiczales and L. Rodriguez. Efficient method dispatch in pcl. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 99–105, New York, NY, USA, 1990. ACM.

[7] O. Zendra, D. Colnet, and S. Collin. Efficient dynamic dispatch without virtual function tables: The smalleiffel compiler. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, pages 125–141, New York, NY, USA, 1997. ACM.

[8] Y. Zibin and J. Y. Gil. Fast algorithm for creating space efficient dispatching tables with application to multi-dispatching. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 142–160, New York, NY, USA, 2002. ACM.