University of Alberta

Multi-Method Dispatch Using Multiple Row Displacement

by

Candy Siu Tung Pang ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 1999

The fear of the Lord is the beginning of knowledge,
but fools despise wisdom and discipline.
Proverbs 1:7

Abstract

Most of the widely used object-oriented languages, such as C++, Java and Samlltalk, use single-receiver dispatch. In single-receiver dispatch, the code for a particular message expression is determined by the dynamic type of the receiver object and the static signature of the message. In contrast, there are object-oriented languages, like CLOS, Cecil and Dylan that use multi-method dispatch. In multi-method dispatch, the code for a message expression is determined at run-time by the name of the message and the dynamic types of all message arguments, including the receiver object.

Multiple Row Displacement(MRD) is a new dispatch technique for multi-method languages. It is based on compressing an n-dimensional table using an extension of the single-receiver row displacement mechanism. This thesis presents the new algorithm and provides experimental results that compare it with implementations of existing techniques: compressed n-dimensional tables, look-up automata and single-receiver projection. MRD has the fastest dispatch performance and uses comparable space to these other techniques. This thesis also discusses how to apply MRD in statically typed and non-statically typed languages.

Acknowledgements

First of all, thanks to the Lord for His sufficient providing in the last two years that I may complete my degree.

> Therefore I tell you, do not worry about your life, what you will eat or drink; or about your body, what you will wear. Is not life more important than food, and the body more important than clothes? (Matthew 6:25)

Second, I would like to thank my, could not be better, supervisor, Duane Szafron, who provided me a thesis topic, and guidance through out the year.

Then, I thank National Sciences and Engineering Research Council for providing financial support.

Next, I have to thank all the members of the Dispatch team. Wade Holst has taught me a lot; shared with me all of his source code and his C++ programming tool, *Cxx*. Yuri Leontiev has been a wonderful source of information, and his suggestion has dramatically improved the time and space efficiency of the Multiple Row Displacement algorithm. Also to Christopher Dutchyn and Thomas Harke, for their invaluable feedback in our weekly meeting.

Finally, I have to thank my family and friends who encouraged me to carry on with my studies.

Contents

List of Figures

List of Symbols

σ - Name of a behavior.

o_i - Argument object at the i position.

type(o_i) - Type of the argument at the i position.

T^i - Type of argument at the i position.

T_i - One of the types in a system, with type number i.

num(T_i) - Type number of the type T_i.

$T_j \prec_1 T_i$ - T_j is a direct subtype of T_i.

$T_j \prec T_i$ - T_j is a subtype of T_i.

$T_j \preceq T_i$ - T_j is T_i, or a subtype of T_i.

P - Product type.

$T^1 \times T^2 \times ... \times T^k$ - Product type for k-arity behavior.

$P_j \prec_1 P_i$ - P_j is a direct subtype of P_i.

$P_j \prec P_i$ - P_j is a subtype of P_i.

$\mathcal{H}$ - A type hierarchy.

$|\mathcal{H}|$ - Number of types in the type hierarchy $\mathcal{H}$.

$\mathcal{H}^k$ - k-degree product-type graph, $k \geq 1$.

$\mathcal{B}_\sigma^k$ - A behavior named σ with k arity.

σ_i - The i^{th} method defined for behavior σ.

K - The maximum arity for all behaviors in the system.

S - Selector table.

M - Master array in row displacement.

I - Index array in row displacement.

B - Behavior array in multiple row displacement.

D_σ^k - K-dimensional table for the k-arity behavior σ.

$D_\sigma^{k,CNT}$ - Compressed n-dimensional table for the k-arity behavior σ.

G_σ^i - Index-group array for the i^{th} argument of the k-arity behavior σ in compressed n-dimensional table.

$\mathcal{H}_i$ - The i^{th} hierarchy table in single-receiver projections.

A_σ^k - Lookup automaton for the k-arity behavior σ.

$D_\sigma^{k,MRD}$ - Method-map for the k-arity behavior σ in multiple row displacement with byte array.

STI - Selector Table Indexing.

SC - Selector Coloring.

RW - Row Displacement.

CNT - Compressed N-Dimensional Table.

SRP - Single-Receiver Projections.

LUA - Lookup Automata.

EPD - Efficient Predicate Dispatch.

PTS - Product Type Search.

MRD - Multiple Row Displacement.

MRD-B - Multiple Row Displacement with byte array.

Chapter 1

Introduction

Object-oriented languages can be separated into *single-receiver languages* and *multi-method languages*. *Single-receiver languages* use the dynamic type of a dedicated *receiver* object in conjunction with the method name to determine the method to execute at run-time. *Multi-method languages* [6] use the dynamic types of one or more arguments[1] in conjunction with the method name to determine the method to execute. In single-receiver languages, a call-site can be viewed as a message send to the receiver object. In multi-method languages, a call-site can be viewed as the execution of a behavior on a set of arguments. The run-time determination of the method to invoke at a call-site is called *method dispatch*. Note that languages like C++ and Java that allow methods with the same name but different static argument types are not performing actual run-time dispatch on the types of these arguments; the static types are simply encoded within the method name at compile time. For example, consider two Java methods, $A.alpha(Integer)$ and $A.alpha(Float)$, defined in a class A. The Java names of these two methods are different, since they are $alphaInteger$ and $alphaFloat$ respectively.

Since most of the commercial object-oriented languages are single-receiver languages, many efficient dispatch techniques have been invented for such languages [21]. However, multi-method dispatch is more suitable to some methods than single-receiver dispatch. For example, the operator '+' can be considered as $add(Number1, Number2)$. If $Number1$ is an integer, and $Number2$ is a float number, dispatching on the type of $Number1$ by single-receiver dispatch returns $add(Integer, Integer)$, while

[1] In the rest of this thesis, I will assume that dispatch occurs on all arguments.

add(*Integer*, *Float*) should be returned. In fact there will be four different methods that implement *add*: *add*(*Integer*, *Integer*), *add*(*Integer*, *Float*), *add*(*Float*, *Integer*) and *add*(*Float*, *Float*). Many other numerical operators also need the extra expressive power of multi-methods.

Multi-method dispatch is also very useful in many graphical user interface operations. For example, the method *drag_and_drop*(*source*, *target*) can be expressed more efficiently by multi-method dispatch. The *source* object could be a circle, rectangle, or other visual component. The *target* object could be a canvas, browser, or other displaying component. In this case, both the type of the *source* object and the type of the *target* object must be considered to perform the actual operation. Multi-method dispatch provides the convenience. There are some multi-method languages in use, such as Cecil [7], CLOS [5], and Dylan [4]. However, They are not as popular as Java or C++.

Since multi-methods are not supported in any popular commercial languages, it is not easy to convince users to switch language, just to use multi-methods. It is also not easy to convince language designers and implementors to extend existing languages to support multi-methods, since multi-method dispatch is slower than single-receiver dispatch. The acceptance of multi-method languages depends on faster dispatch algorithms and faster machines.

There are three major categories of method dispatch: *search-based*, *cache-based* and *table-based*. The simplest search-based technique is called *method lookup*, which looks in a dictionary based on the message name and dynamic argument types. If a match is not found, it looks in other dictionaries based on super-types of the argument types. It keeps looking until a method is found. If no method is found an error is reported. A *cache-based* technique looks in either a global or local cache at the time of dispatch to determine if the method for a particular call-site has already been determined. If it has been determined, that method is used. Otherwise, a *cache-miss* technique is used to compute the method, which is then cached for subsequent executions. A *table-based* technique pre-determines the method for every possible call-site, and records these methods in a table. At dispatch-time, the method name and dynamic argument types form an index into this table. This thesis focuses exclusively on table-based techniques. The advantage of using table-based techniques is that they

have constant dispatch time. In addition, even when cache-based techniques are used, table-based techniques can be effectively used for cache-misses.

This thesis presents a new multi-method table-based dispatch technique. It uses a time efficient n-dimensional dispatch table that is compressed using an extension of a space efficient row displacement mechanism. Since the technique uses multiple applications of row displacement, it is called Multiple Row Displacement and will be abbreviated as MRD. MRD works for methods of arbitrary arity. Its execution speed and memory utilization are analyzed and compared to other multi-method table-based dispatch techniques.

The rest of this thesis is organized as follows. Chapter 2 introduces some notation for describing multi-method dispatch. Chapter 3 reviews existing single-receiver and multi-method dispatch techniques. Chapter 4 presents the new multi-method table-based technique. Chapter 5 shows the data structures used to implement the new algorithm. Chapter 6 presents time and space results for the new technique and compares it to existing techniques. Chapter 7 presents future work and conclusions.

Chapter 2

Terminology for Multi-Method Dispatch

2.1 Notation

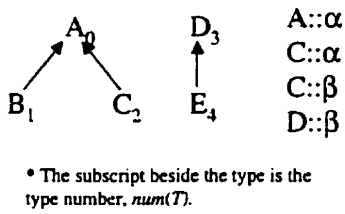
The notation in this thesis originated with the dispatch team at the University of Alberta. Expression 2.1 shows the form of a k-arity multi-method call-site. Each argument, o_i, represents an object, and has an associated *dynamic type*, $T^i = type(o_i)$. Let $\mathcal{H}$ represent a type hierarchy, and $|\mathcal{H}|$ be the number of types in the hierarchy. In $\mathcal{H}$, each type has a type number, $num(T)$. A directed *supertype edge* exists between type T_j and type T_i if T_j is a *direct subtype* of T_i, which is denoted as $T_j \prec_1 T_i$. If T_i can be reached from T_j by following one or more supertype edges, T_j is a *subtype* of T_i, denoted as $T_j \prec T_i$.

$$\sigma(o_1, o_2, \ldots, o_k) \tag{2.1}$$

In the single-receiver domain, Expression 2.1 can be written as Expression 2.2.

$$o_1.\sigma(o_2, \ldots, o_k) \tag{2.2}$$

Method dispatch is the run-time determination of a method to invoke at a call-site. When a method is defined, each argument, o_i, has a specific static type, T^i. However, at a call-site, the dynamic type of each argument can either be the static type, T^i, or any of its subtypes, $\{T | T \preceq T^i\}$. For example, consider the type hierarchy and method definitions in Figure 2.1(a), and the code in Figure 2.1(b). The static type of `anA` is `A`, but the dynamic type of `anA` can be either `A`, `B` or `C`. In general, the dynamic



(a) Type Hierarchy

```
A anA;
if( ... )
    anA = new A();
else if( ... )
    anA = new B();
else
    anA = new C();
anA.α();
```

(b) Code Requiring Method Dispatch

Figure 2.1: An example hierarchy and program segment requiring method dispatch

type of an object at a call-site is not known until run-time, so method dispatch is necessary.

Although multi-method languages might appear to break the conceptual model of sending a message to a receiver, this idea can be maintained by introducing the concept of a product-type. A *k-arity product-type* is an ordered list of k types denoted by $P = T^1 \times T^2 \times ... \times T^k$. The *induced k-degree product-type graph*, $k \geq 1$, denoted $\mathcal{H}^k$, is implicitly defined by the edges in $\mathcal{H}$. Nodes in $\mathcal{H}^k$ are k-arity product-types, where each type in the product-type is an element of $\mathcal{H}$. Expression 2.3 describes when a directed edge exists from a child product-type $P_j = T_j^1 \times T_j^2 \times ... \times T_j^k$ to a parent product-type $P_i = T_i^1 \times T_i^2 \times ... \times T_i^k$, which is denoted $P_j \prec_1 P_i$.

$$P_j \prec_1 P_i \Leftrightarrow \exists u, 1 \leq u \leq k : (T_j^u \prec_1 T_i^u) \wedge (\forall v \neq u, T_j^v = T_i^v) \qquad (2.3)$$

The notation $P_j \prec P_i$ indicates that P_j is a *sub-product-type* of P_i, which implies that P_i can be reached from P_j by following edges in the product-type graph $\mathcal{H}^k$. Figure 2.2 presents a sample inheritance hierarchy $\mathcal{H}$ and its induced 2-arity product-type graph, $\mathcal{H}^2$. Three 2-arity methods (γ_1 to γ_3) for behavior γ have been defined on $\mathcal{H}^2$ and associated with the appropriate product-types.[1] Note that for real inheritance hierarchies, the product-type hierarchies, $\mathcal{H}^2$, $\mathcal{H}^3$, ..., are too large to store explicitly.

[1]The method γ_4 in the dashed box is an implicit inheritance conflict definition, and will be explained later.

Therefore, it is essential to define all product-type relationships in terms of relations between the original types, as in Expression 2.3.

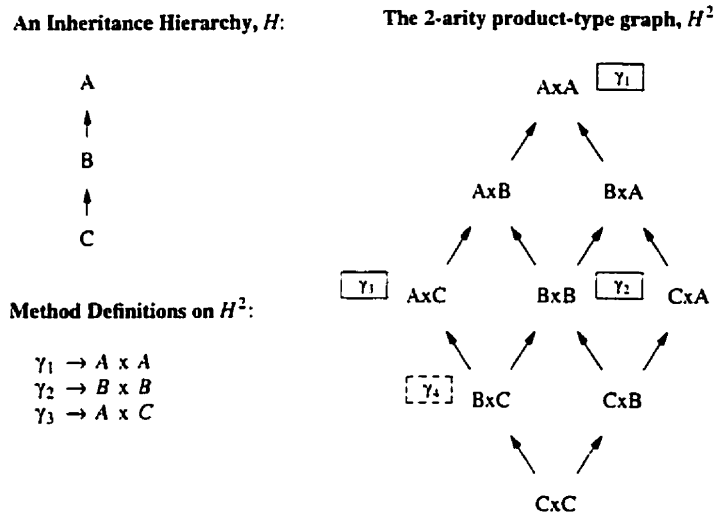


Figure 2.2: An Inheritance Hierarchy, $\mathcal{H}$, and its induced Product-Type Graph $\mathcal{H}^2$

Next, I define the concept of a *behavior*. A behavior corresponds to a generic-function in CLOS and Cecil, to the set of methods that share the same signature in Java, and the set of methods that share the same message selector in Smalltalk. Behaviors are denoted by $\mathcal{B}^k_\sigma$, where k is the arity and σ is the name. The maximum arity for all behaviors in the system is denoted by K. Multiple methods can be defined for each behavior. A method for a behavior named σ is denoted by σ_j. If the static type of the i^{th} argument of σ_j is denoted by T^i, the list of argument types can viewed as a product-type, $dom(\sigma_j) = T^1 \times T^2 \times ... \times T^k$. With multi-method dispatch, the dynamic types of all arguments are needed.

2.2 Inheritance Conflicts

Section 2.1 defines subtype and supertype relationship between types in an inheritance hierarchy. If each type in an inheritance hierarchy is allowed to have at most one immediate supertype, as in Figure 2.3(a), this hierarchy is called a *single inheritance* hierarchy. On the other hand, if types are allowed to have more than one immediate supertype, as in Figure 2.3(b) and (c), this hierarchy is called a *multiple inheritance*

hierarchy.

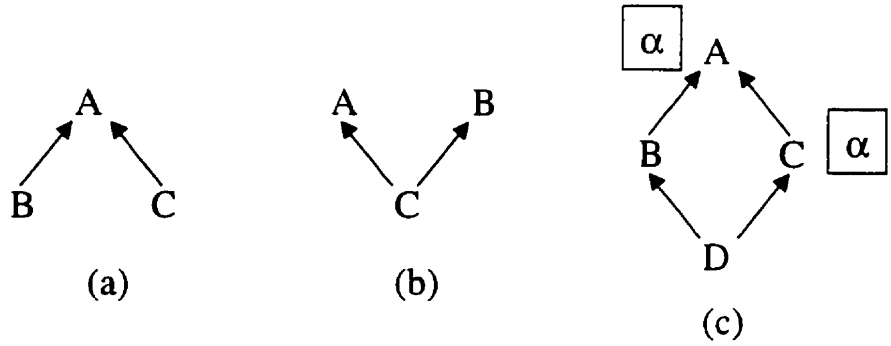


Figure 2.3: Single Inheritance and Multiple Inheritance

In single-receiver languages with multiple inheritance, the concept of *inheritance conflict* arises. In general, an inheritance conflict occurs at a type T if two different methods of a behavior are visible (by following different paths up the type hierarchy) in supertypes T_i and T_j. For example, if two methods, $A :: \alpha$ and $C :: \alpha$, are defined for the type hierarchy in Figure 2.3(c), then there will be an inheritance conflict for behavior α at type D. Since D is a subtype of C, $C :: \alpha$ is visible to type D. At the same time, D is a subtype of B, which in turn is a subtype of A, therefore $A :: \alpha$ is also visible to type D. This situation is called an inheritance conflict.

Most languages relax this definition slightly. Assume that n different methods of a behavior are defined on the set of types $\mathcal{T} = \{T_1, ..., T_n\}$, where $T \preceq T_1, ..., T_n$. Then, the methods defined in two types, T_i and T_j in $\mathcal{T}$, do not cause a conflict in T, if $T_i \prec T_j$, or $T_j \prec T_i$, or $\{\exists\, T_k \in \mathcal{T} \mid T_k \prec T_i \;\&\; T_k \prec T_j\}$. As described in the last paragraph, an inheritance conflict occurs at type D, if $A :: \alpha$ and $C :: \alpha$ are defined for the type hierarchy in Figure 2.3(c). However, after the relaxation, no inheritance conflict results, since type C is a subtype of A. That is, $C :: \alpha$ will be selected over $A :: \alpha$.

Inheritance conflicts can also occur in multi-method languages, and are defined in an analogous manner. A conflict occurs when a product-type can see two different method definitions by looking up different paths in the induced product-type graph $T^1 \times T^2 \times ... \times T^k$. Interestingly, inheritance conflicts can occur in multi-method languages even if the underlying type hierarchy, $\mathcal{H}$, has single inheritance. For example,

in Figure 2.2, the product-type $B \times C$ has an inheritance conflict, since it can see two different definitions for behavior γ (γ_3 in $A \times C$ and γ_2 in $B \times B$). For this reason, an implicit conflict method, γ_4, is defined in $B \times C$ as shown in Figure 2.2. Similar to single-receiver languages, relaxation can be applied. Assume that n methods are defined in product-types $\mathcal{P} = \{P_1, ..., P_n\}$, and let $P \prec P_1, ..., P_n$. Then, the methods in P_i and P_j do not conflict in P if $P_i \prec P_j$, or $P_j \prec P_i$, or $\{\exists\, P_k \in \mathcal{P} \mid P_k \prec P_i \,\&\, P_k \prec P_j\}$. In multi-method languages, it is especially important to use the more relaxed definition of an inheritance conflict. Otherwise, a large number of inheritance conflicts would be generated for almost every method definition.

2.3 Reflexive versus Non-Reflexive Environment

A program is running in a non-reflexive environment, if all types and methods are defined before the execution of the program. Hence, no types or methods can be changed during execution. Programs written in C++ are running in a non-reflexive environment. On the other hand, a program is running in a reflexive environment, if the type hierarchy and method definitions can be changed during program execution. Programs written in Smalltalk, Prolog and CLOS are running in a reflexive environment. The situation in Java is more complicated since limited reflexive compabilities were added in the latest release.

In a reflexive environment, three categories of changes can happen during program execution: (1) add or drop a type, (2) link or unlink a type to another type, (3) add, delete, or change a method. Cache-based dispatch techniques do not work very well in a reflexive environment. A small change (e.g. linking two types) in the environment may require the entire cache to be rebuilt from scratch. This removes the advantage of using a cache. Some table-based dispatch techniques allow their tables to evolve incrementally as the environment evolves [22]. Other table-based dispatch techniques must rebuild the entire set of tables when a change occurs. In either case, since the environment needs access to the table for the changes, no dispatch can occur when the tables are being modified.

Reflexivity is a complicated problem, which requires further study. Therefore, this thesis concentrates on dispatch in a non-reflexive environment only.

2.4 Statically Typed versus Non-Statically Typed

Some programming languages (C++, Java, Eiffel) require each variable to be declared with a static type. These languages are called *statically typed* languages. Other languages (Smalltalk, CLOS) which do not declare static types for variables, are called *non-statically typed* languages. In statically typed languages, a type checker can be used at compile-time to ensure that all call-sites are type-valid. A call-site is *type-valid*, if it has either a defined method for the message or an implicitly defined conflict method. In contrast, a call-site is type-invalid, if dispatching the call-site will lead to *method-not-understood*. For example, the static type of the variable anA is A in Figure 2.1(b). The dynamic type of anA can be either A, B or C, since B and C are subtypes of A. Since the message α is defined for type A, no matter what its dynamic type is, anA can understand the message α. Therefore, the type checker can tell at compile-time that the call-site $anA.\alpha()$ is type-valid. Consider another variable aD with static type D. A call-site $aD.\alpha()$ would be type invalid since no method for α is defined on D. The type checker would find at compile-time that the call-site $aD.\alpha()$ is type-invalid, and return a compile-time error.

In statically typed languages with implicitly defined conflict methods, no type-invalid call-site will be dispatched during execution. However, in non-statically typed languages, call-sites may be type-invalid. As will be shown in Section 3.2.3, any dispatch technique that uses compression may return a method for a different behavior due to selector aliasing. Therefore, in non-statically typed languages, an extra check must be made to ensure that the computed method is applicable for the dispatched behavior. This check must also ensures that the dynamic type of each argument is a subtype of the declared static parameter type in the method. For non-statically typed languages, the Multiple Row Displacement dispatch technique introduced in this thesis must perform this extra check.

Chapter 3

Existing Dispatch Techniques

As mentioned in the introduction, there are three categories of method dispatch: search-based, cache-based and table-based. There is only one viable search-based single receiver dispatch technique called *method lookup*. Method lookup searches the method dictionaries for the behavior, σ, starting from the receiver's type, and going up the inheritance chain, until a method for σ is found. However, method lookup is very time inefficient. Smalltalk and Java use method lookup, but only as a cache-miss technique. Cache-based techniques have been extensively used in single-receiver languages, like Smalltalk [20]. Cache-based single-receiver dispatch techniques are described in Section 3.1. Many time-efficient single-receiver table-based dispatch techniques are also available, and they are presented in Section 3.2. They have been ignored in most commercial implementation due to large memory requirements. However, as the price of memory gets lower, they are more practical. Table-based techniques are even more practical in multi-method languages, since cache-based techniques also require extensive memory for multi-methods, as shown in Section 3.3. Section 3.4 introduces three existing table-based multi-method dispatch techniques, and Section 3.5 introduces two existing search-based multi-method dispatch techniques.

3.1 Cache-Based Single-Receiver Dispatch Techniques

Since cache-based single-receiver dispatch techniques are not very relevant to this thesis, I give only a brief description of each technique.

1. *Global Lookup Cache*([19, 26]) uses $< T, \sigma >$ as a hash key into a global cache, whose entries store a type, T, a selector, σ, and a method address. During a dispatch, if the entry retrieved from the global cache by $< T, \sigma >$ contains a method for the correct type and selector, it can be executed immediately. Otherwise, a cache-miss technique (usually method lookup) is called to obtain the correct method address. The resulting method address is stored in the global cache.

2. *Inline Cache*([12]) caches addresses at each call-site. The initial address at each call-site invokes the cache-miss technique, which modifies the call-site once a method address is obtained. Subsequent executions of the call-site invoke the previously computed method. Within each method, a *method prologue* exists to ensure that the receiver class matches the expected class. Otherwise, the cache-miss technique is called to recompute and modify the call-site address.

3. *Polymorphic Inline Caches*([24]) cache multiple addresses in a behavior specific *stub-routine*. On the first invocation of a stub-routine, the cache-miss technique is called. However, each time the cache-miss algorithm is called the stub is extended by adding code to compare subsequent receiver types against the current type, and providing a direct function call if the test succeeds.

3.2 Table-Based Single-Receiver Dispatch Techniques

There are five known single-receiver table-based dispatch techniques: Selector Table Indexing, Row Displacement, Selector Coloring, Compact Selector-Indexed Tables, and Virtual Function Tables. Since Selector Table, Row Displacement and Selector Coloring are used in the description of other multi-method dispatch techniques, I will briefly introduce these three techniques in this chapter. Please see [28] and [29] for details about Compact Selector Indexed Tables, and see [18] for details about Virtual Function Tables, which are used in C++.

3.2.1 Selector Table Indexing (STI)

In single-receiver table dispatch, the method address can be calculated in advance for every legal class/behavior pair, and stored in a *selector table*, S. Figure 3.1 shows the selector table for the type hierarchy and method definitions in Figure 2.1(a). An empty table entry means that the behavior cannot be applied to the type. At run time, the behavior and the dynamic type of the receiver are used as indices into S [11]. In the literature [15], this algorithm is known as Selector Table Indexing or STI.

S	A_0	B_1	C_2	D_3	E
α	A::α	A::α	C::α	--	--
β	--	--	C::β	D::β	D::β

Figure 3.1: Selector Table

Although STI provides efficient dispatch, its large memory requirements prohibit it from being used in real systems. For example, there are 961 types and 12130 different behaviors in the VisualWorks 2.5 Smalltalk hierarchy. If each method address required 4 bytes, then the selector table would be more than 46.6 Mbytes ($961 \times 12130 \times 4$ *bytes*). Fortunately, 95% of the entries in the selector table for single-receiver languages are empty [14], so the table can be compressed.

3.2.2 Row Displacement (RD)

Row displacement (RD) reduces the number of empty entries by compressing the two-dimensional selector table into a one-dimensional array [14, 16]. As illustrated in Figure 3.2, each row in S is shifted by an offset until there is only one occupied entry in each column. Then, this structure is collapsed into a one-dimensional *master array*, M. When the rows are shifted, the shift indices (number of columns each row has been shifted) are stored in an index array, I.

At run-time, the behavior is used to find the shift index from the index array, I. In fact, each behavior has a unique index determined at compile time, and it is this index which is used to represent the behavior in the compiled code. For simplicity, I will just use the behavior name in this thesis. The shift index is added to the type number of the receiver to form an index into the master array, M. For example, to dispatch behavior β with D as the dynamic type of the receiver, the shift index for β

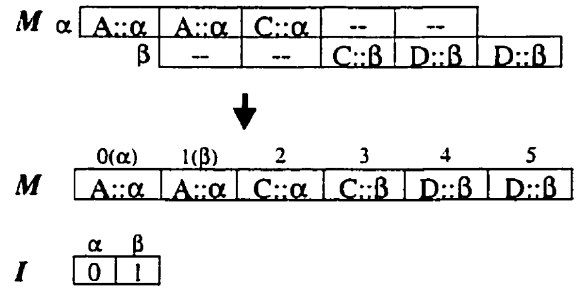


Figure 3.2: Compressing A Selector Table By Row Displacement

is $I[\beta] = 1$. The type number of the receiver, D, is 3. Therefore, the final shift index is $1 + 3 = 4$, and the method to execute is at $M[4]$ which is D::β. Compared with other single-receiver table dispatch techniques, row displacement is highly space and time efficient [21]. I will show how this single-receiver technique can be generalized to multi-method languages in Chapter 4. This is the main research contribution of this thesis.

3.2.3 Selector Coloring (SC)

Selector coloring (SC) compresses the selector table by allowing two rows (behaviors) to be combined, if no type recognizes both behaviors in the type hierarchy [13, 3]. For example, if Figure 2.1(a) was modified, so that no method $C :: \beta$ was included, then the selector table for the types and method definitions in the modified Figure 2.1(a) would be as shown on the left hand side of Figure 3.3. Since no type understands both α and β, the two rows in the selector table can be combined, as shown on the right hand side of Figure 3.3. Since, both α and β are sharing one row index, a behavior to row index table is added to record the correct row index for each behavior. Since this approach is implementable as a graph coloring algorithm, the selector (behavior) indices are usually referred to as colors.

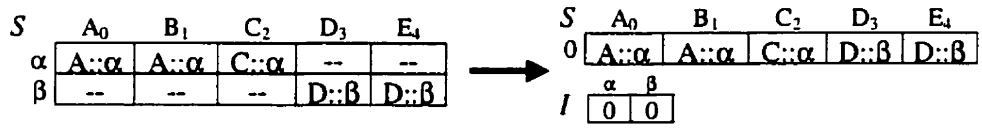


Figure 3.3: Compressing A Selector Table By Selector Coloring

In dispatch, the first index to the 2-dimensional array, S, is from the index ta-

ble I and the second is the type number. For example, to dispatch behavior β with D as the dynamic type of the receiver, the correct method can be found at $S[\ I[\beta]\ ][\ num(D)\ ] = S[0][3] = D :: \beta$. Selector coloring is used in different multi-method dispatch techniques, like Compressed N-Dimensional Tables and Single-Receiver Projections.

In Section 2.4, it was mentioned that in non-statically typed languages, an extra validity check must be made during dispatch due to aliasing during compression. For example, if I dispatch behavior β with dynamic type A, then the compressed table in Figure 3.3 yields $S[\ I[\beta]\ ][\ num(A)\ ] = S[0][0] = A :: \alpha$, which is incorrect. The returned method is not even a method for β. In single receiver dispatch, the validity check simply compares the required behavior, β, to the behavior of the returned method, $A :: \alpha$; since they do not match, there is a *method-not-understood* error.

3.3 Cache-Based Multi-Method Dispatch Techniques

In the cache-based dispatch techniques for single-receiver described in Section 3.1, $< T, \sigma >$ is used as a key to a cache, where T is a type. The same key is used in multi-method caches, except that type T is replaced by a product-type P.

3.4 Table-Based Multi-Method Dispatch Techniques

This section provides a summary of the existing multi-method table dispatch techniques.

3.4.1 N-Dimensional Table

In single-receiver method dispatch, only the dynamic type of the receiver and the behavior name are used in dispatch. However, in multi-method dispatch, the dynamic types of all arguments and the behavior name are used.

The single-receiver dispatch table can be extended to a multi-method table. In multi-method dispatch, each k-arity behavior, $\mathcal{B}^k_\sigma$, has a k-dimensional dispatch table, D^k_σ, with type numbers as indices for each dimension. Therefore, each k-dimensional dispatch table has $|\mathcal{H}|^k$ entries. At a call-site, $\sigma(o_1, o_2, ..., o_k)$, the method to execute is in $D^k_\sigma[num(T^1)][num(T^2)]...[num(T^k)]$, where $T^i = type(o_i)$.

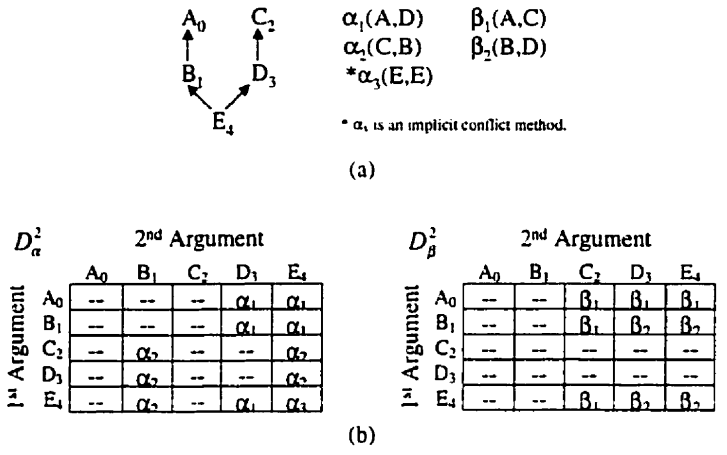


Figure 3.4: N-Dimensional Dispatch Tables

For example, the 2-dimensional dispatch tables for the type hierarchy and method definitions in Figure 3.4(a) are shown in Figure 3.4(b). In building an n-dimensional dispatch table, inheritance conflicts must be resolved. For example, there is an inheritance conflict at $E \times E$ for α, since both α_1 and α_2 are applicable for the call-site $\alpha(anE, anE)$. Therefore, an implicit conflict method α_3 is defined, and inserted into the table at $E \times E$.

N-dimensional table dispatch is very time efficient. However, analogous to the situation with selector tables in single-receiver languages, n-dimensional dispatch tables are impractical because of their huge memory requirements. For example, in the type hierarchy for the Cecil Vortex3 compiler program, there are 1954 types. Therefore, a single 3-arity behavior would require $1954^3 \times 4$ bytes $= 29.84$ gigabytes. Since there are hundreds of different behaviors, the space requirement is prohibitive. The need to compress these n-dimensional tables is even greater than the need to compress single-receiver dispatch tables.

3.4.2 Compressed N-Dimensional Table (CNT)

The Compressed N-Dimensional Table (CNT) [2] technique keeps one k-dimensional dispatch table, $D_\sigma^{k,CNT}$ per behavior, where k represents the arity of a behavior. Starting from a regular n-dimensional dispatch table as described in Section 3.4.1, CNT eliminates rows or columns containing only empty entries. For example, applying this

elimination to the n-dimensional tables D^2_α and D^2_β in Figure 3.4(b) yields the tables shown in Figure 3.5(a). Then, CNT groups identical rows or columns together. This grouping technique is called class sharing. The result of applying class sharing to the tables in Figure 3.5(a) is shown in Figure 3.5(b).

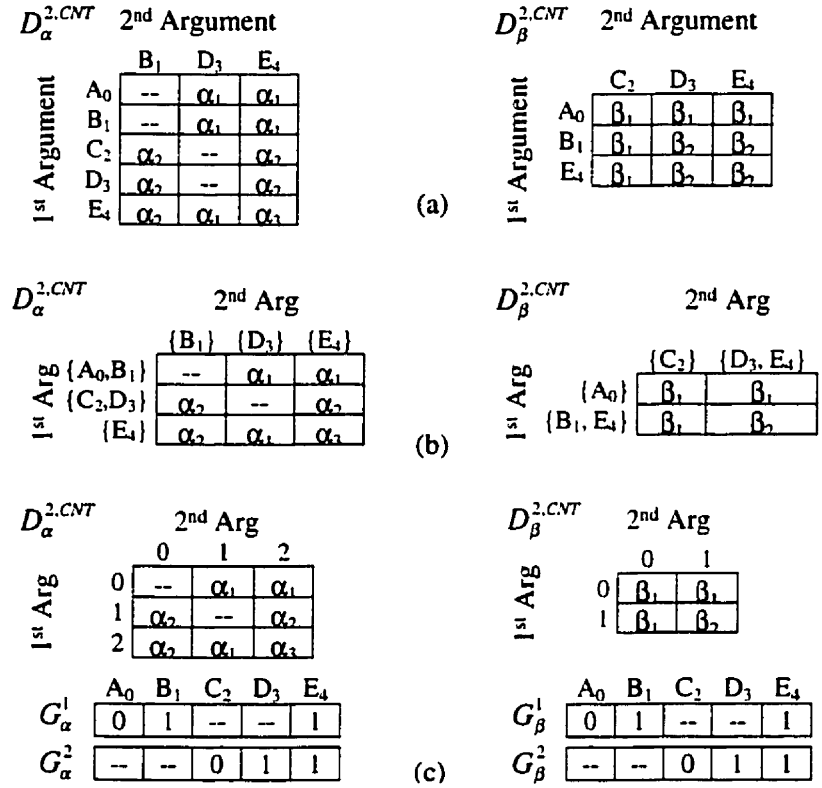


Figure 3.5: Compressed N-Dimensional Table

The groups of types indexing the dimensions of a compressed table are called index-groups. In each dimension, index groups are represented by an index. For example, in the first dimension of the table $D^{2,CNT}_\alpha$ in Figure 3.5(b), the index of the index of group $\{A, B\}$ is 0; $\{C, D\}$ is 1; and $\{E\}$ is 2. However, after this grouping, CNT cannot use type-indices to access the dispatch table, $D^{k,CNT}_\sigma$ directly. Therefore, k type to index arrays, G^1_σ,...,G^k_σ, are created to map each type to its corresponding index-group index in its own dimension. Figure 3.5(c) shows the CNT dispatch table for α and β with the corresponding type to index arrays. These type to index arrays are, then, compressed by row displacement or selector coloring. Expression 3.1 shows the dispatch formula for CNT.

$$D_{\sigma}^{k,CNT}[\ G_{\sigma}^{1}[T^{1}]\ ][\ G_{\sigma}^{2}[T^{2}]\ ]...[\ G_{\sigma}^{k}[T^{k}]\ ] \tag{3.1}$$

For example, dispatch of the call-site $\beta(\ anE, aD\ )$ using the data structure in Figure 3.5(d) is shown in Expression 3.2.

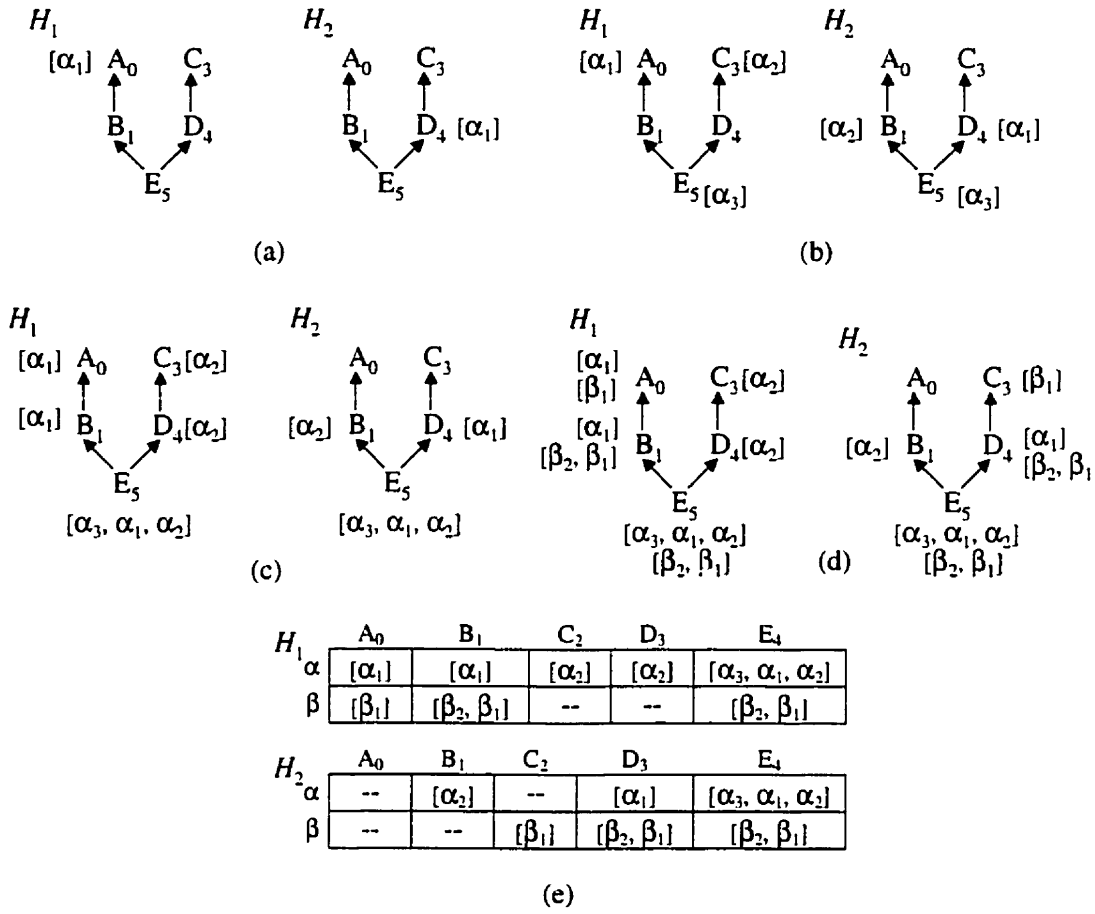
$$\begin{aligned} & D_{\beta}^{2,CNT}[\ G_{\beta}^{1}[\ num(E)\ ]\ ][\ G_{\beta}^{2}[\ num(D)\ ]\ ] \\ & = D_{\beta}^{2.CNT}[\ G_{\beta}^{1}[4]\ ][\ G_{\beta}^{2}[3]\ ] \\ & = D_{\beta}^{2,CNT}[1][1] \\ & = \beta_2 \end{aligned} \tag{3.2}$$

3.4.3 Single-Receiver Projections (SRP)

Single-Receiver Projections (SRP) [23] handles a k-arity behavior multi-method dispatch as k single-receiver dispatches. Instead of maintaining one data structure per behavior, SRP maintains K copies of the type hierarchy, $\mathcal{H}$, which are denoted as $\mathcal{H}_1$, $\mathcal{H}_2$, ..., $\mathcal{H}_K$, where K is the maximum arity across all behaviors. Then, for each behavior, B_{σ}^{k}, its method definitions are projected onto the first k hierarchies, $\mathcal{H}_1$, $\mathcal{H}_2$, ..., $\mathcal{H}_k$.

For example, the method definition $\alpha_1(A, D)$ is projected to $\mathcal{H}_1$ and $\mathcal{H}_2$. α_1 is projected to $\mathcal{H}_1$ for its first arity type, A, and projected to $\mathcal{H}_2$ for its second arity type, D. The result of projecting $\alpha_1(A, D)$ onto $\mathcal{H}_1$ and $\mathcal{H}_2$ is shown in Figure 3.6(a). The results of projecting the rest of the methods defined in Figure 3.4(a) are shown in Figure 3.6(b).

In the next step, SRP extends each of the hierarchies to return a partially ordered method set (poset), which includes all 'natively' defined and inherited methods for each type. This extension is shown in Figure 3.6(c). Method definitions of all behaviors are also projected to the same set of type-hierarchies, $\mathcal{H}_1$, $\mathcal{H}_2$, ..., $\mathcal{H}_K$, in the same way. The result of projecting method definitions for α and β in Figure 3.4(a) is shown in Figure 3.6(d). Each partially ordered method set must satisfy the constraint: if $dom(\sigma_i) \prec dom(\sigma_j)$, then σ_i must preceed σ_j. For example, the $\mathcal{H}_1$ poset $[\alpha_3, \alpha_1, \alpha_2]$ in Figure 3.6(d) may be replaced by the poset $[\alpha_3, \alpha_2, \alpha_1]$, but α_3 must



H_1

	A_0	B_1	C_2	D_3	E_4
α	$[\alpha_1]$	$[\alpha_1]$	$[\alpha_2]$	$[\alpha_2]$	$[\alpha_3, \alpha_1, \alpha_2]$
β	$[\beta_1]$	$[\beta_2, \beta_1]$	--	--	$[\beta_2, \beta_1]$

H_2

	A_0	B_1	C_2	D_3	E_4
α	--	$[\alpha_2]$	--	$[\alpha_1]$	$[\alpha_3, \alpha_1, \alpha_2]$
β	--	--	$[\beta_1]$	$[\beta_2, \beta_1]$	$[\beta_2, \beta_1]$

(e)

Figure 3.6: Single-Receiver Projections

preceed α_1, since $dom(\alpha_3) = E_4 \prec A_0 = dom(\alpha_1)$, and α_3 must preceed α_2, since $dom(\alpha_3) = E_4 \prec C_2 = dom(\alpha_2)$. The partially ordered sets replace methods in the selector tables as shown in Figure 3.6(e). These partially order method sets can be represented by bit vectors [23].

At dispatch, a partially ordered set is obtained from the corresponding hierarchy for each argument type. These posets are intersected to obtain a result poset. Within the final poset, the first element is the dispatch result. For example, consider the call-site, $\beta(\ aB, aC\ )$. First, the poset $[\beta_2, \beta_1]$ is obtained from $\mathcal{H}_1$ for the first argument aB. Then, the poset $[\beta_1]$ is obtained from $\mathcal{H}_2$ for the second argument aC. Intersecting these two posets yields $[\beta_1]$. Since, β_1 is the first element, β_1 is the dispatch result. The dispatch formula for SRP is shown in Expression 3.3.

$$FirstElement(\mathcal{H}_1[\sigma][T^1] \cap \mathcal{H}_2[\sigma][T^2] \cap ... \cap \mathcal{H}_k[\sigma][T^k]) \quad (3.3)$$

Note that the hierarchies, $\mathcal{H}_1$, $\mathcal{H}_2$, ..., $\mathcal{H}_K$, can be compressed by any single-receiver table compression techniques, described in Section 3.2, to obtain better space utilization. Moreover, several other enhancements have been applied to SRP as described in [23], to improve its time and space efficiency.

3.5 Search-Based Multi-Method Dispatch

The only search-based single-receiver dispatch technique is method lookup. However, a simple extension of method lookup will not work for multi-method dispatch. For example, there are three different methods defined for the behavior γ in Figure 2.2. When the call-site $\gamma(aC, aC)$ is dispatched, and γ is not defined in the product-type $C \times C$, where should the method lookup begin? Should the supertype of the first argument, or the second argument be considered first? Assume that the second argument is considered first, and the method γ_2 is found in product-type $B \times B$. However, in multi-method dispatch method search cannot stop after one of the methods is found. Obviously, γ_1 and γ_4, are valid alternatives. Therefore, the search has to continue until all possibilities are exhausted. After all the applicable methods are found, the methods must be ordered, and the most applicable method must be selected. This process is too complicated to be executed at run-time. Therefore, search-based multi-method dispatch techniques need to do some precomputation to simplify the search.

There are three search-based multi-method dispatch techniques: Lookup Automata (LUA), Efficient Predicate Dispatch (EPD), and Product Type Search (PTS). Lookup Automata is the first published search-based dispatch technique, I will review this technique in detail in the following section. Efficient Predicate Dispatch is an extension of lookup automata, as described in [8]. Product Type Search is the simplest search-based multi-method dispatch technique, but it has not been published yet. I will also describe product type search in detail.

All search-based multi-method dispatch techniques have per-behavior dispatch functions. When a call-site is encountered, the corresponding dispatch function of

that behavior is called to handle the dispatch. The per-behavior dispatch function may access global data structures for information.

3.5.1 Lookup Automata (LUA)

Chen et. al describe Lookup Automata (LUA) in detail [10, 9]. The idea of LUA is every simple, it creates one lookup automaton per behavior. Subtype testing is used in transition from one state to another, until a final state is reached. Each final state represents a method of a behavior. To avoid backtracking, and thus exponential dispatch time, some automaton must include more types than are explicitly listed in method definitions (inheritance conflicts are implicitly defined this way).

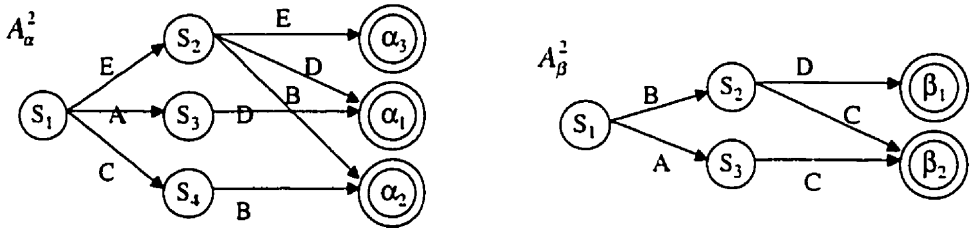


Figure 3.7: Lookup Automata

Figure 3.7 shows the automata for α and β defined in Figure 3.4(a). In each state, the edges leading away from it must be ordered so that a subtype comes before its supertype. In Figure 3.7, the edges are ordered top-down. The automaton for α is translated to the function shown in Figure 3.8.

As shown in Figure 3.8, LUA needs frequent subtype testing. Therefore, an efficient subtype testing mechanism is necessary. The authors of LUA do not specify how subtype testing should be done. Efficient Predicate Dispatch (EPD) extends LUA, so that the subtype testing is done within the per-behavior function by type-number comparing. For that reason, no extra run-time data structure is created for subtype testing.

It can be seen from Figure 3.8 that LUA does not have constant time dispatch. Each dispatch takes at least k comparisons, besides the time for calling the dispatch function itself. The call to the dispatch function involves saving registers and a branch instruction that will clear the instruction pipe-line.

```
void α-automaton ( o1, o2 ) {
    T1 = type( o1 );
    T2 = type( o2 );
    if( T1 ≺ E ) {
        if( T2 ≺ E ) execute α3( o1, o2 );
        else if( T2 ≺ D ) execute α1( o1, o2 );
        else if( T2 ≺ B ) execute α2( o1, o2 );
        else execute method-not-understood;
    }
    else if( T1 ≺ A ) {
        if( T2 ≺ D ) execute α1( o1, o2 );
        else execute method-not-understood;
    }
    else if( T1 ≺ C ) {
        if( T2 ≺ B ) execute α2( o1, o2 );
        else execute method-not-understood;
    }
    else {
        execute method-not-understood;
    }
}
```

Figure 3.8: The LUA dispatch function for α

Since type checking at each stage has to be in subtype order, LUA cannot take advantage of call-site profiles. For example, consider 1,000 call-sites for the behavior $\mathcal{B}^2_\alpha$, where 800 of them are $\alpha(\ aC, aB\ )$. According to Figure 3.8, each of the 800 call-sites has to perform the subtype tests $if(T^1 \prec E)$ and $if(T^1 \prec A)$ before performing the test $if(T^1 \prec C)$, which is the right one. Since the type E is a subtype of the type C, the test $if(T^1 \prec C)$ cannot go before $if(T^1 \prec E)$, even though $if(T^1 \prec C)$ is used more often. The EPD technique extends LUA to take advantage of call-site profiles.

3.5.2 Product Type Search (PTS)

Product Type Search (PTS) is very similar to LUA. Instead of using subtype testing per argument, PTS uses child product type testing. First, all conflict methods are implicitly defined. Then, the product type of each implicitly or explicitly defined

method is retrieved. These product types are ordered so that each child product type preceeds its parents. Finally, a per-behavior function is created to perform child product type testing as ordered. Figure 3.9 shows the per-behavior function for α in PTS.

```
void α-product-type-search( o1, o2 ) {
    Product-Type P = type(o1) × type(o2 );
    if( P ≺ E × E ) execute α3( o1, o2 );
    else if( P ≺ A × D ) execute α1( o1, o2 );
    else if( P ≺ C × B ) execute α2( o1, o2 );
    else execute method-not-understood;
}
```

Figure 3.9: The PTS dispatch function for α

Chapter 4

Multiple Row Displacement (MRD)

4.1 Multiple Row Displacement by Examples

Multiple Row Displacement (MRD) is a new time and space efficient dispatch technique which combines row displacement and n-dimensional dispatch tables. MRD will first be illustrated by examples, and then the algorithm will be given. The first example uses the type hierarchy and 2-arity method definitions from Figure 3.4(a).

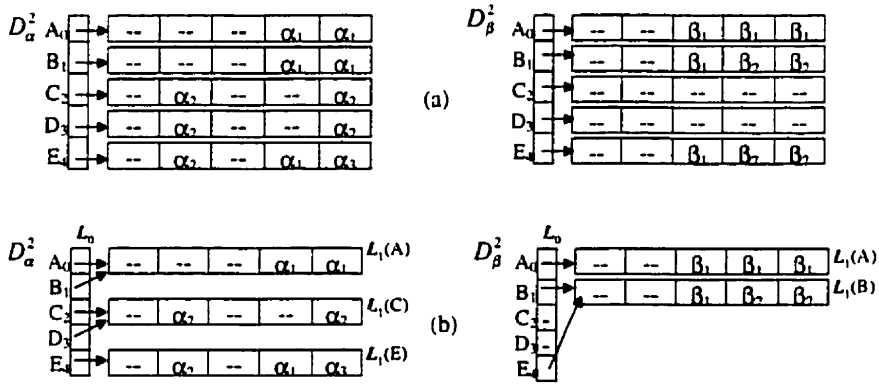


Figure 4.1: Data Structure for Multiple Row Displacement

Instead of representing each dispatch table as a single k-dimensional array as shown in Figure 3.4(b), each table can be represented as an array of arrays as shown in Figure 4.1(a). The arrays indexed by the first argument are called *level*-0 arrays, L_0. There is only one *level*-0 array per behavior. The arrays indexed by the second argument are called *level*-1 arrays, $L_1(\cdot)$. If the arity of the behavior is greater than

two then the arrays indexed by the third arguments are called *level*-2 arrays, $L_2(\cdot)$; and so on. The highest level arrays are *level*-$(k-1)$ arrays, $L_{k-1}(\cdot)$, for k arity behaviors.

It can be seen from Figure 4.1(a) that some of the *level*-1 arrays are exactly the same. The common arrays are combined as shown in Figure 4.1(b). In general, there will be many identical rows in an n-dimensional dispatch table, and many empty rows. These observations are the basis for the CNT dispatch technique mentioned in Chapter 3, and are also one of the underlying reasons for the compression provided by MRD. It is worth noting that this sharing of rows is only possible due to the fact the table uses types to index into all dimensions. In single-receiver languages, the tables being compressed have behaviors along one dimension, and types along the other. Sharing between two behavior rows would imply that both behaviors invoke the same methods for all types, and although languages like Tigukat [27] allow this to happen, such a situation would be highly unlikely to occur in practice. Sharing between two type columns is also unlikely since it occurs only when a type inherits methods from a parent and does not redefine or introduce any new methods. Such sharing of type columns is more feasible if the table is partitioned into subtables by grouping a number of rows together. This strategy was used in the single-receiver dispatch technique called Compressed Dispatch Table (CT) [29].

There is one data structure per behavior, D_α^k, and MRD compresses these per behavior data structures by row displacement into three global data structures: a Global Master Array, M, a set of Global Index Arrays, I_j, where $j = 0, ..., (K-2)$, and a Global Behavior Array, B.

In compressing the data structure D_α^2 in Figure 4.1(b), the *level*-1 array $L_1(A)$ is first shifted into the Global Master Array, M, by row displacement, as shown in Figure 4.2(a). The shift index, 0, is stored in the *level*-0 array, L_0, in place of $L_1(A)$ (and into L_0 at B, since A and B share $L_1(A)$). In the implementation, a temporary array is created to store the shift indices, but in this thesis, I have put them in L_0 for simplicity of presentation. Figure 4.2(b) shows how $L_1(C)$ and $L_1(E)$ are shifted into M by row displacement, and how they are replaced in L_0 by their shift indices 1 and 5. Finally, as shown in Figure 4.2(c), L_0 is shifted into the Global Index Array, I_0 by row displacement. The resulting shift index, 0, is stored in the Global Behavior

Array at $B[\alpha]$. After D^2_α is compressed into the global data structures, the memory for its preliminary data structures can be released. Figure 4.3 shows how to compress the behavior data structure, D^2_β, into the same global data structures, M, I_0 and B. The compression of the *level*-1 arrays, $L_1(A)$ and $L_1(B)$, are shown in Figure 4.3(a). The compression of the *level*-0 array, L_0, is shown in Figure 4.3(b). Note that only I_0 is used in the case of arity-2 behaviors. For arity-3 behaviors, I_1 will also be used. For arity-4 behaviors, I_2 will also be used, etc.

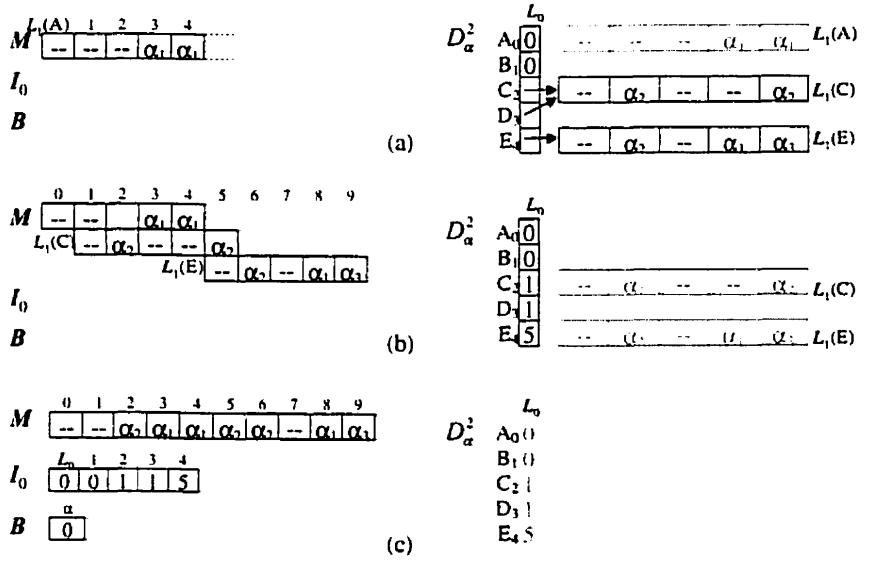


Figure 4.2: Compressing The Data Structure for α

As an example of dispatch, I will demonstrate how to dispatch a call-site $\beta(anE, aD)$ using the data structures in Figure 4.3(b). The method dispatch starts by obtaining the shift index of the behavior, β, from the Global Behavior Array, B. From Figure 4.3(b), $B[\beta]$ is 5. The next step is to obtain the shift index for the first argument, E, from the Global Index array, I_0. Since the shift index of β is 5, and the type number of E, num(E), is 4, the shift index of the first argument is $I_0[5+4] = I_0[9] = 11$. Finally, by adding the shift index of the first argument to the type number of the second argument, $num(D) = 3$, an index to M is formed, which is $11 + 3 = 14$. The method to execute can be found in $M[14] = \beta_2$, as expected.

MRD can be extended to handle behaviors of any arity. Figure 4.4(a) shows the method definitions of a 3-arity behavior, δ, and Figure 4.4(b) shows its preliminary

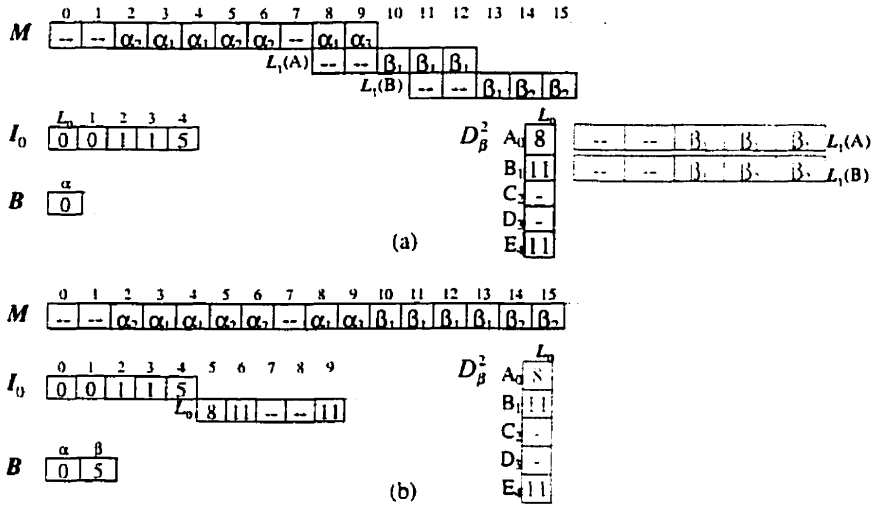


Figure 4.3: Compressing The Data Structure For β, with α in place from Figure 4.2

behavior data structure, D_δ^3. Figures 4.4(c) to 4.4(e) show the compression of this data structure. First, the *level*-2 arrays, $L_2(B \times D)$, $L_2(D \times B)$ and $L_2(E \times E)$ are shifted into the existing M as shown in Figure 4.4(c). Their shift indices (15, 14, 19) are stored in $L_1(B)$, $L_1(D)$ and $L_1(E)$. In fact, every pointer in Figure 4.4(b) that pointed to $L_2(B \times D)$ is replaced by the shift index 15. Pointers to $L_2(D \times B)$ are replaced by the shift index 14 and the single pointer to $L_2(E \times E)$ is replaced by the shift index 19. Then, the *level*-1 arrays, $L_1(B)$, $L_1(D)$ and $L_1(E)$, are shifted into the Global Index Array I_1 as shown in Figure 4.4(d). The shift indices (0,1,5) are stored in L_0. Finally, L_0 is shifted into the Global Index Array I_0 and its shift index (7) is stored in the Global Behavior Array at $B[\delta]$, as shown in Figure 4.4(e).

4.2 The Multiple Row-Displacement Dispatch Algorithm

I have shown, by examples, how MRD compresses an n-dimensional dispatch table by row displacement. On the behavior level, a preliminary data structure, D_σ^k, is created for each behavior. D_σ^k is a data structure for a k-arity behavior named σ, as shown in Figure 4.4(b). It is actually an n-dimensional dispatch table, which is an array of pointers to arrays. Each array in D_σ^k has the size of $|\mathcal{H}|$. The *level*-0 array, L_0, is indexed by the type of the first argument. The *level*-1 arrays, $L_1(\cdot)$, are indexed by

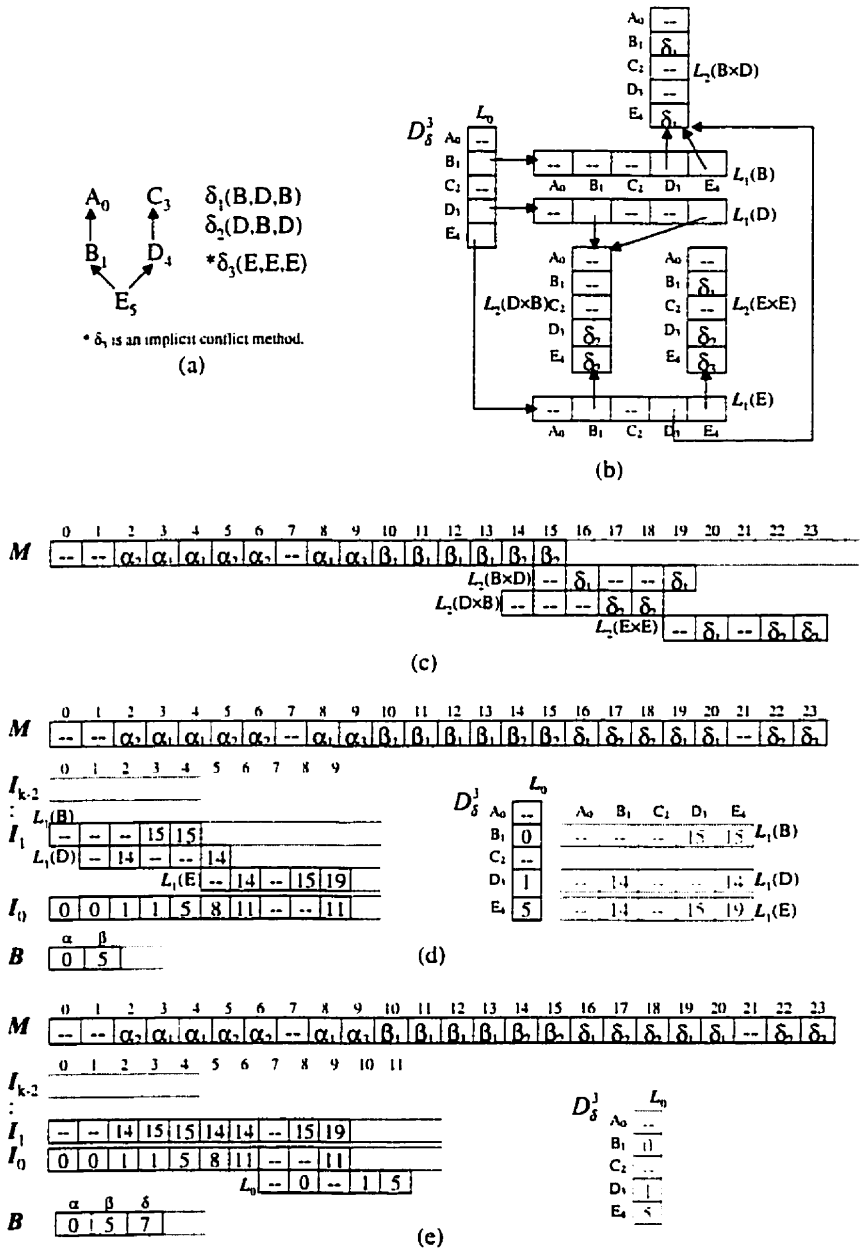


Figure 4.4: Compressing The Data Structure For δ

the type of the second argument. The *level*-$(k-1)$ arrays, $L_{k-1}(\cdot)$, always contain method addresses. All other arrays contain pointers to arrays at the next level.

After the compression has finished, there are a Global Master Dispatch Array, M, $K-1$ Global Index Arrays, I_0, ..., I_{k-2}, and a Global Behavior Array, B. The Global Master Dispatch Array, M, stores method addresses of all methods. Each Global Index Array, I_j, contains shift indexes for I_{j+1}. The Global Behavior Array, B stores

the shift indices of the behaviors.

At compile time, a D_σ^k data structure is created for each behavior. The *level-*$(k-1)$ arrays, L_{k-1}, are shifted into M by row displacement. The shifted indices are stored in L_{k-2}. Then, the *level-*$(k-2)$ arrays, L_{k-2}, are shifted into the index array, I_{k-2}. The shift indices are stored in L_{k-3}. This process is repeated until the *level-*0 array, L_0, is shifted into I_0, and the shift index is stored in $B[\sigma]$. The whole process is repeated for each behavior. The algorithm to compress all behavior data structures is given in Section 4.4.

The dispatch formula for a call-site, $\sigma(o_1, ..., o_k)$, is given by Expression 4.1, where $T^i = type(o_i)$.

$$\begin{aligned} &M[\ I_{k-2}[\ I_{k-3}[\ ...\ I_1[\ I_0[\ B[\ \sigma\ ] + num(T^1)\ ] \\ &\quad + num(T^2)\ ] + ...\ ] + num(T^{k-2})\ ] + num(T^{k-1})\ ] + num(T^k)\ ] \end{aligned} \tag{4.1}$$

As an example of dispatch with Expression 4.1, I will demonstrate how to dispatch a call-site $\delta(anE, aD, aB)$ using the data structures in Figure 4.4(e). Since δ is a 3-arity behavior, Expression 4.1 becomes Expression 4.2.

$$M[\ I_1[\ I_0[\ B[\ \delta\ ] + num(E)\ ] + num(D)\ ] + num(B)\ ] \tag{4.2}$$

Substituting the data from Figure 4.4(e) into Expression 4.2 yields the method δ_1, as shown in Expression 4.3.

$$\begin{aligned} &M[\ I_1[\ I_0[\ 7+4\ ]+3\ ]+1\ ] \\ &= M[\ I_1[\ I_0[\ 11\ ]+3\ ]+1\ ] \\ &= M[\ I_1[\ 5+3\ ]+1\ ] \\ &= M[\ I_1[\ 8]+1\ ] \\ &= M[\ 15+1\ ] \\ &= M[\ 16\ ] = \delta_1 \end{aligned} \tag{4.3}$$

Note that all index arrays, I_0, I_1, I_2, ..., can be further compressed into one big index array by row displacement to save more memory. However, for presentation simplicity I have ignored this final compression.

4.3 Improvements

4.3.1 Eliminating the Global Behavior Array

Each behavior has its own data structure to store information unrelated to dispatch. A field named *shift_index* can be added to this behavior data structure to support MRD. Then, the shift index of each behavior can be stored in the data structure of each behavior, instead of a Global Behavior Array, B. The advantage of this change is that one array lookup is eliminated from method dispatch, without increasing the memory usage. The new dispatch formula for a call-site, $\sigma(o_1, ..., o_k)$, after the change is given in Expression 4.4.

$$M[\, I_{k-2}[\, I_{k-3}[\, ... \, I_1[\; I_0[\; \sigma.shift\_index + num(T^1)\,] + num(T^2)\,] + ...\,] + num(T^{k-2})\,] + num(T^{k-1})\,] + num(T^k)\,] \qquad (4.4)$$

Note that in a non-reflexive environment, the shift_index is a compile-time constant that can be inserted into the dispatch code.

4.3.2 Use a Single Global Index Array

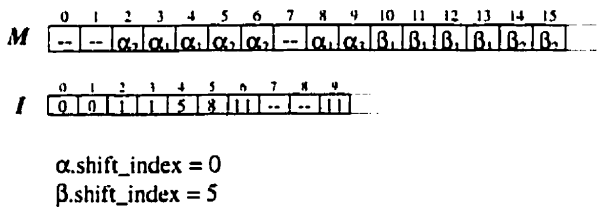


Figure 4.5: Global Data Structure With One Index Array

For simplicity of presentation, Section 4.1 and Section 4.2 had one Index Array per arity position. Actually, only one Global Index Array, I, is needed to store all *level*-0 to *level*-$(k-2)$ arrays. Figure 4.5 shows the global data structure, after D^2_α and D^2_β from Figure 4.1(b) has been compressed using a single Global Index Array. Since α and β are both 2-arity behaviors, they use only one index array, the Index Array I_0 in Figure 4.3(b). This index array has been re-named I in Figure 4.5. The effects of using one index array are illustrated when D^3_δ from Figure 4.4(b) is compressed into the global data structure. Figure 4.6(a) shows the global data structure after

$L_2(B \times D)$, $L_2(D \times B)$ and $L_2(E \times E)$ have been compressed into the Global Master Array, M. Then, Figures 4.6b and 4.6c shows how to compress $L_1(B)$, $L_1(D)$ $L_1(E)$, and L_0 into the single Global Index Array.

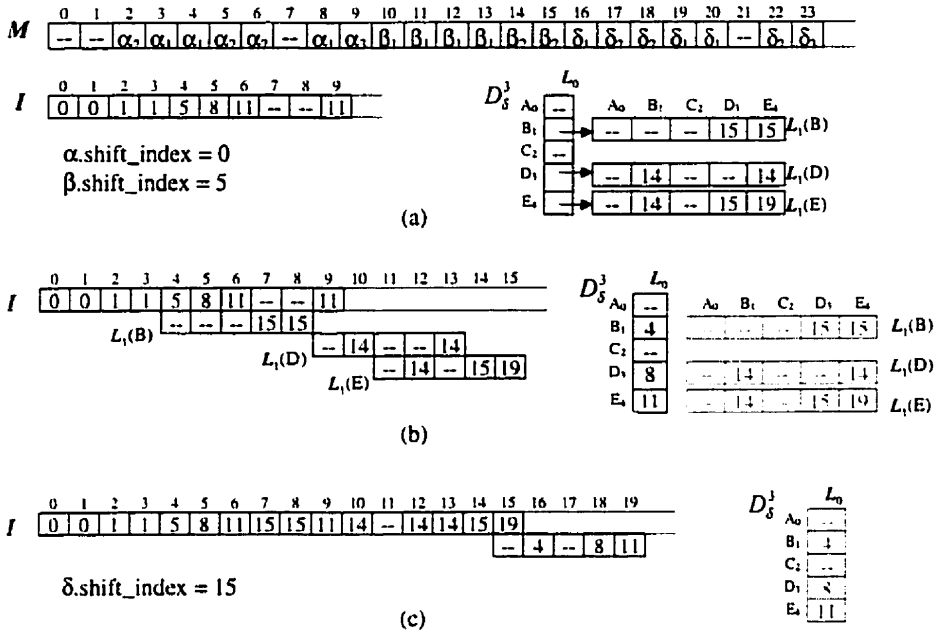


Figure 4.6: Global Data Structure With One Index Array

Using a single Index Array provides additional compression, and has no negative impact on dispatch speed. Notice that this change has simplified the global data structure. Now, only 2 arrays are maintained. Expression 4.6 shows the modified dispatch formula that accesses one Global Index Array. As an example, the formula to dispatch the call-site, $\delta(anE, aD, aB)$ using Expression 4.5 is shown in Expression 4.6. Substituting the data from Figure 4.6 to Expression 4.6 yields method δ_1, as shown in Expression 4.7. This is the same result that was derived in Section 4.2.

$$M[\, I[\, I[\, \ldots\, I_1[\, I[\, \sigma.shift_index + num(T^1)\,] + num(T^2)\,] + \ldots\,] + num(T^{k-2})\,] + num(T^{k-1})\,] + num(T^k)\,] \quad (4.5)$$

$$M[\, I[\, I[\, \delta.shift_index + num(E)\,] + num(D)\,] + num(B)\,] \quad (4.6)$$

$$
\begin{aligned}
& M[\, I[\, I[\, 15+4\,]+3\,]+1\,] \\
&= M[\, I[\, I[\, 19\,]+3\,]+1\,] \\
&= M[\, I[\, 11+3\,]+1\,] \\
&= M[\, I[\, 14]+1\,] \\
&= M[\, 15+1\,] \\
&= M[\, 16\,] = \delta_1
\end{aligned}
\tag{4.7}
$$

This improvement simplifies the data structure, and reduces total memory usage, especially memory for the high arity position Index Array. For example, if there is only one 10-arity behavior in the environment, for this one behavior I_8 of size $|\mathcal{H}|$ has to be maintained, even though I_8 is a sparse array. According to [14] and [16], Row Displacement is highly space efficient in compressing sparse short arrays. Therefore, compressing I_8 into the single Global Index Array, I, reduces the overall memory usage. The same reason applies, when I_7, I_6 and I_5 are collapsed into I.

4.3.3 Row Matching

Note that the row-shifting mechanism used in my implementation of row displacement is not the most space-efficient technique possible. When the row-shifting algorithm is replaced by a more general algorithm called *row-matching* (based on string-matching), a higher compression rate is obtained. In row-matching, two table entries match if one entry is empty or if both entries are identical. For example, using row-shifting to compress rows R1 and R2 in Figure 4.7(a) produces a master array with 9 elements as shown in Figure 4.7(b). However, using a row-matching algorithm to compress R1 and R2 produces a master array with only 6 elements as shown in Figure 4.7(c). Using row-matching instead of row-shifting provides an additional 10-14% compression. Row-matching cannot be used in single-receiver row displacement, since different rows contain different behaviors, and thus different addresses.

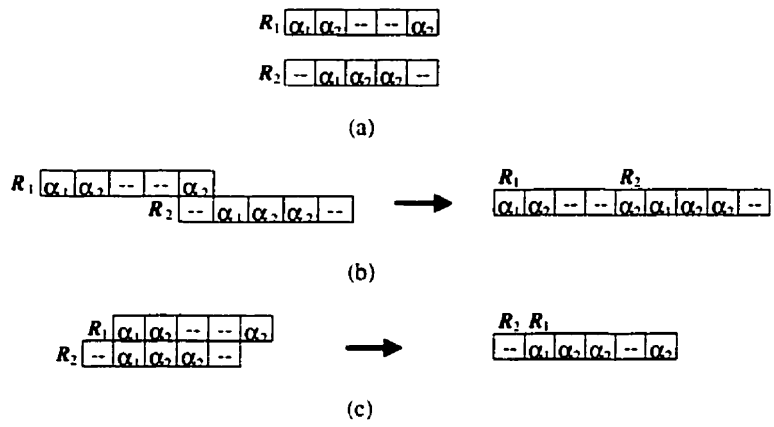


Figure 4.7: Row-Shifting vs. Row-Matching

4.3.4 Byte vs. Word Storage (MRD-B)

MRD stores four-byte function addresses in M. In a large hierarchy, M is the largest data structure. To reduce the size of M, a method-map, $D_\sigma^{k,MRD}$, can be introduced for each behavior. Since all methods of a behavior are stored in $D_\sigma^{k,MRD}$, a method can be represented by an index into $D_\sigma^{k,MRD}$. Since it is very unlikely that more than 256 methods are defined per behavior, only one byte is needed to store the index to the corresponding $D_\sigma^{k,MRD}$. If M stores this index instead of the function address, the size of M is reduced to one-forth of its original size. However, there is an extra indirection to access the method-map at dispatch time. I denote the technique which stores bytes instead of words by MRD-B.

4.3.5 Type Ordering

In single receiver row displacement, type ordering has a significant impact on compression ratios [14]. I have investigated type ordering in multi-method row displacement and found that the impact is not as significant, since the fill-rate for both Global Master Array and Global Index Array is higher than 95%.

4.4 The MRD Data Structure Creation Algorithm

The algorithm to build the global data structure for MRD is given below:

```
Array M, I;
createGlobalDataStructure() begin
    for(each behavior B_σ^k ) do
        BehaviorStructure D_σ^k = B_σ^k.createStructure();
        createRecursiveStructure( D_σ^k.L_0, 0 );
        B_σ^k.shiftIndex = D_σ^k.L_0.getShiftIndex();
    endfor
end

createRecursiveStructure( Array L, int level ) begin
    for(int i=0; i<L.size(); i++ ) do
        if( L[i] == null ) then
            continue;
        elseif( L[i].getShiftIndex() == -1 ) then
            if( level == k-2 ) then
                L[i] = M.add( L[i] );
            else
                createRecursiveStructure(L[i],level+1);
                L[i] = L[i].getShiftIndex();
            endif
        else
            L[i] = L[i].getShiftIndex();
        endif
    endfor
    I.add( L );
end
```

This algorithm uses three support routines: `Array.add(Array)`, `Array.getShiftIndex()`, and `Behavior.createStructure()`. The `Array.add(Array)` function shifts the given array into the current array by row-matching or row-shifting, and returns the shift index. The returned shift index is also stored in the given array. The `Array.getShiftIndex()` function returns

the shift index of the current array, which is stored in the current array when it is added to another array. If the current array has never been added to another array, this function returns -1. The `Behavior.createStructure()` function creates an n-dimensional table for the current behavior.

4.5 Separate Compilation

With table-based dispatch, the tables must be built before they can be used. If a language does not support separate compilation, then the tables can be built at compile-time when the entire type hierarchy and all the method definitions are compiled. If a language supports separate compilation, then neither the complete type hierarchy nor the set of all method definitions for a particular behavior are available when a class is being compiled. In this case, the dispatch tables must be built at link-time. Fortunately, these tables only take a few seconds to build. In addition to building the dispatch tables, call-sites in compiled code must be patched with base table start addresses and global behavior shift indices. However, this is no more difficult than resolving other external references in separately compiled object files.

4.6 Non-Static Typing in MRD

As discuss in Section 2.4 and Section 3.2.3, dispatch techniques that alias different selectors during compression may return a wrong method for invalid call-sites in non-statically typed languages. Actually, there are two potential errors when applying MRD to non-statically typed languages: *index out of bounds*, and *wrong method*. Two examples illustrate the errors.

Assume that the call-site $\delta(aC, anA, aB)$ is dispatched using the data structures in Figure 4.4(e). The formula to dispatch this call-site is shown in Expression 4.8. From Figure 4.4(e), $B[\delta]$ is equal to 7, and $I_0[\ B[\delta] + num(C)\ ] = I_0[\ 7 + 2\ ] = I_0[\ 9\ ]$ which is equal to 11. The next step is to find $I_1[\ 11 + num(A)\ ]$, which is $I_1[\ 11\ ]$. Unfortunately, $I_1[\ 11\ ]$ does not exist, since the index 11 is out of the bounds of the Index Array I_1. This is an example of an index out of bounds error.

$$M[\ I_1[\ I_0[\ B[\ \delta\ ] + num(C)\ ] + num(A)\ ] + num(B)\ ] \tag{4.8}$$

Consider the call-site $\delta(aB, aC, anA)$ using the data structure in Figure 4.4(e). The steps of the dispatch are shown in Expression 4.9. The returned method is β_2, which is not a method defined for δ. This is an example of a wrong method error.

$$
\begin{aligned}
& M[\, I_1[\, I_0[\, B[\, \delta \,] + num(B) \,] + num(C) \,] + num(A) \,] \\
&= M[\, I_1[\, I_0[\, 7+1 \,] + 2 \,] + 0 \,] \\
&= M[\, I_1[\, I_0[\, 8 \,] + 2 \,] + 0 \,] \\
&= M[\, I_1[\, 0+2 \,] + 0 \,] \\
&= M[\, I_1[\, 2] + 0 \,] \\
&= M[\, 14 + 0 \,] \\
&= M[\, 14 \,] = \beta_2
\end{aligned}
\tag{4.9}
$$

4.6.1 Eliminating the Index Out Of Bounds Error

There are two ways to eliminate the index out of bounds error. The first approach is to compare the index against the length of the array before each Index Array access. If the index is bigger than the length of the Index Array, return *method-not-understood*, otherwise, progress to the next step. This solution slows down dispatch, because of the k extra testings.

The second approach extends the improvement described in Section 4.3.2, using a single index array. To eliminate the out of bounds error, Index Array is extended to be at least as long as the Global Master Array, and all the empty-entries in the Index Array are replaced with the number 0. The result of doing such an extension is shown in Figure 4.8. After this extension, if the dispatch formula hits one of the empty-entries, 0 will be used as the index, and $0 + num(T)$ will never be an out of bounds error. If the dispatch formula hits an index of another behavior, that index will never exceed $|M| - |\mathcal{H}|$. Then, $(|M| - |\mathcal{H}|) + num(T)$ will never be an out of bounds, since the Index Array is at least as large as the Master Array.

The second solution solves the out of bounds problem, without decreasing the dispatch speed. However, in general I do not know the size of I compared with M, so I do not know how space inefficient this extension is. In a non-reflexive programming

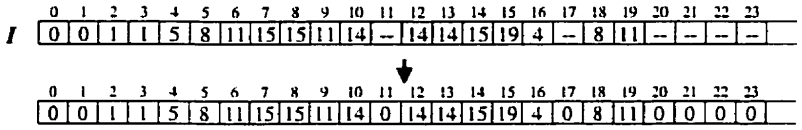


Figure 4.8: Extend the Index Array

environment, where speed is more important than memory, the second solution is the winner. However, in a reflexive environment, the first solution may be a better choice, since once all empty-entries are replaced by 0, it is impossible to insert other arrays into the Index Array. Recall that the index out of bound error only occurs in non-statically typed languages, not statically typed ones. Unfortunately, neither solutions solves the wrong method error.

4.6.2 Eliminating the Wrong Method Error

There are also two ways to eliminate wrong method errors. The first approach is to attach a behavior indicator to each index in the Index Array. The behavior indicator can be a number representing the behavior. This change has been applied to part of the Index Array in Figure 4.8, and the result is shown in Figure 4.9, where *mnu* represents *method-not-understood.* At dispatch, the attached behavior indicator is compared against the behavior of the call-site, after each index is retrieved. If they match, the dispatch algorithm continues to the next step, otherwise, it returns *method-not-understood.* This solution slows down dispatch by k extra testings. In addition, memory usage for the Index Array is doubled.

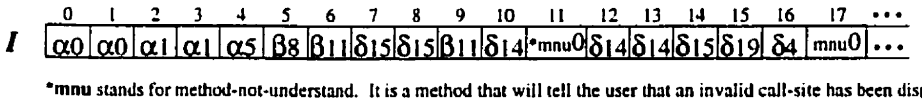


Figure 4.9: Attaching Behavior Indicator to Indices in the Index Array

The second approach is to delay all checking until the end of the dispatch process. In order to do this, all empty-entries in the Global Master Array should be replaced by *method-not-understood.* If a call-site is invalid, either a wrong method, or *method-not-understood* will be returned under the original MRD algorithm. If the result method is *method-not-understood*, return *method-not-understood* as the dispatch

result. Otherwise, check whether the return method matches the behavior of the current call-site. If not, return *method-not-understood.* If so, make sure that the type of each argument is either the defined-type or its subtype for each arity position. If not, return *method-not-understood.* If so, the method address is correct.

Chapter 5

Implementation of Multiple Row Displacement

MRD is implemented in C++, with classes being defined to represent the critical object-oriented concepts affecting dispatch. In particular, a Behavior class represents information about methods sharing the same name and arity; a Type class represents types; and instances of different Table classes store precomputed dispatch results. Instances of a Table Entry class represents elements stored in tables. The following sections provide the implementation details for the MRD algorithm described in the last chapter.

5.1 Behavior

Each Behavior instance describes a particular behavior, where a behavior is defined as the set of methods sharing the same name and having the same arity. Each Behavior instance consists of the following fields.

1. *A name.* The name of the behavior. This name is necessary in eliminating the wrong method error.

2. *An arity.* The number of arguments the behavior needs. Multiple references to the same behavior name with the same number of arity should refer to the same behavior, so a mapping from behavior name and number of arity to behavior instance is maintained.

3. *A number uniquely identifying the behavior.* An array of all behaviors is also

maintained, and the behavior at index n has number n. In reflexive environment, this number is used as an index to access the Global Behavior Array in dispatch.

4. *A behavior shift index.* In non-reflexive environment, this shift index can be pre-computed and stored at compile-time for the use of method dispatch.

5. *An array of product-type/method pairs.* This array lists all explicitly (user-defined) and implicitly (inheritance conflict) defined methods of the behavior. MRD uses the list to construct its dispatch tables, but MRD does not refer to the list during dispatch. The order of the list does not matter in MRD.

5.2 Type

Each type instance records the dispatch-related information about a single type in the type hierarchy for which dispatch is being implemented. Each instance has a name, a unique number, an array of supertypes, an array of subtypes, and a bit vector for subtype testing. Each Type instance has the following fields.

1. *A name.* Similar to behaviors, reference to types of the same name should refer to the same type (i.e. type name uniquely identify type instances)

2. *A number uniquely identifying the type.* This number is used as an index to access dispatch tables, and for subtype testing.

3. *An array of supertypes.* This array lists only immediate supertypes. MRD uses this array to traverse the type hierarchy during dispatch table construction.

4. *An array of subtypes.* This array lists only immediate subtypes. MRD uses this array to traverse the type hierarchy during dispatch table construction.

5. *A bit vector for subtype testing.* This bit vector has the size of the total number of types in the hierarchy. If type T is a subtype (directly or indirectly) of a type instance, the bit at the index representing T will be set.

5.3 Table

The last critical class is Table, which is used to encapsulate the data and functionality for maintaining a row displacement dispatch table. Each instance of Table stores multiple instances of Table Entry. Instances of Table can also compress other instances of Table into themselves. After each compression, the shift index is returned. Whether row matching will be applied in the row displacement process is determined by the instances of Table Entry in a table, which will be discussed in the next section.

The Table class is also responsible for table access functionality. Given a shift index and a type, the Table class returns the instance of Table Entry at the proper position. An instance of Table has the following fields.

1. *A master array.* This master array stores instance of Table Entry.

2. *A list of empty entries.* This list is used in table maintenance. In non-reflexive environment, this list can be deleted after the table has been computed.

5.4 Table Entry

Each instance of Table Entry is either empty or non-empty. Each non-empty entry stores either a method address, (in the Global Master Array), or a shift index (in the Global Index Array). The Table Entry class implements two functions: *isEmpty()* and *isEqual(TableEntry)*. The Table class uses these two functions to perform row displacement. The Table class allows two instances of Table Entry to share a space, if one of the entries is empty, or the entries are equal.

If row-shifting is used, there are only two sets of Table Entry: empty and non-empty. A non-empty entry can only share space with an empty entry. Therefore, the *isEqual(TableEntry)* function return false, unless both of the operands are empty. To use row matching, the *isEqual(TableEntry)* function is altered to return true, if the content of the entries are equal. Each instance of Table Entry consists of the following fields.

1. *A field.* If the entry is not empty, the field stores either a method address or a shift index.

Chapter 6

Performance Results

This chapter presents memory and timing results for the new technique, MRD, and three other techniques, CNT, LUA and SRP. When analyzing dispatch techniques, both execution timing and memory usage need to be addressed. A technique that is extremely fast is still not viable if it uses excessive memory, and a technique that uses very little memory is not desirable if it dispatches methods very slowly. Both timing and memory results are presented for MRD, SRP, LUA and CNT. This is the first time a comparison of multi-method techniques has appeared in the literature.

The rest of this section is organized into three subsections. The first subsection discusses the data-structures and dispatch code required by the various techniques. The second subsection presents timing results. The third subsection presents memory results.

6.1 Data Structures and Dispatch Code

This section provides a brief description of the required data-structures for each of the four dispatch techniques in a static context. The code that needs to be generated at each call-site is also presented. In this chapter, the code presented refers to the code that would be generated by the compiler upon encountering the call-site $\sigma(\ o_1,\ o_2,\ ...,\ o_k\ )$.

The notation $N(o_i)$ represents the code necessary to obtain a type number for the object at argument position i of the call-site. Naturally, different languages implement the relation between object and type in different ways, and dispatch is affected by this choice. My timing results are based on an implementation in which every object is a

pointer to a structure that contains a 'typeNumber' field (in addition to its instance data).

6.1.1 MRD

MRD has an M array that stores function addresses, an I array that stores level-array shift indices, and a B array that stores behavior shift indices.

The dispatch sequence is given in Expression 6.1.

$$\begin{aligned}&(*(M[\ I[\ ...I[\ I[\#b^{\sigma} + N(o_1)\ ] \\ &\quad +N(o_2)\ ] + ...\ ] + N(o_{k-1})\ ] + N(o_k)\ ])\ )(o_1, o_2, ..., o_k)\end{aligned} \tag{6.1}$$

Note that the Global Behavior Array, B, from Expression 4.1, is known at compile-time, so $B[\sigma]$ is known at compile-time. Thus $\#b^{\sigma}$ is a literal integer obtained from $B[\sigma]$. The sequence, $M[...]$, in Expression 6.1 returns the address of the method to be executed. Therefore, $*(M[...])$ returns the method to be executed. The method is executed by passing the parameters, $(o_1, o_2, ..., o_k)$, to the method $*(M[...])$. This $(*(...))(o_1, o_2, ..., o_k)$ format is used to indicated method execution in the rest of this chapter.

6.1.2 MRD-B

The dispatch sequence for MRD-B is given in Expression 6.2.

$$\begin{aligned}&(*(D_{\sigma}^{k,MRD}[\ M[\ I[\ ...I[\ I[\ \#b^{\sigma} + N(o_1)\ ] \\ &\quad +N(o_2)\ ] + ...\ ] + N(o_{k-1})\ ] + N(o_k)\ ]\ ])\ )(o_1, o_2, ..., o_k)\end{aligned} \tag{6.2}$$

6.1.3 CNT

For each behavior, CNT has a k-dimensional array, but since I am assuming a non-reflexive environment, this k-dimensional array can be linearized into a one-dimensional array. Indexing into the array requires a sequence of multiplications and additions to convert the k indices into a single index. For a particular behavior, its one-dimensional dispatch table is denoted by $D_{\sigma}^{k,CNT}$.

In addition to the behavior-specific information, CNT requires arrays that map types to type-groups. In [17], these group arrays are compressed by selector coloring (SC). My dispatch results are based on such a compression scheme, and assume that the maximum number of groups is less than 256, so that the group array can be an array of bytes. Furthermore, since the compiler knows exactly which group array to use for a particular type, it is more efficient to declare n statically allocated arrays than it is to declare an array of arrays. Thus, I assume that there are arrays G_1, ..., G_n, and that the compiler knows which group array to use for each dimension of a particular behavior.

If I assume that the compressed n-dimensional table for k-arity behavior σ has dimensions $n_1^\sigma, n_2^\sigma, ..., n_k^\sigma$, where the n_i^σ values are behavior specific, and that the group arrays for these dimensions are $G_1^\sigma, G_2^\sigma, ..., G_k^\sigma$ then the call-site dispatch code is given in Expression 6.3.

$$
\begin{aligned}
(*\ (D_\sigma^{k,CNT}[\quad & G_1^\sigma[N(o_1)] \times \#(n_1^\sigma \times n_2^\sigma \times ... \times n_{k-1}^\sigma) \\
& + G_2^\sigma[N(o_2)] \times \#(n_2^\sigma \times ... \times n_{k-1}^\sigma) \\
& + ... \\
& + G_k^\sigma[N(o_k)]\ ]\ )\ )\ (o_1, o_2, ...o_k)
\end{aligned}
\tag{6.3}
$$

Note that since the n_i^σ are known constants, the products of the form: $\#(n_1^\sigma \times ... \times n_j^\sigma)$, can be precomputed. Thus, only $k-1$ multiplications are required at run-time.

Note that Dujardin et.al assume a behavior specific function-call to compute the dispatch using Expression 6.3 [17]. Although this function-call reduces call-site size, it significantly increases dispatch time. The function-call has been inlined to make CNT more competitive in my timings.

6.1.4 SRP

SRP has K selector tables, denoted $S_1, ..., S_K$ where S_i represents the applicable method sets for types in argument position i of all methods. These dispatch tables can be compressed by any single-receiver dispatch technique, such as selector coloring (SRP/SC), row displacement (SRP/RD), or compressed dispatch table (SRP/CT). The timing and space results, and the code that follows, are for SRP/RD.

In addition to the argument-specific dispatch tables, SRP has, for each behavior, an array that maps method indices to method addresses, which is denoted by $D_\sigma^{k,SRP}$. The dispatch code for SRP is given in Expression 6.4, where FirstBit() is some macro or function that implements the operation of finding the position of the first '1' bit in a bit-vector. Holst et. al. discuss this in some detail [23]. My timing and space results assume that this is a hardware-supported operation with the same performance as shift-right.

$$\begin{aligned}(*(D_\sigma^{k,SRP}[\ FirstBit(S_1[N(o_1) + \#b_1^\sigma]\ \&\\ S_2[N(o_2) + \#b_2^\sigma]\ \&\\ \ldots\ \&\\ S_k[N(o_k) + \#b_k^\sigma])\ ]\ )\ )\ (o_1, o_2, ..., o_k)\end{aligned} \tag{6.4}$$

Note that $\#b_i^\sigma$ is the shift index assigned to behavior σ in argument-table i and is a literal integer.

6.1.5 LUA

LUA is, in some ways, the most difficult technique to evaluate accurately. First, there are a number of variations possible during implementation, that have vastly different space vs. time performance results. For example, in order to provide dispatch in $O(k)$, the technique must resort to an array access in certain situations, at the expense of substantially more memory. Second, Chen et. al. do not provide any explicit description of what the code at a particular call-site would look like [9]. They discuss the technique in terms of data structures, and do not mention that in a statically-typed environment, a collection of *if-then-else* statements would be a much more efficient implementation. It is only indicated later in [8] that method dispatch will happen as a function-call to a behavior-specific function. Given this assumption the call-site code for LUA is given in Expression 6.5.

$$dispatch_\sigma(o_1, o_2, ..., o_k); \tag{6.5}$$

Although the published discussion of LUA also assumes such a behavior-specific call, I have provided a more time-efficient implementation of LUA by inlining the dispatch computation (Expression 6.5), at the expense of more memory per call-site. Unfortunately, it is not feasible to inline the dispatch computation for LUA because the call-site code would grow too much.

My timing results assume the best possible dispatch situation for LUA, in which there are only two k-arity methods from which to choose. In such a situation, LUA needs to perform at most k subtype tests. Although numerous subtype-testing implementations are possible [25, 8], I have chosen one that provides a reasonable trade-off between time and space efficiency. Each type, T, maintains a bitvector, sub_T, in which the bit corresponding to every subtype of T is set to 1, and all other bits are set to 0. Assuming the bit-vector is implemented as an array of bytes, I can pack 8 bits into each array index, so determining whether T_j is a subtype of T_i consists of the expression: $sub_{T_i}[num(T_j) >> 3]\&(\ 1 << (num(T_j)\ \&\ 0x7)\ )$. However, note that the actual subtype testing implementation does not really affect the overall dispatch time because LUA invokes a behavior-specific dispatch function, and this extra function call is, in general, much more expensive than the actual computation itself.

The size of the per behavior function to be executed depends on the number of methods defined for the behavior. In the best possible case, there are only two methods, m_1 and m_2 defined for each behavior in a statically typed language (if there is only one method, no dispatch is necessary). I reiterate that this is a rather liberal under-estimate of the actual time a particular call-site takes to dispatch. The simplest function that a behavior can have is shown in the code:

```
dispatch_σ( o_1, ..., o_k ) {
        if ( sub_T1[N(o_1) >> 3] & ( 1 << (N(o_1) & 0x7) ))
             ...
                if ( sub_Tk[N(o_k) >> 3] & ( 1 << (N(o_k) & 0x7) ))
                     return call m_1(o_1, ..., o_k);
        return call m_2(o_1, ..., o_k);
}
```

6.2 Timing Results

In order to compare the address-computation time of the various techniques I generate technique-specific C++ programs that perform the computations listed in the previous section. Each program consists of a loop that iterates 2000 times over 500 blocks of code representing the address-computation for randomly generated call-sites, where a call-site consists of a behavior name and a list of k applicable types (for a k-arity behavior). Each block consists of two expressions. The first expression assigns to a global variable the result of an address-computation (i.e. the code described in the previous section, without the actual invocation). The second expression in each group calls a dummy function that modifies the previously assigned variable. These contortions are performed in order to stop the compiler from doing optimizations (such as only performing the last assignment in each group of 500, or in moving the code outside the 2000-iteration loop). Note that I am timing just the computation of addresses, since this is the only part of the dispatch process that varies from technique to technique (the actual invocation of the computed address is the same in all techniques). I also time a loop over 500 constant assignments interleaved with calls to the dummy function in order to time the overhead incurred (this is referred to as *noop* in the results).

Thus, each execution of one of these programs computes the time for 1,000,000 method-address computations. For each technique, such a program is generated and executed 20 times. The program is then regenerated (thus resulting in a different collection of 500 call-sites) an additional 9 times, and each such program is executed 20 times. This provides 200 timings of 1,000,000 call-sites for each of the techniques. The average time and standard-deviation of these 200 timings are reported in my results. In the graph, the histograms represent the mean, and the error-bars indicate the potential error in the results, as plus and minus twice the standard deviation.

In order to establish the effect that architecture and optimization have on the various techniques, the above timing results are performed on five different platforms using optimization levels from -O0 to -O3. All code is compiled using GNU C++ (in future work, it would be useful to obtain timings for a variety of different compilers). I present results for two platforms, and only for optimization level -O2. The other

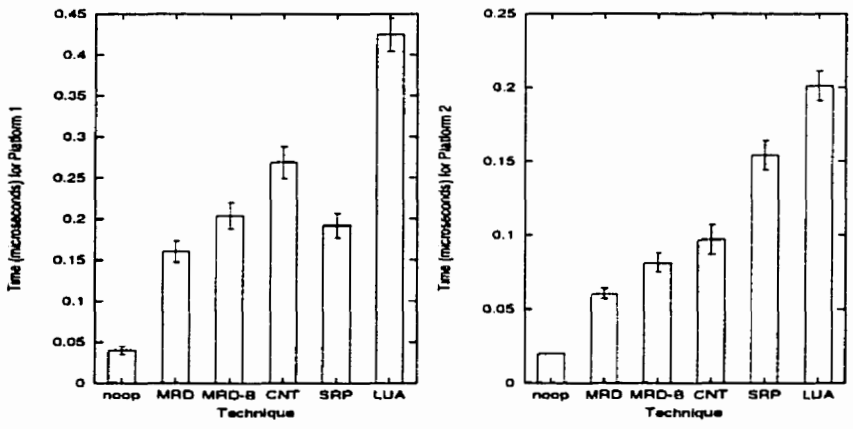


Figure 6.1: Number of microseconds required tc compute a method at a call-site

platforms and optimization level are similar. Furthermore, I only present results for 2-arity dispatch, since all techniques scale similarily for higher-arity dispatch sequences. In this chapter, Platform1 refers to a 299MHz Sun Microsystems Ultra 5/10 running Solaris 2.6 with 128 Mb of RAM and Platform2 refers to a 400MHz Prospec PII running Linux 2.0.34 with 256Mb of RAM.

From Figure 6.1, it can be seen that MRD provides the fastest dispatch time on both platforms, and did so for all five platforms tested.[1] Furthermore, LUA has the slowest dispatch time on all platforms. However, the relative performance of MRD-B, SRP and CNT varied with platform, although MRD-B was usually fastest, followed by SRP, followed by CNT.

6.3 Memory Utilization

Memory usage can be divided into two different categories: 1) data-structures, and 2) call-site code-size. The amount of space taken by each of these depends on the application, but in different ways. An application with many types and methods will naturally require larger data-structures than an application with fewer types and methods. As well, although the size of an individual call-site is independent of the application, the number of call-sites (and hence the amount of code generated) is

[1]The other three platforms were: a Sun SPARCstation 10 Model 50 running SunOS 4.1.4 with 128 Mb, an 180MHz SGI O2 running IRIX 6.5 with 64 Mb, and an IBM RS6000/360 running AIX 4.1.4 with 128 Mb

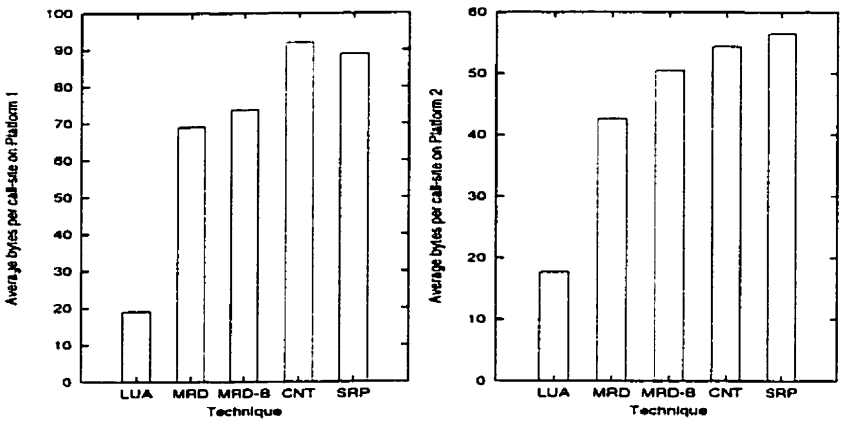


Figure 6.2: Call-Site Memory Usage

application dependent.

In order to compare the call-site size of the various techniques, I generated another set of technique-specific C++ programs. For each technique, a program was created that represented the code for 200 consecutive two-arity method invocations, including the dispatch computation. The program placed a label at the beginning and end of this code and reported the computed average call-site size based on the difference between the addresses of the labels. Note that the call-site size for a particular technique can vary slightly if the randomly generated arguments happen to be identical, or if the constants in the dispatch computation happen to be less than 256 or less than 65536, allowing them to be stored using smaller instructions.

Figure 6.2 shows the number of bytes required by the call-site dispatch code. Similar results are returned from higher arity behaviors.

Since the data-structure size is dependent on an application, I chose to measure the size required to maintain information for all types and all behaviors in two representative applications, the Cecil Vortex3 (Cecil compiler [7]) hierarchy and the Harlequin Dylan hierarchy (a Dylan [4] GUI hierarchy called *duim*). Harlequin is a commercial implementation of Dylan. The Cecil Vortex-3.0 hierarchy contains 1954 types, 11618 behaviors and 21395 method definitions. The Dylan hierarchy contains 666 types, 2146 behaviors and 3932 method definitions.

In order to measure the amount of space required by the various techniques, I filtered the set of all possible behaviors to arrive at the set of behaviors that truly

# Arity	# Behavior
2	203
3	22
4	11
Method Count	*# Behavior*
2	53
3	33
4	35
5-8	57
9-16	27
17-32	16
33+	5

(a) Cecil Vortex3 Type Hierarchy

# Arity	# Behavior
2	95
3	13
4	0
Method Count	*# Behavior*
2	21
3	11
4	32
5-8	23
9-16	13
17-32	6
33+	2

(b) Harlequin Type Hierarchy

Figure 6.3: Type Hierarchy Details for Two Different Hierarchies

require multi-method dispatch. In particular, I do not consider any 0-arity or 1-arity behaviors, because the address for such behaviors can be identified at compile-time and with single-receiver techniques respectively. Furthermore, since my data assumes a statically-typed language, I ignore behaviors with only one method defined on them, since they too can be determined at compile-time. Finally, for each remaining behavior, I remove any arguments in which only one type participates. If there is only one type in an argument position, no dispatch is required on that argument. For example, if behavior σ is defined only on $A \times A$, $B \times A$ and $C \times A$, then no dispatch on the second argument is required (because I am assuming statically typed languages). By reducing behaviors down to the set of arguments upon which multiple dispatch is truly required, I get an accurate measure of the amount of multi-method support the language requires. After the reduction, the Cecil Vortex3 hierarchy has 1954 types, 226 behaviors and 1879 methods, and the Dylan hierarchy has 666 types, 108 behaviors and 738 methods. The method distributions of these hierarchies are shown in Figure 6.3. The data-structure memory usage for each technique is shown in Figure 6.4.

In these reduced Cecil Vortex3 and Dylan hierachies, many of the method definitions have arguments typed as the *root-type*. Whenever an argument is typed as the

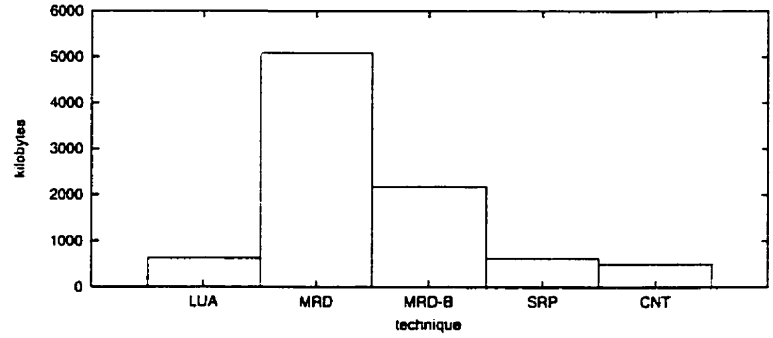


Figure 6.4: Static Data Structure Memory Usage for Cecil Vortex3

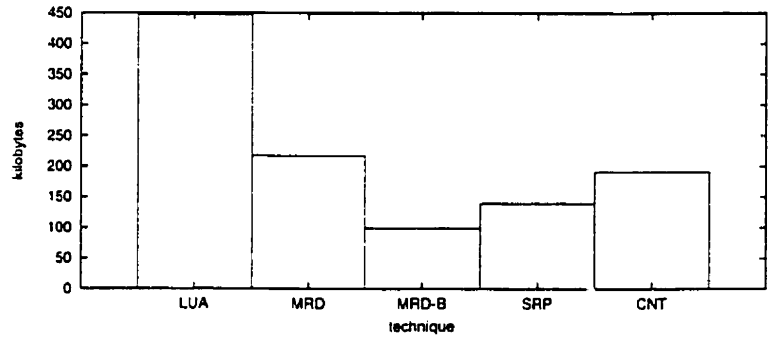


Figure 6.5: Static Data Structure Memory Usage for *No Root-Typed* Cecil Vortex3

root-type, MRD suffers. All rows on the dimension of that argument will be filled, so that, not much compression can be claimed from row-shifting or row-matching. More research is needed to find out whether it is a common practice to define many methods with arguments typed as the *root-type* in multi-method programming languages. However, if I remove all methods with *root-typed* argument(s) from the reduced Cecil Vortex3 hierarchy, the data structure size of each technique is profoundly different from those shown in Figure 6.4. As multi-methods become more common, I expect that the actual distribution of methods will be somewhere between these two extremes.

After removing all methods with *root-typed* argument(s), there are 1661 types, 660 behaviors and 1299 methods remaining in the Cecil Vortex3 hierarchy. The data structure size of each technique for this *no root-type* Cecil Vortex3 hierarchy is shown in Figure 6.5. The results for the Dylan hierarchy are similar.

Chapter 7

Future Work and Conclusion

7.1 Implementation

The research that produced MRD is part of a larger research project analyzing various multi-method dispatch techniques. Numerous issues impact the performance results given in this paper. For example, the simple loop-based timing approach poses a problem. It reports an artificially deflated execution time due to caching effects. Since the same data is being executed 10 million times, it stays hot. This problem can be partially solved by generating large sequences of random call-sites on different behaviors with different arguments. However, this approach might actually discount caching effects that would occur in a real program, since random distributions of call-sites will have poorer cache performance than real-world applications that have locality of reference.

Furthermore, some of the techniques allow for a variety of implementations. The implementations usually trade space for time, so an implementation can be chosen with the execution and memory footprint that most closely satisfies the requirements of a particular application. Also related to the issue of implementation is the impact of inlining of dispatch code. In single-receiver languages, the dispatch code is placed inline at each call-site, but some of the multi-method dispatch techniques have large call-site code chunks. For example, LUA defines a single dispatch function for each behavior. This function reduces call-site size, but significantly increases dispatch time. Rather than always calling a function, conditional inlining of a call-site is an open area of future research.

For all of the multi-method table-based dispatch techniques introduced in this

thesis, except LUA, dispatch code is placed inline. Therefore, as the size of a program grows, the number of call-sites increases, so the dispatch memory usage increases. Alternatively, function calls can be used instead of inline code for all techniques. In this case, memory usage will not increase as the number of call-sites increase. More expermentation is needed to assess the time and space trade-off for function calls instead of inline code.

7.2 Non-Statically Typed Languages

In the thesis, I have investigated multi-method table-based dispatch techniques for statically typed languages only. All table-based techniques described in this thesis can be extended to handle non-statically typed languages. Each technique can be extended in many different ways to handle non-statically typed variables. As described in Section 4.6, MRD has two ways to solve the index out of bounds error, and two ways to solve the wrong method error. Each strategy has its own advantages and disadvantages. Therefore, a detailed investigation is needed to find out which strategy has the best time and space trade off for each dispatch technique. Then, different techniques can be compared on their performance for non-statically typed languages.

7.3 Reflexive Environment

Currently not much effort has been spent on extending multi-method table-based dispatch techniques to handle a reflexive environment. Obviously, N-Dimensional Tables can handle reflexivity easily, however it has been ruled out because of its huge memory usage. Nothing has been done on extending CNT, LUA or MRD to handle incremental environment changes. Since it takes only a few seconds to rebuild the whole data structure for each of these techniques, it may be acceptable to rebuild the whole data structure for each environment change. The only table-based technique that has an advantage in a reflexive environment is SRP. SRP is based on single-receiver table-based dispatch techniques. Holst et. al. described how single-receiver table-based techniques handle incrementally environment changes [22]. Therefore, there is still a lot of study necessary for table-based dispatch techniques for reflexive environments.

Another question is how to compare different techniques on their performance for reflexive environments. What should be measured?

7.4 Object-Oriented Language Usage Metrics

Randomly generated call-sites are used in performance evaluation in this thesis. However, in order to obtain the best possible analysis of the various techniques, we need some indepth metrics on the distribution of behaviors in multi-method languages. In particular, the number of behaviors of each arity, and the numbers of methods defined per behavior are critical. As more and more multi-method languages are introduced, we will be able to get a better feel for realistic distributions. Note that call-site distributions are especially important for accurate analysis of LUA, since its dispatch time depends on the average number of types that need to be tested before a successful match occurs.

7.5 Summary

As described in the Introduction, mutli-method languages have more expressive power than single-receiver lanugages. When dispatch algorithms become more efficient and computing power increases, multi-method languages will be more popular. In this thesis, a new time and space efficient multi-method dispatch technique. Multiple Row Displacement (MRD), is presented. MRD compresses an n-dimensional table by row displacement. It has been compared with existing table-based multi-method techniques, CNT, LUA and SRP. MRD has the fastest dispatch time and the second smallest per-call-site code size (next to LUA, which uses a function call). If the other techniques used a function call, they could reduce their call-site size at the expense of dispatch time.

In addition to presenting the new technique, this thesis has provided a performance comparison of existing table-based multi-method dispatch techniques. This thesis is one step in making multi-method languages more suitable for general use.

Bibliography

[1] *ECOOP'97 Conference Proceedings*, 1997.

[2] Eric Amiel, Olivier Gruber, and Eric Simon. Optimizing multi-method dispatch using compressed dispatch tables. In *OOPSLA'94 Conference Proceedings*, 1994.

[3] P. Andre and J.C. Royer. Optimizing method search with lookup caches and incremental coloring. In *OOPSLA'92 Conference Proceedings*, 1992.

[4] Apple Computer, Inc. *Dylan Interim Reference Manual*, 1994.

[5] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp Object System specification, June 1988. X3J13 Document 88-002R.

[6] Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. Commonloops: Merging lisp and object-oriented programming. In *OOPSLA'86 Conference Proceedings*, pages 17–29, 1986.

[7] Craig Chambers. Object-oriented multi-methods in cecil. In *ECOOP'92 Conference Proceedings*, 1992.

[8] Craig Chambers and Weimin Chen. Efficient predicate dispatch, 1998. Technical Report UW-CSE-98-12-02.

[9] Weimin Chen. Efficient multiple dispatching based on automata. Master's thesis, Darmstadt, Germany, 1995.

[10] Weimin Chen, Volker Turau, and Wolfgang Klas. Efficient dynamic look-up strategy for multimethods. In *ECOOP'94 Conference Proceedings*, 1994.

[11] Brad Cox. *Object-Oriented Programming, An Evolutionary Approach*. Addison-Wesley, 1987.

[12] L. Peter Deutsch and Alan Schiffman. Efficient implementation of the Smalltalk-80 system. In *Principles of Programming Languages*, Salt Lake City, UT, 1994.

[13] R. Dixon, T. McKee, P. Schweizer, and M. Vaughan. A fast method dispatcher for compiled languages with multiple inheritance. In *OOPSLA'89 Conference Proceedings*, 1989.

[14] K. Driesen and U. Hölzle. Minimizing row displacement dispatch tables. In *OOPSLA'95 Conference Proceedings*, 1995.

[15] K. Driesen, U. Hölzle, and J. Vitek. Message dispatch on pipelined processors. In *ECOOP'95 Conference Proceedings*, 1995.

[16] Karel Driesen. Selector table indexing and sparse arrays. In *OOPSLA'93 Conference Proceedings*, 1993.

[17] Eric Dujardin, Eric Amiel, and Eric Simon. Fast algorithms for compressed multimethod dispatch table generation. In *Transactions on Programming Languages and Systems*, 1996.

[18] M.A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

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

[20] Wade Holst and Duane Szafron. Inheritance management and method dispatch in reflexive object-oriented languages. Technical Report TR-96-27, University of Alberta, Edmonton, Canada, 1996.

[21] Wade Holst and Duane Szafron. A general framework for inheritance management and method dispatch in object-oriented languages. In *ECOOP'97 Conference Proceedings* [1].

[22] Wade Holst and Duane Szafron. Incremental table-based method dispatch for reflexive object-oriented languages. In *Technology of Object-Oriented Languages and Systems*, 1997.

[23] Wade Holst, Duane Szafron, Yuri Leontiev, and Candy Pang. Multi-method dispatch using single-receiver projections. Technical Report TR-98-03, University of Alberta, Edmonton, Canada, 1998.

[24] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object oriented languages with polymorphic inline caches. In *ECOOP'91 Conference Proceedings*, 1991.

[25] Andreas Krall, Jan Vitek, and R. Nigel Horspool. Near optimal hierarchical encoding of types. In *ECOOP'97 Conference Proceedings* [1].

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

[27] M.T. Ozsu, R.J. Peters, D. Szafron, B. Irani, A. Lipka, , and A. Munoz. Tigukat: A uniform behavioral objectbase management system. In *The VLDB Journal*, pages 100–147, 1995.

[28] Jan Vitek and R. Nigel Horspool. Taming message passing: Efficient method lookup for dynamically typed languages. In *ECOOP'94 Conference Proceedings*, 1994.

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