Dartmouth College

Dartmouth Digital Commons

Dartmouth College Undergraduate Theses

Theses and Dissertations

6-1-2001

Efficient Compression of Generic Function Dispatch Tables

Eric Kidd
Dartmouth College

Recommended Citation

Efficient Compression of Generic Function Dispatch Tables

Eric Kidd

Dartmouth College

Hanover, NH, USA

`eric.kidd@pobox.com`

June 1, 2001

Abstract

A generic function is similar to an overloaded operator, but provides a way to select an appropriate behavior at run-time instead of compile-time. Dujardin and colleagues have proposed an algorithm for building and compressing generic function dispatch tables.

We present several modifications to their algorithm, including an improvement to PSEUDO-CLOSEST-POLES and two new algorithms for compressing pole tables. The two new compression algorithms are simple and fast, and one produces smaller output than the original.

1 Introduction

Traditional object-oriented languages provide **member functions**. Languages such as CLOS, Cecil and Dylan [Sha96], however, provide **generic functions**. Generic functions are more flexible than member functions, but their additional flexibility comes at a higher cost. Reducing this cost makes generic functions more practical.

Both member functions and generic functions are **polymorphic**—they provide different behaviors for different types of objects, and they choose among these behaviors at run-time. But member functions and generic functions differ in several key ways. A member function belongs to a particular class, and only the descendants of that class may define new behaviors. To create a new member function, a programmer must modify the appropriate class definitions.[1] And a member function provides only **single dispatch**—the ability to select an appropriate behavior based on the type of a single object. Generic functions, on

---

[1] If the original class definitions belong to a different organization, the programmer may be out of luck.

the other hand, do not belong to a particular class. A programmer may define new generic functions without modifying existing classes.[2] And generic functions provide **multiple dispatch**—the ability to select an appropriate behavior based on the types of more than one object.

Generic functions first appeared in CommonLoops [BKK+86]. In the fifteen years since then, relatively few languages have chosen to provide generic functions. Personal experience suggests that the slow adoption of generic functions stems from implementation difficulty and performance concerns.[3] Therefore, if we want more languages to provide generic functions, we must find simple, efficient implementation techniques.

1.1   A Simple Example: Rock, Paper, Scissors

Consider the children's game of "rock, paper, scissors." In this game, two players each secretly select one of three objects. They then reveal their choices simultaneously. According to the traditional rules of this game, each object defeats exactly one of the others: "paper covers rock, rock crushes scissors, and scissors cut paper."

We can model this game using a small class hierarchy and a single generic function. The pseudo-code in Figure 1 defines an abstract base class `Thing`, three classes representing the objects in the game, and a generic function with four methods. Methods 2–4 represent the three victory conditions, and Method 1 provides a default behavior.

Note that `defeats` shows the typical characteristics of a generic function. It stands alone, independent of the class definitions. It provides four behaviors, each appropriate to different object types. And it uses multiple dispatch— there's no way to select the appropriate behavior without looking at the types of both arguments.

1.2   Generic Function Terminology

In Figure 1, `defeats` is a **generic function** with four **methods**. The variables `t1` and `t2` are the **formal parameters** of each method. The values of the variables `paper` and `rock` are the **actual parameters** of the call `defeats(paper, rock)`. The process of choosing the correct method for a given list of actual parameters is referred to as **multiple dispatch**.

A method **specializes** on the types of its formal parameters. For example, method 2 specializes on `Paper` and `Rock`.

In the above example, methods 1 and 2 are **applicable** to the call `defeats(paper, rock)`. A method is applicable to a given list of actual parameters if and only if

---

[2]Generic functions do not automatically gain access to private members of a class, so this ability raises fewer encapsulation issues than the reader might surmise.

[3]This is far from a consensus opinion. Jonathan Bachrach, upon reviewing this paper, pointed out that several implementers have chosen to omit generic functions from new languages because of concerns about namespace pollution and programmer-level semantics. These same implementers, however, raised no objections about the current techniques for implementing generic functions.

```
// Class hiearchy.
abstract class Thing
class Paper (Thing)
class Rock (Thing)
class Scissors (Thing)

// Default behavior.
method defeats (Thing t1, Thing t2): return false    // 1

// Victory conditions.
method defeats (Paper t1, Rock t2): return true      // 2
method defeats (Rock t1, Scissors t2): return true   // 3
method defeats (Scissors t1, Paper t2): return true // 4

// Sample invocation.
paper = new Paper
rock = new Rock
defeats(paper, rock) // Returns true.
```

Figure 1: A model of the game "rock, paper, scissors."

every actual parameter is an instance of the type specified for the corresponding formal parameter.

We define a partial ordering called **specificity** on the applicable methods of a generic function. In the example above, method 2 is **more specific** than method 1. Method 3, however, is neither more nor less specific than method 2, because the two methods are applicable to different argument types. The exact rules for method specificity vary from language to language.[4]

Thus, we can state our problem as follows:

> Given a generic function and a list of actual parameters, choose the most-specific applicable method.

1.3   Overview of Method Lookup Using Tables

Ideally, given a a generic function with N arguments and M methods, we should be able to choose an appropriate method in $O(N)$ operations. In general, programmers expect function call overhead to be small, and most programmers would be surprised (and appalled) to discover that a Dylan function call can easily take $O(NM \log M)$ operations [Sha96, pp. 95–96].

Dujardin and colleagues propose a hypothetical implementation of $O(N)$ dispatch [DAS98]. In this imaginary scenario, we would assign sequential ID

---

[4]In particular, it may be possible for two methods to have an **ambiguous** ordering, or for method specificity to depend on the types of the actual parameters. Dujardin and colleagues [DAS98] and Shalit [Sha96, p. 96] provide more details.

```
func dispatch_m (arg1, arg2):
  index1 = class_id(arg1)
  index2 = class_id(arg2)
  method = m_methods[m_dispatch_matrix[index1][index2]]
  return method(arg1, arg2)
```

Figure 2: Naive dispatch code for a generic function m, based on hypothetical suggestion by Dujardin and colleagues [DAS98].
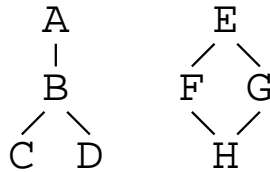
```
    A           E
    |          / \
    B         F   G
   / \         \ /
  C   D         H
```

Figure 3: Inheritance relationship for classes A through H. Class B, for example, is a subclass of class A.

numbers to every class in our program, and use these IDs as indexes into a giant, N-dimensional matrix. We could then dispatch a two-argument generic function m using the code in Figure 2.

Unfortunately, these matrices would require enormous amounts of space. For a single generic function with N arguments, defined in a program with C classes, we would need a matrix with C^N elements, a number that could easily range into the terrabytes [DAS98, p. 155].

Such a dispatch matrix, though, would contain many duplicate entries. Dujardin and colleagues propose an algorithm to eliminate these duplicates and shrink a dispatch matrix down to a reasonable size [DAS98].

Consider the classes and methods in Figures 3 and 4.[5] Since the generic function m has 2 arguments, and our program has 8 classes, the generic function m could be dispatched using a 8×8 matrix. But we can reduce this substantially. Notice that, in the first argument position, no method ever specializes on class C or class D. In fact, these classes inherit all their behaviors from class B. If we can somehow combine these classes into a group, we can shorten one dimension of our dispatch matrix.

In Figure 5, we mark off the specializers on the first argument position, and divide the classes into three groups. Group 0 contains classes which may not be used in the first argument position. Group 1 contains class A. Group 2 contains classes B, C and D. We call classes A and B the **primary 1-poles** of function m, where "1" is the argument position. We say that a pole **influences** all the classes in its group. For example, class B influences classes B, C and D.

---

[5]This example closely follows several examples from the original paper by Dujardin and colleagues [DAS98].

$m_1(A, E)$
$m_2(B, E)$
$m_3(A, F)$
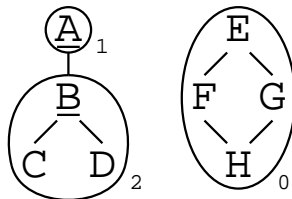$m_3(B, G)$

Figure 4: Methods on the generic function m.

Figure 5: 1-poles for generic function m.

In Figure 6, we mark off the specializers for the second argument position. We discover three **primary 2-poles** (E, F, and G). But we also discover a strange problem: Class H inherits from two different poles. Depending on the type of the first argument, either method m_3 or method m_4 might have the most specific specializers. Since the behavior of class H may differ from the behavior of class F or G, we need to make H a **secondary 2-pole**.

We can now build two **pole tables** (Figure 7) mapping classes to the corresponding pole ID.[6] We use the pole IDs as indices into a 3×5 dispatch matrix (Figure 8) containing the most-specific-applicable methods for each combination. The dispatch matrix also contains two special values: α represents an ambiguous-method error, and ϵ represents a no-applicable-methods error.

The dispatch matrix is built in a language-specific fashion.[7]

If we count the entries in these tables, we find that we have saved space. The new tables contain $2*8+3*5 = 31$ entries, while the old tables contained $8*8 = 64$ entries. And in larger programs, the savings typically will be much greater—

---

[6]Dujardin and colleagues referred to pole tables as **argument arrays**. We use slightly different terminology in this paper.

[7]Dujardin and colleagues provide more details [DAS98]. Note that no appropriate algorithms exist to build dispatch matrices for Dylan, as explained in that paper.
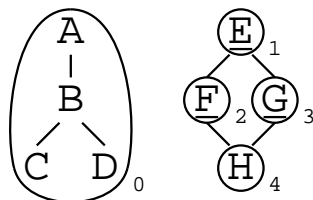

Figure 6: 2-poles for generic function m.

Argument	A	B	C	D	E	F	G	H
1	1	2	2	2	0	0	0	0
2	0	0	0	0	1	2	3	4

Figure 7: Uncompressed pole tables for generic function m.

	2-pole				
1-pole	0	1	2	3	4
0	ϵ	ϵ	ϵ	ϵ	ϵ
1	ϵ	m_1	m_3	m_1	m_3
2	ϵ	m_2	α	m_4	α

Figure 8: Dispatch matrix for generic function m.

we will often be able shrink multi-megabyte tables down to a few hundred bytes.

The dispatch code for m now resembles Figure 9.

1.4 Overview of Compression Techniques

As seen in Table 1, this technique produces very small dispatch matrices. The pole tables, however, are still uncomfortably large. Therefore, we need to compress the pole tables themselves using a second technique.

Dujardin and colleagues compress the pole tables using an argument-coloring algorithm [DAS98, pp. 150–152]. If two pole tables contain no conflicting entries, they can be merged as shown in Figure 10. But before the dispatch code looks up an entry, it must verify that all the argument types are correct—otherwise, it will use an entry intended for another pole table, with disastrous consequences. In a statically-typed language, the compiler may verify argument types at compile-time. But in a dynamically-typed language, the dispatch code must perform this verification at run-time.

Although space-efficient, argument-coloring compression has several drawbacks. It requires us to spend a lot of time building conflict lists and assigning colors. It can only operate on a global basis—we can't use argument-coloring

```
func dispatch_m (arg1, arg2):
  index1 = m_pole_table_1[class_id(arg1)]
  index2 = m_pole_table_2[class_id(arg2)]
  method = m_methods[m_dispatch_matrix[index1][index2]]
  return method(arg1, arg2)
```

Figure 9: Improved dispatch code for generic function m, based on an algorithm by Dujardin and colleagues [DAS98].

	Hello-1	Hello-2	ICFP	d2c
Pole Tables	16,100	36,960	46,436	320,574
Dispatch Matrices	1,176	1,527	1,731	3,397

Table 1: Relative number of entries in pole tables and dispatch matrices.

	A	B	C	D	E	F	G	H
Pole	1	2	2	2	1	2	3	4
Color (not stored)	1	1	1	1	2	2	2	2

Figure 10: Pole tables compressed using coloring algorithm by Dujardin and colleagues [DAS98].

to compress a single pole table at a time. And in a dynamic language, it forces us to perform extra type checks.

We could avoid much of the compile-time overhead using a simpler approach.[8] As seen in Figure 7, there tend to be lots of boring entries at the beginning or end of each row. We can trim these long runs of 0's from the tables, leaving only the interesting entries shown in Figure 11. If related classes tend to clump together, this approach will provide us with excellent compression and very respectable dispatch speeds.

We also propose a third form of compression, based on fixed-size partitioning.[9] In this approach, we break the pole tables into fixed-size chunks, and re-used duplicate chunks as shown in Figure 12. This technique allows us to deal with run-time class creation gracefully, because we can simply add new partitions to the end of the pole table.

2   Pole Table Generation

Dujardin and colleagues generate the uncompressed pole tables using the POLE-COMPUTATION algorithm [DAS98, p. 139]. Using this algorithm, the compiler

---

[8]This compression technique is based on a conversation between the author and Thomas Cormen. Both of us currently deny authorship.

[9]Vitek and Horspool describe several variable-sized partitioning techniques for single-dispatch languages in [VH96].

			Index			
Argument	Offset	Size	0	1	2	3
1	A	4	1	2	2	2
2	E	4	1	2	3	4

Figure 11: Pole tables compressed using offset table fragments.

Argument	A-D	E-H
1	ptr_1	ptr_0
2	ptr_0	ptr_2

Pointer	0	1	2	3
ptr_0	0	0	0	0
ptr_1	1	2	2	2
ptr_2	1	2	3	4

Figure 12: Pole tables compressed using partitions.

can construct a pole table for argument position n of generic function m as follows:

1. Compute the set of specializer classes for this argument position. This is the set of class types of the nth formal parameters of each method on m.

2. Allocate an empty pole table, and mark each specializer class.

3. If `Object` is specializer class, then mark it as a primary pole. Otherwise, mark it as an error pole.

4. For each remaining entry in the pole table, check to see if the entry has been marked as a specializer class.

   (a) If the entry has been marked as a specializer class, then mark it as a primary pole.

   (b) If the entry has not been marked as specializer class, then calculate the PSEUDO-CLOSEST-POLES of the corresponding class. If there is a single "close" pole, then copy down the appropriate pole information to the current entry. If there is more than one such pole, then mark the entry as a new secondary pole.

Note that POLE-COMPUTATION makes a single pass over the pole table. For this to work, classes must appear in inheritance order—no class may appear before its superclasses [DAS98, p. 133].

Figure 13 presents a slightly modified version of the POLE-COMPUTATION algorithm. The new version deviates from the original in one place, replacing the original call to PSEUDO-CLOSEST-POLES [DAS98, p. 136] with a call to a slightly-modified algorithm (here called SINGLE-CLOSEST-POLE).

2.1 Class Ordering

As previously mentioned, we may not assign class IDs in an arbitrary fashion. We must satisfy two goals, one mandatory and one related to compression quality.

Ordering. The POLE-COMPUTATION algorithm requires the ID of a class to be greater than the IDs of its superclasses. [DAS98, p. 133].

```
func compute_pole_table (gf, position):
  specializers = get_specializer_classes(gf, position)
  pole_table = new array of size class_count, filled with 0

  // Mark our specializers.
  for specializer in specializers:
    pole_table[specializer.class_id] = -1

  // Define a local method to create a new pole.
  // This updates several local data structures.
  next_pole_id = 0
  pole_classes = []
  func create_pole (id):
    pole_table[id] = next_pole_id
    next_pole_id += 1
    append(pole_classes, id)

  // Entry 0 will always correspond to Object,
  // the ultimate parent of all classes. We
  // always need to assign a pole, either for a
  // regular method or for a no-such-method error.
  if pole_table[0] == -1:
    default_pole_is_error = false
  else:
    default_pole_is_error = true
  create_pole(0)

  // Assign poles to all other classes.
  for (i = 1; i < class_count; i++):
    if pole_table[i] == -1:
      // There's a specializer on this class, so
      // create a new primary pole.
      create_pole(i)
    else:
      // Try to find the single "closest" pole, if any.
      closest = single_closest_pole(i, pole_table, pole_classes)
      if closest != None:
        // Copy down the closest i-pole.
        pole_table[i] = pole_table[closest]
      else:
        // Create a secondary pole.
        create_pole(i)
```

Figure 13: A modified version of the POLE-COMPUTATION algorithm by Dujardin and colleagues [DAS98, p. 139].

Locality. Our two new compression techniques rely on the pattern of "interesting" entries in the pole tables. If our class ordering groups related classes closely together, our compression ratios will be higher than they would with a random ordering.

If class IDs are assigned after all classes are known, we can use a topological sort of the class hierarchy [DAS98, p. 133]. A topological sort appears to produce reasonably locality, and it satisfies the ordering constraint.

Other class-ordering algorithms can be found in the literature, not all of which meet these constraints. The Gwydion Dylan compiler, for example, assigns class IDs using a pre-order traversal through inverse primary superclass relationships [Fah94]. This produces excellent locality—it groups classes near their primary superclass, and ignores "mix-in" classes. Unfortunately, this algorithm violates the ordering constraint.

2.2   Computing the Single Closest Pole

The POLE-COMPUTATION algorithm relies on the SINGLE-CLOSEST-POLE algorithm. The latter algorithm must distinguish between two cases:

- The current class is under the influence of a pre-existing pole. In this case, SINGLE-CLOSEST-POLE must return the ID of that pole.

- The current class is a secondary pole. In this case, SINGLE-CLOSEST-POLE must return `None`.

Dujardin and colleagues provide a mathematically-rigorous way of deciding between these two cases. Let C be a non–primary-pole class with superclasses $S_0 \ldots S_n$. Let $P_0 \ldots P_n$ be the poles influencing $S_0 \ldots S_n$. If there exists some P_i such that P_i is a subclass of $P_0 \ldots P_{i-1}$ and $P_{i-1} \ldots P_n$, then P_i influences C. If no such P_i exists, then C is a secondary pole [DAS98, pp. 133–135].

Because we call SINGLE-CLOSEST-POLE from the inner loop, it needs to be extremely efficient. We can achieve this efficiency in two ways: we can either reduce the number of subtype tests needed to find P_i among $P_0 \ldots P_n$, or we can identify special cases in which no such search is necessary. Dujardin and colleagues follow the former course with great success; we pursue the latter.

Dujardin and colleagues provide an excellent algorithm for finding P_i using only $O(n)$ subtype tests [DAS98, p. 135]. This algorithm takes advantage of the class ordering properties described in Section 2.1. Because the ID of a class must always be greater than the IDs of its superclasses, any such P_i—if it exists—must have the greatest class ID of $P_0 \ldots P_n$. So we can simply scan through $P_0 \ldots P_n$, looking for the greatest class ID. We can then perform $O(n)$ subtype tests manually. If all these subtype tests succeed, then we have found P_i. Otherwise, no such P_i exists.

Our optimizations, however, attempt to bypass this search completely. Figure 14 shows SINGLE-CLOSEST-POLE, our replacement for the original PSEUDO-CLOSEST-POLES [DAS98, p. 136]. The new algorithm handles four cases:

```
func single_closest_pole (id, pole_table, pole_classes):
  superclasses = superclass_ids_for_class_id(id)

  if superclasses.size == 1:
    // Case 1: We have only one superclass.
    return pole_table[superclasses[0]]
  else:

    // Scan over our superclasses, checking to see if our poles
    // are identical, and finding our largest pole.
    first_pole = pole_table[superclasses[0]]
    all_poles_are_identical = true
    current_largest = first_pole
    for s in superclasses:
      pole = pole_table[s]
      if pole != first_pole:
        all_poles_are_identical = false
      if pole > current_largest:
        current_largest = pole

    if all_poles_are_identical:
      // Case 2: All our superclasses share the same pole.
      return first_pole

    // Return None if our largest pole doesn't hide the rest.
    candidate = find_class_by_id(pole_classes[current_largest])
    for s in superclasses:
      pole = find_class_by_id(pole_classes[pole_table[s]])
      if not is_identical_or_subclass(candidate, pole):
          // Case 3b: At least two close poles.
          return None

    // Case 3a: One pole "hides" the rest.
    return current_largest
```

Figure 14: The Single-Closest-Pole algorithm, based on the Pseudo-Closest-Poles algorithm [DAS98, p. 136].

Case 1: 87%.[10] Class C has only one superclass S_0.

Return the pole ID of S_0.

Case 2: 13%. Classes $S_0 \ldots S_n$ all have the same pole ID.

Return the pole ID of S_0.

Case 3a: 0.003%. P_i exists.

Return the pole ID of P_i.

Case 3b: <0.0001% P_i does not exist.

Return no pole ID.

The first two cases—covering 99.997% of the entries in real world pole tables—can be handled without any subtype checks. Cases 3a and 3b can be handled as usual.

We also replaced the original subtype tests [DAS98, pp. 135–137] with the Packed Encoding described by Vitek and colleagues [VHK97].[11] The latter can be computed once for all the types in a program, eliminating the pole initialization steps from POLE-COMPUTATION. This change produces no noticeable performance penalty.

3 Compression Techniques

We implemented two new techniques for compressing pole tables: offset table fragments, and fixed-size partitioning. Table 2 lists various properties of each. Both of the new techniques provided reasonable compression despite their extreme simplicity.

Table 3 shows the compression ratios achieved by both techniques on a variety of real-world Dylan programs. We made several assumptions when calculating these pole-table sizes:

- Pole table entries require either 1 or 2 bytes, depending on the total number of entries in the corresponding dispatch matrix. The dispatch matrix itself is stored as a flat array, and pole IDs are replaced with linear offsets. This technique, as described by Dujardin and colleagues [DAS98], minimizes storage space and lookup time.

- A pointer to a fixed-size partition requires 4 bytes. This could be reduced by adding an extra layer of indirection, at the cost of some performance.

We omitted certain argument positions from these calculations. Dylan's **sealing** declarations allow us to prove that certain argument positions can never affect dispatch [Sha96]. No tables were constructed for these argument positions. Dylan also allows generic functions to dispatch on non-class types [Sha96] or to

---

[10]We calculated these percentages from the pole tables of the `d2c` compiler.
[11]Andreas Bogk implemented Packed Encoding for the `d2c` compiler.

Property	Compression Technique		
	Coloring	Offset	Partition
Worst Case	$O(CS^2)$	$O(CS)$	$O(CS)$
External type checks	Required	Unnecessary	Unnecessary
Dispatch overhead	None	Small	Small
Implementation complexity	Moderate	Trivial	Trivial
Single-table compression	No	Yes	Yes
Sensitivity to class order	None	High	Medium
Compression	46%-79%	79–89%	18–72%

Table 2: Compression techniques.

	Program	Hello-1	Hello-2	ICFP	d2c
Statistics	Classes	115	165	188	606
	Methods	744	1042	1253	2703
	Executable	857K	1.3M	2.0M	5.5M
	Generics	102	173	194	463
Arguments	Total	166	262	309	897
	Active	120	186	197	405
	Hairy	11	21	28	84
Dispatch	Single	57	111	128	342
	Multiple	41	55	58	90
Pole Tables					
Uncompressed	Total	16K	36K	46K	315K
	Single	6K	18K	24K	202K
	Multiple	9K	19K	22K	113K
Coloring	% saved	46%	58%	60%	79%
	Total	8K	15K	18K	66K
Offset	% saved	80%	79%	82%	89%
	Total	3K	8K	8K	34K
	Single	1K	3K	4K	22K
	Multiple	2K	4K	5K	11K
Partition-32	% saved	18%	33%	44%	72%
	Total	13K	24K	26K	87K
	Single	5K	12K	13K	56K
	Multiple	7K	13K	13K	31K
Dispatch Matrices					
	Entries	1,176	1,527	1,731	3,397

Coloring compression performs better if we combine the space used by single-dispatch and multi-dispatch functions, because this increases the opportunity to overlay pole tables in memory.

Table 3: Compression results.

rely on information not available at compile time. These arguments have been labeled "hairy" in Table 3.

We collected data by modifying the d2c compiler, an optimizing Dylan compiler provided by the Gwydion Project [Fah94]. Test programs included d2c itself, a ray-tracer submitted to the ICFP 2000 programming contest [BDGH00], and two versions of "Hello, World." The first version included only the core Dylan run-time library, but the second version included several large, semi-standard libraries. These applications represent a range of sizes and programming styles. Executable sizes were calculated using static linking, even though recent releases of the d2c compiler support shared libraries.

3.1 Argument Coloring

Dujardin and colleagues suggest a compression algorithm based on graph coloring [DAS98, pp. 150–152]. This compression algorithm uses information about legal argument types to overlay pole tables in memory. It adds no dispatch overhead to a statically-typed language, but it requires one type-check per argument in a dynamically-typed language.

The code in Figure 15 performs the actual color assignment. The nested loops in Step 2 have a worst-case time of $O(CS^2)$, where C is the number of classes in the program, and S is the number of selectors. This case occurs when the pole-tables are "full"—when every entry points to a valid method—and some constant proportion of the classes in the program have no subclasses.

The dispatch code for a two-argument generic function m resembles Figure 16.

3.2 Offset Table Fragment

We can also compress pole tables by trimming redundant entries from each end of the ends. If class IDs are assigned in the right order, the "interesting" portion of the table typically appears somewhere in the middle. The ends of the table, on the other hand, usually inherit their behavior from the class *Object*. (This default behavior may be either a regular method specialized on *Object* or a no-applicable-method error.) By trimming away any such leading or trailing entries, we can achieve very respectable compression.

Unfortunately, this method is very sensitive to the order in which class IDs are assigned (see Section 2.1). The results in Table 3 are based on a topological sort of the entire class hierarchy, with no respect for compilation units. If we were to assign consecutive IDs to the classes of each compilation unit, the "interesting" portion of the pole array would become much larger.

The dispatch code for a two-argument generic function m resembles Figure 17. Note that the compiled version of this function will involve two[12] con-

---

[12]Chris Hawblitzel suggested performing each bounds check using an unsigned integer comparison. This causes negative indices to be treated as large positive indices, and reduces the number of branch instructions required from four to two.

```
func assign_colors (pole_tables):

  // Step 1: Build conflict sets for individual selectors.
  selector_conflict_sets = new array of size class_count,
                                 filled with empty sets
  for pole_table in pole_tables:
    for class in pole_table:
      if class has no subclasses:
        if class dispatches to non-error pole:
          add pole table to matching conflict set

  // Step 2: Caclute conflict sets for entire pole tables.
  for scs in selector_conflict_sets:
    for pole_table_1 in scs:
      for pole_table_2 in scs:
        if pole_table_1 is not pole_table_2:
          add pole_table_2 to conflict set for pole_table_1

  // Step 3: Allocate colors.
  sort pole_tables by descreasing conflict set size
  assigned_colors = empty list
  for pole_table in pole_tables:
    for color in assigned_colors:
      for assigned_pole_table in color:
        if pole_table conflicts with assigned_pole_table:
          skip to next color
        assign pole_table to color
        skip to next pole_table
    create new_color
    assign pole_table to new_color
    append new_color to colors
```

Figure 15: Pseudo-code for coloring compression [DAS98, pp. 150–152].

```
func dispatch_m (arg1, arg2):

  // Check argument types. In static languages,
  // this may be done at compile time.
  typecheck(arg1, m_arg1_type)
  typecheck(arg2, m_arg2_type)

  // Dispatch code.
  index1 = m_pole_table_1[class_id(arg1)]
  index2 = m_pole_table_2[class_id(arg2)]
  method = m_methods[m_flat_dispatch_matrix[index1 + index2]]
  return method(arg1, arg2)
```

Figure 16: Dispatch code for argument-coloring compression [DAS98].

ditional branch statements, which may cause some pipeline problems on certain processors.

3.3 Fixed-Size Partitioning

Vitek and colleagues describe several ways to compress traditional C++ vtables using variable-sized partitions [VH96]. These partitioning algorithms can be adapted for compressing pole tables.

We designed a simple algorithm based on fixed-size partitions. We begin by breaking the pole table into fixed-size blocks (typically 32 or 64 entries wide), and searching for blocks which contain only one value. In most cases, this repeated value will be either zero or one. We then replace these blocks with pointers to statically-allocated blocks from our run-time library. Finally, we reassemble our blocks into a two-tier pole table.

The dispatch code for a two-argument generic function m resembles Figure 18. Note that when the chunk size is a power of 2, the division and modulo operations can be performed using bit-wise operations.

4 Related Research

Many techniques for dispatching generic functions appear in the literature. These techniques include tree-based dispatch, table-based dispatch and predicate dispatch. Good bibliographies can be found in [EKC98] and [DAS98].

Early systems relied heavily on run-time caching of dispatch information either at the call site or on a per-generic-function basis. Other systems built decision trees at compile-time. Recent examples of both approaches can be found in the d2c compiler [Fah94].

According to Bachrach [Bac00], the Functional Developer compiler (formerly known as Harlequin Dylan) creates new dispatch trees on the fly by plugging

```
func dispatch_m (arg1, arg2):

  // Get pole table index for the first argument.
  offset_id = class_id(arg1) - m_pole_table_1_offset
  if offset_id < 0 || offset_id >= m_pole_table_1_size:
    index1 = 0
  else:
    index1 = m_pole_table_1[offset_id]

  // Get pole table index for the second argument.
  offset_id = class_id(arg2) - m_pole_table_2_offset
  if offset_id < 0 || offset_id >= m_pole_table_2_size:
    index2 = 0
  else:
    index2 = m_pole_table_2[offset_id]

  // Call the appropriate method.
  method = m_methods[m_flat_dispatch_matrix[index1 + index2]]
  return method(arg1, arg2)
```

Figure 17: Dispatch code using offset table fragments.

```
func dispatch_m (arg1, arg2):

  // Get pole table index for the first argument.
  id = class_id(arg1)
  partiton = m_partition_table_1[id / PARTITION_SIZE]
  index1 = partition[id % PARTITION_SIZE]

  // Get pole table index for the second argument.
  id = class_id(arg2)
  partiton = m_partition_table_2[id / PARTITION_SIZE]
  index2 = partition[id % PARTITION_SIZE]

  // Call the appropriate method.
  method = m_methods[m_flat_dispatch_matrix[index1 + index2]]
  return method(arg1, arg2)
```

Figure 18: Dispatch code using fixed-size partitions.

together **engine nodes** [Unk]. These are small, tail-recursive code fragments with local data. Various sorts of nodes can make dispatch decisions, modify the dispatch tree, or signal errors. More recently, Bachrach has implemented a specialized compiler that lives in the Functional Developer run-time environment and transparently compiles these dispatch trees into optimized native code during garbage collection.

Ernst and colleagues have developed a much more general dispatch scheme based on predicates [EKC98]. Their approach combines features of generic functions, logical inference systems, and ML-style dispatch. They convert predicate expressions into decision trees and balance the decision trees using heuristics and profiler feedback.

5 Future Directions for Research

Several directions remain for future research. These include work with dispatch matrices, class ordering, alternate compilation scenarios and on-the-fly compilation.

5.1 The Dispatch Matrix and Method Monotonicity

Dujardin and colleagues provide an algorithm for efficiently computing the dispatch matrix [DAS98]. As they note, however, this algorithm only applies to languages which guarantee **method monotonicity**. In such languages, the relative specificity of two methods depends solely on their formal parameters, and never on their actual parameters. Dylan, unfortunately, violates this constraint [Sha96, pp. 96–98]. Therefore, new algorithms must be designed before any of the techniques in this paper can be applied to Dylan. This does not appear to be an easy problem.

However, it would be possible to guarantee method monotonicity in Dylan if certain rules about inheritance were enforced. In particular, Dylan's rules for method specificity depend on the order of the class precedence lists of the actual parameters of a generic function [Sha96, p. 96]. If all classes were required to have consistent class precedence lists, then method monotonicity would be guaranteed.

5.2 New Class Ordering Algorithms

All the compression techniques in this paper depend, to some extent, on the order in which we assign class IDs. If we can group related classes more closely, we can improve our compression ratios.

The Gwydion Dylan compiler currently ignores "mix-in" classes when assigning class IDs, as described in Section 2.1. This means that all the subclasses of a non-"mix-in" class will normally have consecutive IDs. Unfortunately, this algorithm does not assign class IDs in the order required for building pole tables.

Therefore, it might be worthwhile to design a biased topological sort which attempts to preserve certain grouping properties whenever practical.

5.3   Separate Compilation, Shared Libraries and Run-time Updates

The algorithms in this paper make some inconvenient assumptions about the compilation model we choose. In particular, they assume that we have already seen the entire program before we assign class IDs or build dispatch tables.

These assumptions cause problems in several situations:

Separate compilation. We shouldn't plan to build all our tables just before linking an application. This slows down compilation cycle, and makes development less pleasant. In an ideal world, we would build our dispatch tables incrementally as we compiled individual classes and generic functions.

Shared libraries. Shared libraries present two issues. First, we'd like to store as many dispatch tables as possible in shared libraries, not in the main application. But we don't know the full class hierarchy (or all the methods defined on a generic function) until we've seen the whole program. Second, we need to be able to fix bugs in shared libraries without breaking applications which have already linked against them.

Run-time updates. Dylan allows the creation of new classes and the modification of existing generic functions *at run-time*. To support run-time creation of classes, we must be able to add new entries to all our pole tables and dispatch matrices. To support run-time modification of generic functions, we must be able to re-compute all the tables for single generic function efficiently. These operations are normally used to support interactive development environments similar to those traditionally provided by various LISP dialects.

It might be possible to modify the algorithms in this paper to handle these scenarios better. In particular, if we keep the tables small—and we can build them quickly enough—we may be able to store them in the application binary. If we assign class IDs in the right order, we might be able to store pre-compressed table fragments in shared libraries and determine the correct offsets at run-time. If we allow new entries to be inserted into our pole tables and dispatch matrices, we can support run-time class creation.

The hardest of these problems, though, is fixing bugs in shared libraries without breaking pre-existing applications. C++ provides few solutions to this problem. Java escapes it by using a high-level linkage model and just-in-time compilation. These are important challenges for any object-oriented language, and they should be seriously considered when designing new languages.

5.4 Run-time Recompilation of Dispatch Functions

The dispatch routines in Figures 9, 16, 17 and 18 contain various constant values beginning with "m_". Depending on the compilation model used, these values may not be known while generating the dispatch routine. But repeatedly fetching these values from memory may become quite expensive.

Bachrach faced a similar problem [Bac00] when working on Functional Developer. The FD run-time library created new dispatch trees on the fly using engine nodes [Unk]. According to Bachrach, these engine nodes spent too much time fetching data from memory.

To solve this problem, Bachrach wrote a tiny, specialized compiler which translated engine nodes into native machine code [Bac00]. This compiler ran during garbage collection, and optimized all the engine nodes created since the last pass of the collector.

The dispatch routines described in this paper could be optimized in a similar fashion. The necessary compiler would be even simpler, and it could run during the first invocation of a generic function.

6 Conclusion

We have identified two special-case optimizations that cover 99.997% of pole table entries, reducing the number of typechecks required during table creation by several orders of magnitude.

We have also presented two new algorithms for compressing pole tables. One of these algorithms, OFFSET-TABLE-FRAGMENTS, provides better compression results than the previous technique, despite its extreme simplicity and lower compile-time complexity. Both of the new compression algorithms, however, present a run-time performance trade-off. Although they provide "built-in" typechecks for function arguments, they also require slightly more complex dispatch code.

To evaluate our algorithms, check the `GD_2_3_3_GF_RESEARCH` branch out of the Gwydion Dylan CVS archive described on the web at
`http://www.gwydiondylan.org/downloading.phtml`.

List of Figures

List of Tables

References

[Bac00]   Jonathan Bachrach. Unpublished talk at MIT Media Lab, 2000. Slides available at:
http://www.ai.mit.edu/~jrb/Projects/dylan-dispatch.htm.

[BDGH00] Andreas Bogk, Jeff Dubrule, Gabor Greif, and Bruce Hoult. Raytracer submitted to the ICFP 2000 programming contest, 2000.

[BKK+86] Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. CommonLoops: merging Lisp and object-oriented programming. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Application, OOPSLA'86*, Portland, OR, September 1986. ACM Press.

[Cor00]   Thomas Cormen, 2000. Private conversation.

[DAS98]   Eric Dujardin, Eric Amiel, and Eric Simon. Fast algorithms for compressed multimethod dispatch table generation. *ACM Transactions on Programming Languages and Systems*, 20(1):116–165, 1998. Originally written in 1996.

[EKC98]   Michael D. Ernst, Craig Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In *ECOOP'98*, pages 116–165, Brussels, Belgium, July 1998.

[Fah94]   Scott E. Fahlman. Gwydion: An integrated software environment for evolutionary software development and maintenance. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15217, March 1994.

[Sha96]   Andrew Shalit. *The Dylan Reference Manual*. Adison–Wesley Developers Press, Reading, Massachusetts, 1996.

[Unk]   Unknown. Unpublished engine node dispatch paper, Harlequin, Ltd. Mentioned by Jonathan Bachrach in talk at MIT Media Lab. Availability and authorship unknown.

[VH96]   Jan Vitek and R. Nigel Horspool. Compact dispatch tables for dynamically typed programming languages. In *Proceedings of the International Conference on Compiler Construction*, 1996.

[VHK97]   Jan Vitek, R. Nigel Horspool, and Andreas Krall. Efficient type inclusion tests. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Application, OOPSLA '97*, Atlanta, GA, October 1997. ACM Press.