

# Multi-Method Dispatch Using Multiple Row Displacement<sup>\*</sup>

Candy Pang, Wade Holst, Yuri Leontiev, and Duane Szafron

University of Alberta, Edmonton AB T6G 2H1 Canada  
{candy,wade,yuri,duane}@cs.ualberta.ca

**Abstract.** 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 paper 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 uses comparable space to the other techniques, but has faster dispatch performance.

## 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* use the dynamic types of one or more arguments<sup>1</sup> 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 dispatch on the types of these arguments; the static types are simply encoded within the method name.

Since most of the commercial object-oriented languages are single-receiver languages, many efficient dispatch techniques have been invented for such languages [1]. However, there are some multi-method languages in use, such as Cecil [2], CLOS [3], and Dylan [4]. In such languages, multi-method dispatch is necessary.

There are two major categories of method dispatch: *cache-based* and *table-based*. *Cache-based* techniques look in either global or local caches at the time of dispatch to determine if the method for a particular call-site has already

---

<sup>\*</sup> This research was supported in part by the Natural Sciences and Engineering Research Council (NSERC) of Canada under grant OGP8191

<sup>1</sup> In the rest of this paper, we will assume that dispatch occurs on all arguments.

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. *Table-based* techniques pre-determine the method for every possible call-site, and record these methods in a table. At dispatch-time, the method name and dynamic argument types form an index into this table. This paper focuses exclusively on table-based techniques. Table-based techniques have constant dispatch time. In addition, even when cache-based techniques are used, table-based techniques can be effectively used for cache-misses.

In this paper we present 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 paper is organized as follows. Sect. 2 introduces some notation for describing multi-method dispatch. Sect. 3 presents the row displacement single-receiver dispatch technique. Sect. 4 summarizes the existing multi-method dispatch techniques. Sect. 5 describes n-dimensional table dispatch and presents the new multi-method table-based technique. Sect. 6 presents time and space results for the new technique and compares it to existing techniques. Sect. 7 discusses future work, and Sect. 8 presents our conclusions.

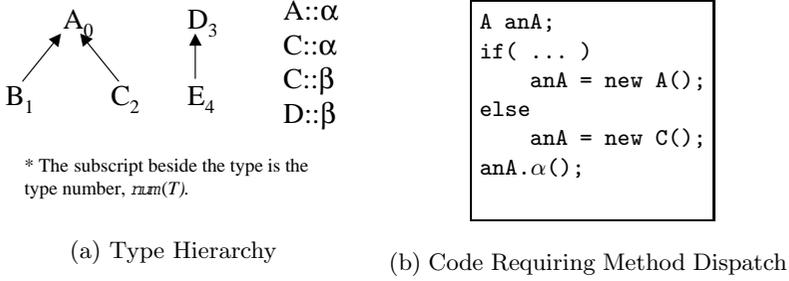
## 2 Terminology for Multi-Method Dispatch

### 2.1 Notation

Expr. 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*,  $T^i = \text{type}(o_i)$ . 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,  $\text{num}(T)$ . A directed *supertype edge* exists between type  $T_j$  and type  $T_i$  if  $T_j$  is a *direct subtype* of  $T_i$ , which we denote 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  $T_j \prec T_i$ .

$$\sigma(o_1, o_2, \dots, o_k) \tag{1}$$

*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 Fig. 1a, and the code in Fig. 1b. The static type of `anA` is `A`, but the dynamic type of `anA` can be either `A` or `C`. In general, we do not know the dynamic type of an object at a call-site until run-time, so method dispatch is necessary.



**Fig. 1.** An example hierarchy and program segment requiring method dispatch

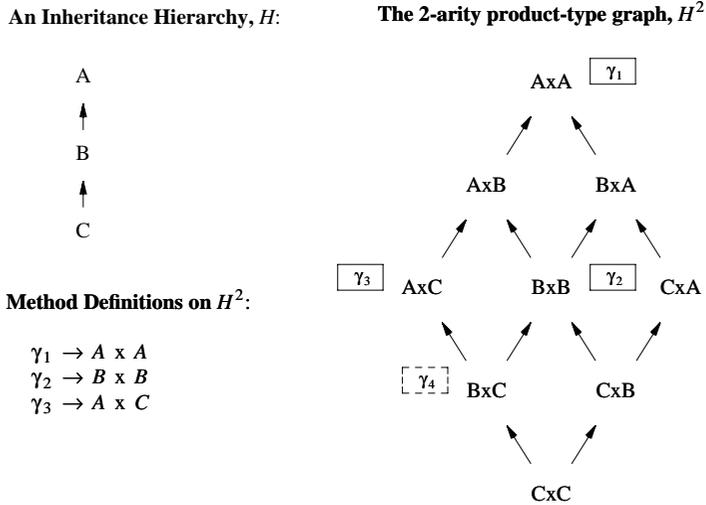
Although multi-method languages might appear to break the conceptual model of sending a message to a receiver, we can maintain this idea by introducing the concept of a product-type. A *k-arity product-type* is an ordered list of  $k$  types denoted by  $P = T^1 \times T^2 \times \dots \times T^k$ . 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  $\mathcal{H}$ . Expr. 2 describes when a directed edge exists from a child product-type  $P_j = T_2^1 \times T_2^2 \times \dots \times T_2^k$  to a parent product-type  $P_i = T_1^1 \times T_1^2 \times \dots \times T_1^k$ , which is denoted  $P_j \prec_1 P_i$ .

$$P_j \prec_1 P_i \Leftrightarrow \exists i, 1 \leq i \leq k : (T_2^i \prec_1 T_1^i) \wedge (\forall j \neq i, T_2^j = T_1^j) \quad (2)$$

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  $\mathcal{H}^k$ . Fig. 2 presents a sample inheritance hierarchy  $\mathcal{H}$  and its induced 2-arity product-type graph,  $\mathcal{H}^2$ . Three 2-arity methods ( $\gamma_1$  to  $\gamma_3$ ) for behavior  $\gamma$  have been defined on  $\mathcal{H}^2$  and associated with the appropriate product-types.<sup>2</sup> Note that for real inheritance hierarchies, the product-type hierarchies,  $\mathcal{H}^2, \mathcal{H}^3, \dots$ , are too large to store explicitly. Therefore, it is essential to define all product-type relationships in terms of relations between the original types, as in Expr. 2. Next, we 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  $\mathcal{B}_\sigma^k$ , where  $k$  is the arity and  $\sigma$  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  $\sigma$  is denoted by  $\sigma_j$ . If the static type of the  $i^{th}$  argument of  $\sigma_j$  is denoted by  $T^i$ , the list of argument types can be viewed as a product-type,  $dom(\sigma_j) = T^1 \times T^2 \times \dots \times T^k$ . With multi-method dispatch, the dynamic types of all arguments are needed.<sup>3</sup>

<sup>2</sup> The method  $\gamma_4$  in the dashed box is an implicit inheritance conflict definition, and will be explained later.

<sup>3</sup> In single-receiver languages, the first argument is called a receiver.



**Fig. 2.** An Inheritance Hierarchy,  $\mathcal{H}$ , and its induced Product-Type Graph  $\mathcal{H}^2$

### 2.2 Inheritance Conflicts

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  $T_j$ . Most languages relax this definition slightly. Assume that  $n$  different methods of a behavior are defined on the set of types  $\mathcal{T} = \{T_1, \dots, T_n\}$ , where  $T \preceq T_1, \dots, T_n$ . 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  $\{\exists T_k \in \mathcal{T} \mid T_k \prec T_i \ \& \ T_k \prec T_j\}$ .

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  $T^1 \times T^2 \times \dots \times T^k$ . Interestingly, inheritance conflicts can occur in multi-method languages even if the underlying type hierarchy,  $\mathcal{H}$ , has single inheritance. For example, in Fig. 2, the product-type  $B \times C$  has an inheritance conflict, since it can see two different definitions for behavior  $\gamma$  ( $\gamma_3$  in  $A \times C$  and  $\gamma_2$  in  $B \times B$ ). For this reason, an implicit conflict method,  $\gamma_4$ , is defined in  $B \times C$  as shown in Fig. 2. Similar to single-receiver languages, relaxation can be applied. Assume that  $n$  methods are defined in product-types  $\mathcal{P} = \{P_1, \dots, P_n\}$ , and let  $P \prec P_1, \dots, P_n$ . 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 Statically Typed Versus Dynamically Typed

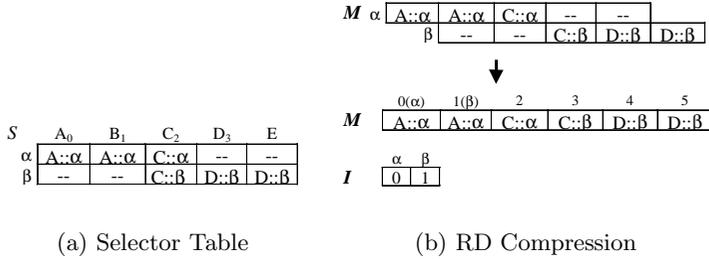
Some programming languages (C++, Java, Eiffel) require each variable to be declared with a static type. These languages are called statically typed languages. Other languages (Smalltalk, CLOS) which do not bind variables to static types, are called dynamically typed languages. In statically typed languages, 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 Fig. 1b. The dynamic type of *anA* can be either *A* or *C* (which is a subtype of *A*). Since the message  $\alpha$  is defined for type *A*, no matter what its dynamic type is, *anA* can understand the message  $\alpha$ . Therefore, the type checker can tell at compile-time that the call-site *anA.alpha()* is type-valid. If the static type of *anA* is *D*, neither *D* nor any of its supertypes understand the message  $\alpha$ . The type checker will find at compile-time that the call-site *anA.alpha()* is type-invalid, and return a compile-time error.

With implicitly defined conflict methods in statically typed languages, no type-invalid call-site will be dispatched during execution. However, in dynamically typed languages, call-sites may be type-invalid. All dispatch techniques that use compression may return a method totally unrelated to the call-site. Therefore, in dynamically typed languages, a method prologue is used to ensure that the computed method is applicable for the dispatched behavior. It also ensures that each of the arguments is a subtype of the associated parameter type in the method.

The dispatch technique, Multiple Row Displacement introduced in this paper also has the problem of returning a wrong method for a type-invalid call-site in dynamically typed languages. The problem can be solved by minor changes to the data structure, see [18] for details. However, in this paper we assume the call-sites are statically typed.

## 3 Single-Receiver Row Displacement Dispatch (RD)

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*. Fig. 3a shows the selector table for the type hierarchy and method definitions in Fig. 1a. 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* [5]. This algorithm is known as STI in the literature [6]. 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



**Fig. 3.** Compressing A Selector Table By Row Displacement

required 4 bytes, then the selector table would have 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 [7], so the table can be compressed.

Row displacement (RD) reduces the number of empty entries by compressing the two-dimensional selector table into a one-dimensional array [7,8]. As illustrated in Fig. 3b, 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, we will just use the behavior name in this paper. 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  $\beta$  with  $D$  as the dynamic type of the receiver, the shift index for  $\beta$  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  $D::\beta$ . Compared with other single-receiver table dispatch techniques, row displacement is highly space and time efficient [1]. We will show how this single-receiver technique can be generalized to multi-method languages in Sect. 5.

## 4 Existing Multi-Method Dispatch Techniques

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

1. *CNT: Compressed N-Dimensional Tables* [9,10,11] represents the dispatch table as a behavior-specific  $k$ -dimensional table, where  $k$  represents the arity of a particular behavior. Each dimension of the table is compressed by grouping identical dimension lines into a single line. The resulting table is indexed by *type groups* in each dimension, and mappings from type number to type group are kept in auxiliary data structures.

2. *LUA: Lookup Automata* [12,13] creates a lookup automaton for each behavior. In order to avoid backtracking, and thus exponential dispatch time, the automata must include more types than are explicitly listed in method definitions (inheritance conflicts must be implicitly defined). The automaton can then be converted to a function containing only *if-then-else* statements. At dispatch, this function is called to locate the correct method. LUA has been extended in [19].
3. *SRP: Single-Receiver Projections* [14] maintains  $k$  extended single-receiver dispatch tables and projects  $k$ -arity multi-method definitions onto these  $k$  tables. Each table maintains a bit-vector of applicable method indices, so dispatch consists of logically anding bit-vectors, finding the index of the right-most on-bit and returning the method associated with this index.
4. *Extended Cache-Based Techniques* are used in Cecil [2]. The cache-based techniques from single-receiver languages [6] are extended to work for product-types instead of just simple types.

## 5 Multiple Row Displacement (MRD)

### 5.1 N-dimensional Dispatch 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 multi-method dispatch. In multi-method dispatch, each  $k$ -arity behavior,  $\mathcal{B}_\sigma^k$ , has a  $k$ -dimensional dispatch table,  $D_\sigma^k$ , with type numbers as indices for each dimension. Therefore, each  $k$ -dimensional dispatch table has  $|\mathcal{H}|^k$ . At a call-site,  $\sigma(o_1, o_2, \dots, o_k)$ , the method to execute is in  $D_\sigma^k[num(T^1)][num(T^2)]\dots[num(T^k)]$ , where  $T^i = type(o_i)$ . For example, the 2-dimensional dispatch tables for the type hierarchy and method definitions in Fig. 4a are shown in Fig. 4b. In building an  $n$ -dimensional dispatch table, inheritance conflicts must be resolved. For example, there is an inheritance conflict at  $E \times E$  for  $\alpha$ , since both  $\alpha_1$  and  $\alpha_2$  are applicable for the call-site  $\alpha(anE, anE)$ . Therefore, we define an implicit conflict method  $\alpha_3$ , and insert it 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 Cecil Vortex3 type hierarchy there are 1954 types. Therefore, a single 3-arity behavior would require  $1954^3$  bytes = 7.46 gigabytes.

### 5.2 Multiple Row Displacement by Examples

Multiple Row Displacement (MRD) is a time and space efficient dispatch technique which combines row displacement and  $n$ -dimensional dispatch tables. We will first illustrate MRD through examples, and then give the algorithm. The

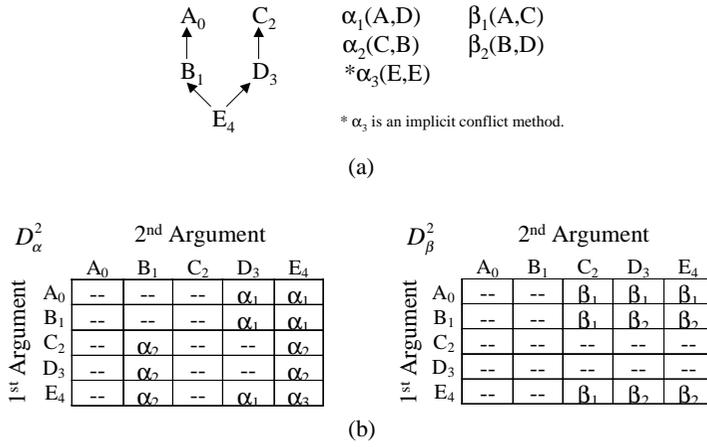


Fig. 4. N-Dimensional Dispatch Tables

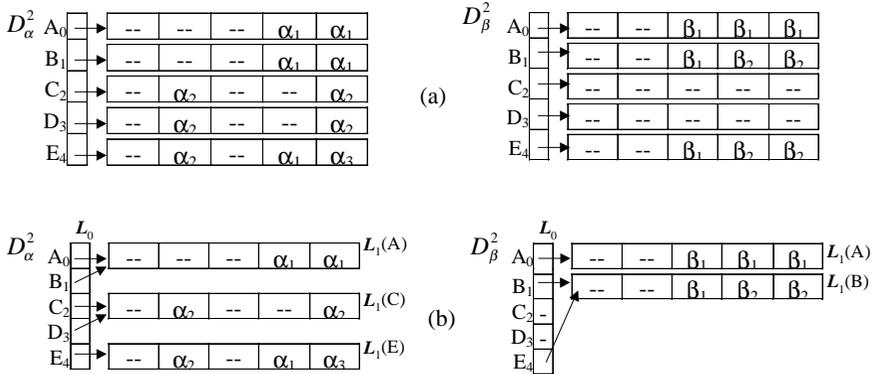


Fig. 5. Data Structure for Multiple Row Displacement

first example uses the type hierarchy and 2-arity method definitions from Fig. 4a. Instead of the single  $k$ -dimensional array shown in Fig. 4b, each table can be represented as an array of arrays as shown in Fig. 5a. 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 Fig. 5a that some of the *level-1* arrays are exactly the same. Those arrays are combined as shown in Fig. 5b. 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 Sect. 4, 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 that we are compressing a table that 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 [15] 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) [16].

We have one data structure per behavior,  $D_\sigma^k$ , 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  $j = 0, \dots, (K-2)$ , and a Global Behavior Array,  $B$ .

In compressing the data structure  $D_\alpha^2$  in Fig. 5b, the *level-1* array  $L_1(A)$  is first shifted into the Global Master Array,  $M$ , by row displacement, as shown in Fig. 6a. The shift index, 0, is stored in the *level-0* array,  $L_0$ , in place of  $L_1(A)$ . In the implementation, a temporary array is created to store the shift indices, but in this paper, we will put them in  $L_0$  for simplicity of presentation. Fig. 6b 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. Finally, as shown in Fig. 6c,  $L_0$  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  $D_\alpha^2$  is compressed into the global data structures, the memory for its preliminary data structures can be released. Fig. 7 shows how to compress the behavior data structure,  $D_\beta^2$ , 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 Fig. 7a. The compression of the *level-0* array,  $L_0$ , is shown in Fig. 7b. 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. As an example of dispatch, we will demonstrate how to

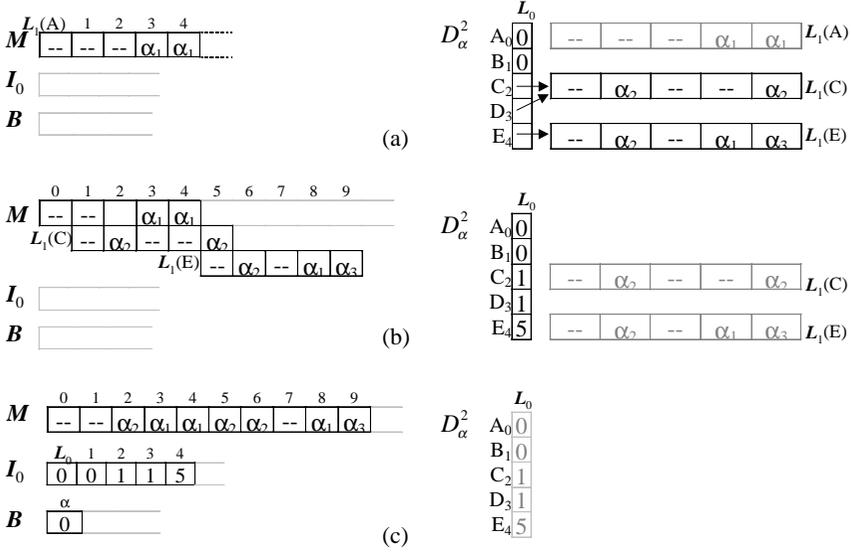


Fig. 6. Compressing The Data Structure for  $\alpha$

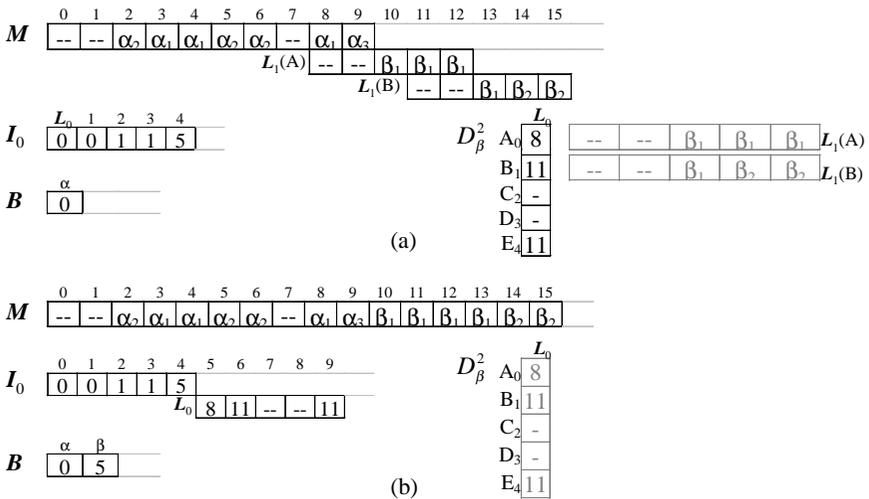


Fig. 7. Compressing The Data Structure For  $\beta$

dispatch a call-site  $\beta(anE, aD)$  using the data structures in Fig. 7b. The method dispatch starts by obtaining the shift index of the behavior,  $\beta$ , from the Global Behavior Array,  $B$ . From Fig. 7b,  $B[\beta]$  is 5. The next step is to obtain the shift index for the type of the first argument,  $E$ , from the Global Index array,  $I_0$ . Since the shift index of  $\beta$  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. Fig. 8a shows the method definitions of a 3-arity behavior,  $\delta$ , and Fig. 8b shows its preliminary behavior data structure,  $D_\delta^3$ . Figs. 8c to 8e 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 Fig. 8c. Their shift indices (15, 14, 19) are stored in  $L_1(B)$ ,  $L_1(D)$  and  $L_1(E)$ . In fact, every pointer in Fig. 8b 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 Fig. 8d. 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 Fig. 8e.

### 5.3 A Description of the Multiple Row-Displacement Algorithm

We have shown, by examples, how MRD compresses an n-dimensional dispatch table by row displacement. On the behavior level, a preliminary data structure,  $D_\sigma^k$ , is created for each behavior.  $D_\sigma^k$  is a data structure for a k-arity behavior named  $\sigma$ , as shown in Fig. 8b. It is actually an n-dimensional dispatch table, which is an array of pointers to arrays. Each array in  $D_\sigma^k$  has size  $|\mathcal{H}|$ . 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 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 is a Global Master Dispatch Array,  $M$ ,  $K - 1$  Global Index Arrays,  $I_0, \dots, I_{k-2}$ , 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  $D_\sigma^k$  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 shown in Sect. 5.5.

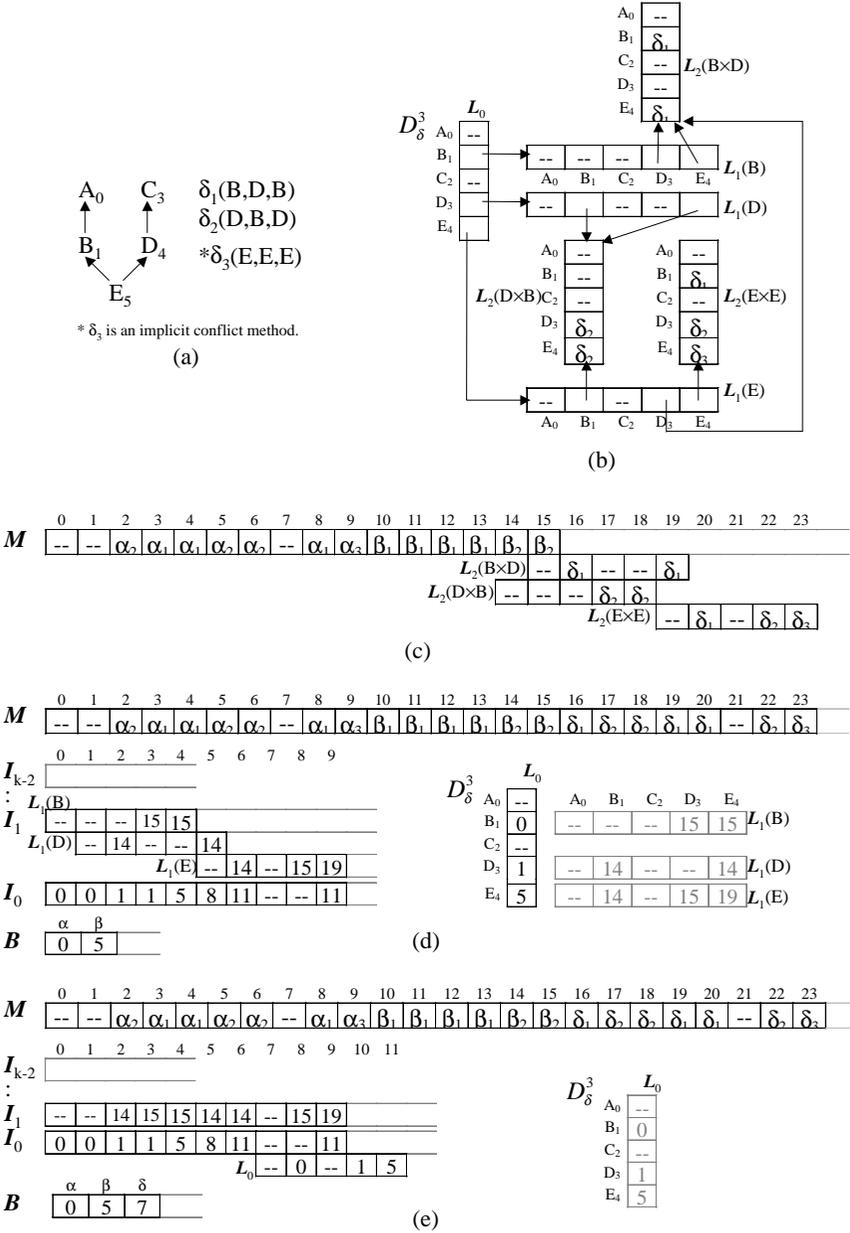


Fig. 8. Compressing The Data Structure For  $\delta$

The dispatch formula for a call-site,  $\sigma(o_1, \dots, o_k)$ , is given by Expr. 3, where  $T^i = \text{type}(o_i)$ .

$$M[ I_{k-2}[ I_{k-3}[ \dots I_1[ I_0[ B[ \sigma ] + \text{num}(T^1) ] \\ + \text{num}(T^2) ] + \dots ] + \text{num}(T^{k-2}) ] + \text{num}(T^{k-1}) ] + \text{num}(T^k) ] \quad (3)$$

As an example of dispatch with Expr. 3, we will demonstrate how to dispatch a call-site  $\delta(anE, aD, aB)$  using the data structures in Fig. 8e. Since  $\delta$  is a 3-arity behavior, Expr. 3 becomes Expr. 4.

$$M[ I_1[ I_0[ B[ \delta ] + \text{num}(E) ] + \text{num}(D) ] + \text{num}(B) ] \quad (4)$$

Substituting the data from Fig. 8e into Expr. 4 yields the method  $\delta_1$ , as shown in Expr. 5.

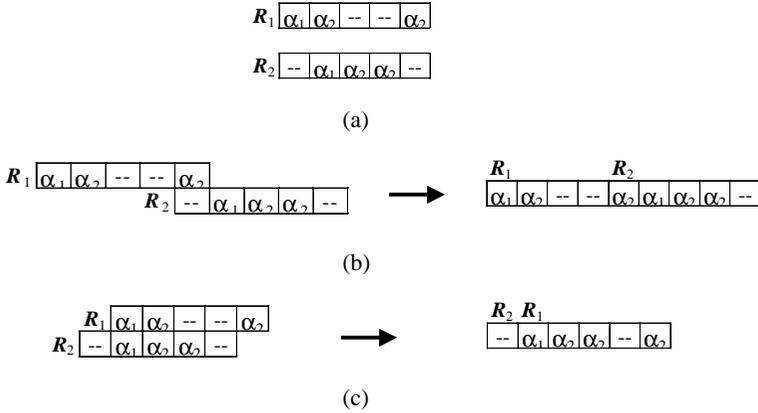
$$\begin{aligned} & M[ I_1[ I_0[ 7 + 4 ] + 3 ] + 1 ] \\ & = M[ I_1[ I_0[ 11 ] + 3 ] + 1 ] \\ & = M[ I_1[ 5 + 3 ] + 1 ] \\ & = M[ I_1[ 8 ] + 1 ] \\ & = M[ 15 + 1 ] \\ & = M[ 16 ] = \delta_1 \end{aligned} \quad (5)$$

## 5.4 Optimizations

**Single  $I$ .** For simplicity of presentation, we defined an Index Array per arity position. Actually, we only need one Global Index Array,  $I$ , to store all *level-0* to *level-(k-2)* arrays. Using a single Index Array provides additional compression, and has no negative impact on dispatch speed. Expr. 6 shows the modified dispatch formula that accesses one Global Index Array.

$$M[ I[ I[ \dots I[ I[ B[ \sigma ] + \text{num}(T^1) ] \\ + \text{num}(T^2) ] + \dots ] + \text{num}(T^{k-2}) ] + \text{num}(T^{k-1}) ] + \text{num}(T^k) ] \quad (6)$$

**Row-Matching.** Note that the row-shifting mechanism used in our 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), we get a higher compression rate. 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 Fig. 9a produces a master array with 9 elements as shown in Fig. 9b. However, using the improved algorithm to compress R1 and R2 produces a master array with only 6 elements as shown in Fig. 9c. Using row-matching instead of row-shifting provides an additional 10-14% compression. Our improved algorithm cannot be used in single-receiver row displacement, since different rows contain different behaviors, and thus different addresses.



**Fig. 9.** Row-Shifting vs. Row-Matching

**Byte vs. Word Storage.** MRD stores four bytes function addresses in  $M$ . In a large hierarchy,  $M$  is the most memory consuming data structure. To reduce the size of  $M$ , a method-map,  $D_\sigma^{MRD}$ , is introduced per behavior. Since all methods of a behavior are stored in  $D_\sigma^{MRD}$ , a method can be represented by an index into  $D_\sigma^{MRD}$ . 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  $D_\sigma^{MRD}$ . If  $M$  stores this index instead of the function address, the size of  $M$  is reduced to one-fourth of its original size. However, there is an extra indirection to access the method-map at dispatch time. We denote the technique which stores bytes instead of words by MRD-B.

**Type Ordering.** In single receiver row displacement type ordering has a significant impact on compression ratios [7]. We have investigated type ordering in multi-method row displacement and found that the impact is smaller.

## 5.5 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  $B_\sigma^k$ ) do
    BehaviorStructure  $D_\sigma^k = B_\sigma^k.createStructure()$ ;
    createRecursiveStructure(  $D_\sigma^k.L_0$ , 0 );
     $B_\sigma^k.shiftIndex = D_\sigma^k.L_0.getShiftIndex()$ ;
  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.

## 5.6 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 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.

## 6 Performance Results

Here we present memory and execution results for the new technique, MRD, and three other techniques, CNT, LUA and SRP. When analyzing dispatch tech-

niques, both execution performance 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. We present both timing and memory results 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 the subsections that follow, the code presented refers to the code that would be generated by the compiler upon encountering the call-site  $\sigma(o_1, o_2, \dots, o_k)$ .

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. Our 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).

**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 Expr. 7.

$$(* (M [ I [ \dots I [ I [ \#b^\sigma + N(o_1) ] + N(o_2) ] + \dots ] + N(o_{k-1}) ] + N(o_k) ] ] ) (o_1, o_2, \dots, o_k) \quad (7)$$

Note that the Global Behavior Array,  $B$ , from Expr. 3, is known at compile-time, so  $B[\sigma]$  is known at compile-time. Thus  $\#b^\sigma$  is a literal integer obtained from  $B[\sigma]$ .

**MRD-B.** The dispatch sequence for MRD-B is given in Expr. 8.

$$(* (D_\sigma^{MRD} [ M [ I [ \dots I [ I [ \#b^\sigma + N(o_1) ] + N(o_2) ] + \dots ] + N(o_{k-1}) ] + N(o_k) ] ] ) (o_1, o_2, \dots, o_k) \quad (8)$$

**CNT.** For each behavior, CNT has a  $k$ -dimensional array, but since we are assuming a static 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, we denote its one-dimensional dispatch table by  $D_\sigma^{CNT}$ .

In addition to the behavior-specific information, CNT requires arrays that map types to type-groups. In [11], these group arrays are compressed by selector coloring (SC). Our 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, we assume that there are arrays  $G_1, \dots, G_n$ , and that the compiler knows which group array to use for each dimension of a particular behavior.

If we assume that the compressed  $n$ -dimensional table for  $k$ -arity behavior  $\sigma$  has dimensions  $n_1^\sigma, n_2^\sigma, \dots, n_k^\sigma$ , where the  $n_i^\sigma$  values are behavior specific, and that the group arrays for these dimensions are  $G_1^\sigma, G_2^\sigma, \dots, G_k^\sigma$  then the call-site dispatch code is given in Expr. 9.

$$\begin{aligned}
 (* (D_\sigma^{CNT} [ & G_1^\sigma[N(o_1)] \times \#(n_1^\sigma \times n_2^\sigma \times \dots \times n_{k-1}^\sigma) \\
 & + G_2^\sigma[N(o_2)] \times \#(n_2^\sigma \times \dots \times n_{k-1}^\sigma) \\
 & + \dots \\
 & + G_k^\sigma[N(o_k)] ] ) ) (o_1, o_2, \dots, o_k)
 \end{aligned} \tag{9}$$

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

Note that [11] assumes a behavior specific function-call to compute the dispatch using Expr. 9. Although this function-call reduces call-site size, it significantly increases dispatch time. We have remove the function-call by inlining to make CNT more competitive in our timings.

**SRP.** SRP has  $K$  selector tables, denoted  $S_1, \dots, S_K$  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 we denote by  $D_\sigma^{SRP}$ .

The dispatch code for SRP is given in Expr. 10, where `FirstBit()` is some macro or function that implements the operation of finding the position of the first '1' bit in a bit-vector. [14] discusses this in some detail. Our timing and space results assume that this is a hardware-supported operation with the same

performance as shift-right.

$$\begin{aligned}
 & ( * ( D_{\sigma}^{SRP} [ FirstBit( S_1 [ N(o_1) + \#b_1^{\sigma} ] \& \\
 & \qquad \qquad \qquad S_2 [ N(o_2) + \#b_2^{\sigma} ] \& \\
 & \qquad \qquad \qquad \dots \& \\
 & \qquad \qquad \qquad S_k [ N(o_k) + \#b_k^{\sigma} ] ] ) ) ( o_1, o_2, \dots, o_k ) \quad (10)
 \end{aligned}$$

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

**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, [13] does not provide any explicit description of what the code at a particular call-site would look like. 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 [19] 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 Expr. 11.

$$dispatch_{\sigma}(o_1, o_2, \dots, o_k); \quad (11)$$

Although the published discussion of CNT also assumes such a behavior-specific call, we have provided a more time-efficient implementation of CNT by inlining the dispatch computation (Expr. 9), 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.

Our 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 [17,19], we 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, we can pack 8 bits into each array index, so determining whether  $T_j$  is a subtype of  $T_i$  consists of the expression:  $sub_{T_i}[num(T_j) \gg 3] \& ( 1 \ll ( num(T_j) \& 0x7) )$ . 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). We 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 $\sigma$ (  $o_1, \dots, o_k$  ) {
    if (  $sub_{T^1}[N(o_1) >> 3]$  & (  $1 << (N(o_1) \& 0x7)$  ))
        ...
        if (  $sub_{T^k}[N(o_k) >> 3]$  & (  $1 << (N(o_k) \& 0x7)$  ))
            return call  $m_1(o_1, \dots, o_k)$ ;
        return call  $m_2(o_1, \dots, o_k)$ ;
}
```

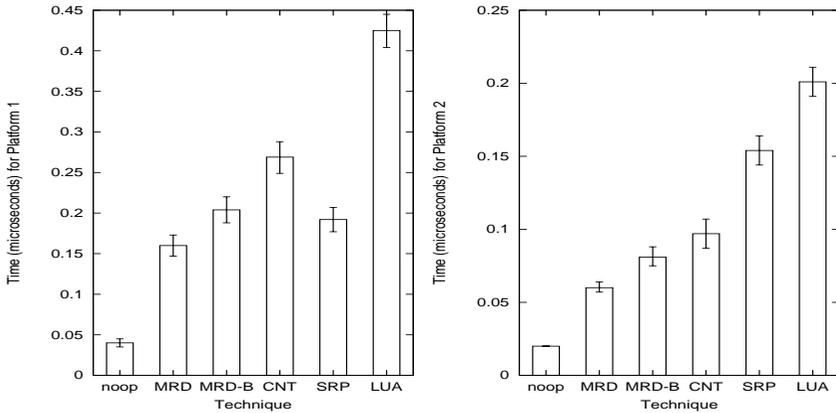
## 6.2 Timing Results

In order to compare the address-computation time of the various techniques we 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 we are 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). We 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 our 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, we will obtain timings for a variety of different

compilers). In the interest of space, we present results for two platforms, and only for optimization level -O2. Furthermore, we only present results for 2-arity dispatch, since all techniques scale similarly for higher-arity dispatch sequences. In this and subsequent sections, 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 Fig. 10, it



**Fig. 10.** Number of microseconds required to compute a method at a call-site

can be seen that MRD provides the fastest dispatch time on both platforms, and did so for all five platforms tested.<sup>4</sup> 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.

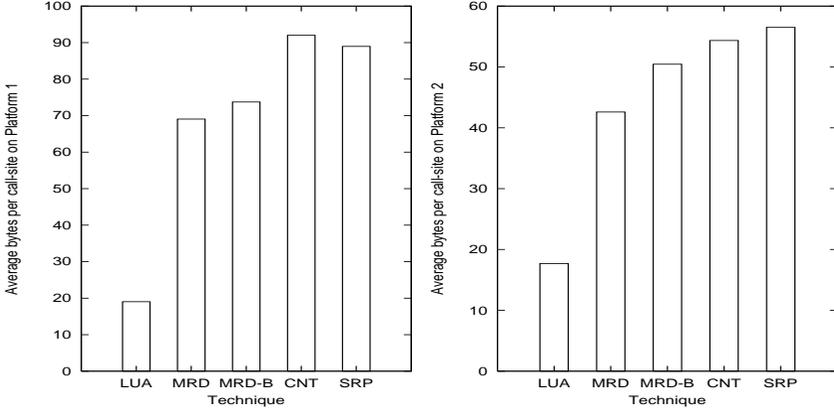
### 6.3 Memory Utilization

We can divide memory usage 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 application dependent.

In order to compare the call-site size of the various techniques, we generated another set of technique-specific C++ programs. For each technique, a program

<sup>4</sup> 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

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 generate arguments happen to be identical, or if the constants in the dispatch computation happen to be less than 256 or less than 65535, allowing them to be stored using smaller instructions. Fig. 11 shows the number of bytes required by the call-site



**Fig. 11.** Call-Site Memory Usage

dispatch code. Similar results are returned from higher arity behaviors. Since the data-structure size is dependent on an application, we chose to measure the size required to maintain information for all types and all behaviors in the Cecil Vortex3 (Cecil compiler [20]) 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, we filtered the set of all possible behaviors to arrive at the set of behaviors that truly require multi-method dispatch. In particular, we 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 our data assumes a statically-typed language, we ignore behaviors with only one method defined on them, since they too can be determined at compile-time. Finally, for each remaining behavior, we 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  $\sigma$

# Arity	# Behavior
2	203
3	22
4	11
Method Count	# Behavior
2	53
3	33
4	35
5-8	41
9-15	14
16-32	12
33+	4

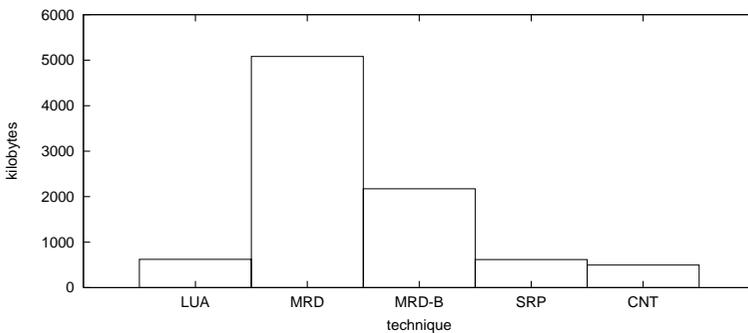
(a) Cecil Vortex3 Type Hierarchy

# Arity	# Behavior
2	95
3	13
Method Count	# Behavior
2	21
3	11
4	32
5-8	23
9-15	12
16-32	7
33+	2

(b) Harlequin Type Hierarchy

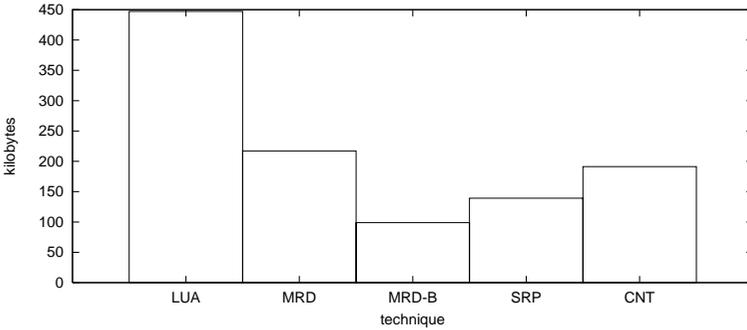
**Fig. 12.** Type Hierarchy Details for Two Different Hierarchies

is defined only on  $A \times A$ ,  $B \times A$  and  $C \times A$ , then no dispatch on the second argument is required (because we are assuming statically typed languages). By reducing behaviors down to the set of arguments upon which dispatch is truly required, we 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 Fig. 12. The data-structure memory usage for each technique is shown in Fig. 13. In these reduced Cecil Vortex3 and Dylan hierarchies, many

**Fig. 13.** Static Data Structure Memory Usage for Cecil Vortex3

of the method definitions have arguments typed as the *root-type*. Whenever an argument is typed as the *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 we 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 Fig. 13. As multi-methods become more common, we 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 Fig. 14. The results for the Dylan hierarchy are similar.



**Fig. 14.** Static Data Structure Memory Usage for *No Root-Typed* Cecil Vortex3

## 7 Future Work

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 we can choose the implementation with the execution and memory footprint that most closely satisfies our application requirements. 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.

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.

## 8 Conclusion

We have presented Multiple Row Displacement (MRD), a new multi-method dispatch technique that 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, we have provided the first performance comparison of the existing table-based multi-method dispatch techniques.

## References

1. Holst, W., Szafron, D.: A General Framework For Inheritance Management and Method Dispatch in Object-Oriented Languages, ECOOP'97.
2. Chambers, C.: Object-Oriented Multi-Methods in Cecil, ECOOP'92 Conference Proceedings, 1992.
3. Bobrow, B., DeMichiel, D., Gabriel, R., Keene, S., Kiczales G., Moon, D.: Common Lisp Object System Specification, June 1988, X3J13 Document 88-002R.
4. Dylan Interim Reference Manual, Apple Computer, Inc., 1994.
5. Cox, B.: Object-Oriented Programming, An Evolutionary Approach, Addison-Wesley, 1987.
6. Driesen, K., Hölzle, U., Vitek, J.: Message Dispatch on Pipelined Processors, ECOOP'95 Conference Proceedings, 1995.
7. Driesen, K., Hölzle, U.: Minimizing Row Displacement Dispatch Tables, OOP-SLA'95 Conference Proceedings, 1995.

8. Driesen, K.: Selector Table Indexing and Sparse Arrays, OOPSLA'93 Conference Proceedings, 1993.
9. Amiel, E., Gruber, O., Simon, E.: Optimizing Multi-Method Dispatch Using Compressed Dispatch Table, OOPSLA'94 Conference Proceedings, 1994.
10. Dujardin, E., Amiel, E., Simon, E.: Fast Algorithms for Compressed Multi-Method Dispatch Table Generation, TOPLAS'96 Conference Proceedings, 1996.
11. Dujardin, E., Amiel, E., Simon, E.: Fast Algorithms for Compressed Multi-Method Dispatch Table Generation, TOPLAS Journal, vol. 20, no. 1, Jan 1998, p116-165.
12. Chen, W., Turau, V., Klas, W.: Efficient Dynamic Look-up Strategy for Multi-methods, ECOOP'94 Conference Proceedings, 1994.
13. Chen, W.: Efficient Multiple Dispatching Based on Automata, Darmstadt, Germany, 1995.
14. Holst, W., Szafron, D., Leontiev, Y., Pang, C.: Multi-Method Dispatch Using Single-Receiver Projections, TR-98-03, University of Alberta, Edmonton, Alberta, Canada, 1998.
15. Özsü, T., Peters, J., Szafron, D., Irani, B., Lipka, A., Muñoz, A.: TIGUKAT: A Uniform Behavioral Objectbase Management System, VLDB'95 Conference Proceedings, 1995.
16. Vitek, J., Nigel Horspool, R.: Compact Dispatch Tables for Dynamically Typed Programming Languages, CC'96 Conference Proceedings, 1996.
17. Krall, A., Vitek, J., Nigel Horspool, R.: Near Optimal Hierarchical Encoding of Types, ECOOP'97 Conference Proceedings, 1997.
18. Pang, C.: Multi-Method Dispatch Using Multiple Row Displacement, thesis, University of Alberta, 1999.
19. Chambers, C., Chen, W.: Efficient Predicate Dispatching, Technical Report UW-CSE-98-12-02, Department of Computing Science and Engineering, University of Washington.
20. Chambers, C.: Object-oriented multi-methods in Cecil, ECOOP'92 Conference Proceedings, 1992.