University of Alberta

Multi-Method Dispatch Using Multiple Row Displacement

by

Candy Pang Wade Holst Yuri Leontiev Duane Szafron

Technical Report TR 98–12
November 1998

DEPARTMENT OF COMPUTING SCIENCE
University of Alberta
Edmonton, Alberta, Canada

Multi-Method Dispatch Using Multiple Row Displacement

Candy Pang
Wade Holst
Yuri Leontiev
Duane Szafron
{candy,wade,yuri,duane}@cs.ualberta.ca

Department of Computing Science
University of Alberta
Edmonton, Canada

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 has faster dispatch performance and uses less space than any of these techniques.

RESEARCH PAPER

Subjects: language design and implementation
Keywords: multi-method, dispatch, object-oriented, implementation
Word Count: 7576

This paper has been submitted to ECOOP'99.

1 Introduction

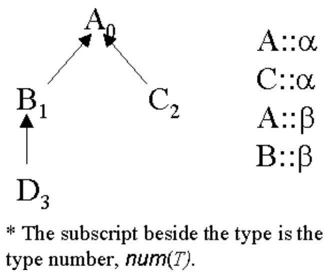
Object-oriented languages can be separated into *single-receiver languages* and *multi-method languages*. *Single-receiver languages* use the dynamic type of a dedicated *receiver* object in conjunction with the method name to determine the method to execute at run-time. *Multi-method languages* use the dynamic types of one or more arguments[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 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 ([HS97]). However, there are some multi-method languages in use, such as Cecil ([Cha92]), CLOS ([BDG$^+$88]), and Dylan ([App94]). By looking at the distribution of methods in such languages, it is interesting to note that almost all behaviors have less than four arguments. For example, in the Cecil hierarchy ([Cha92]), 54% of the behaviors have one or no arguments; 90% have two or fewer arguments; and 96% have three or fewer arguments. Behaviors with no arguments do not need method dispatch. Many efficient 1-arity (single-receiver) dispatch techniques already exist. Therefore, the most important research problem in multi-method language dispatch is how to efficiently dispatch 2 and 3-arity behaviors.

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 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.

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, and is especially time and space efficient for 2-arity and 3-arity methods. Its execution

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



A_0
B_1
C_2
D_3
A::α
C::α
A::β
B::β
* The subscript beside the type is the type number, *num*(*T*).

(a) Type Hierarchy

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

(b) Code Requiring Method Dispatch

Figure 1: An example hierarchy and program segment requiring method dispatch

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

2 Terminology for Multi-Method Dispatch

2.1 Notation

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

Expression 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_2 and type T_1 if T_2 is a *direct subtype* of T_1, which we denote as $T_2 \vec{\prec} T_1$. If T_1 can be reached from T_2 by following one or more supertype edges, T_2 is a *subtype* of T_1, denoted as $T_2 \prec T_1$.

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 Figure 1 (a), and the code in Figure 1 (b). 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.

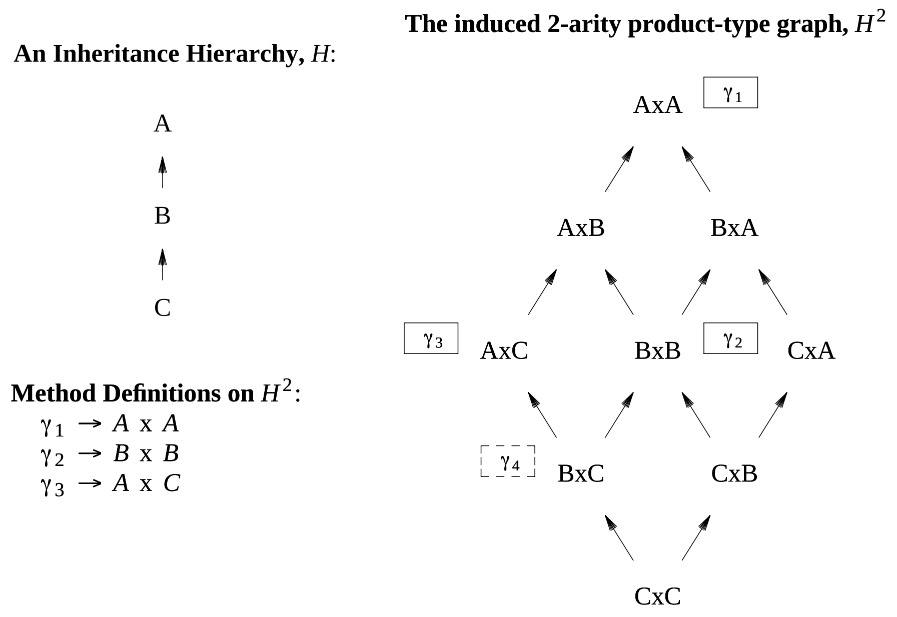

An Inheritance Hierarchy, H:
A
B
C
Method Definitions on H^2:
$\gamma_1 \rightarrow A$ x A
$\gamma_2 \rightarrow B$ x B
$\gamma_3 \rightarrow A$ x C
The induced 2-arity product-type graph, H^2
AxA
γ_1
AxB
BxA
γ_3
AxC
BxB
γ_2
CxA
γ_4
BxC
CxB
CxC

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

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 $P = T^1 \times T^2 \times ... \times T^k$. 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 describes when a directed edge exists from a child product-type $P_2 = T_2^1 \times T_2^2 \times ... \times T_2^k$ to a parent product-type $P_1 = T_1^1 \times T_1^2 \times ... \times T_1^k$, which is denoted $P_2 \vec{\prec} P_1$.

$$P_2 \vec{\prec} P_1 \Leftrightarrow \exists i, 1 \leq i \leq k : T_2^i \vec{\prec} T_1^i \wedge \forall j \neq i, T_2^j = T_1^j \tag{2}$$

The notation $P_2 \prec P_1$ indicates that P_2 is a *sub-product-type* of P_1, which implies that P_1 can be reached from P_2 by following edges in the product-type graph $\mathcal{H}^k$. Figure 2 presents a sample inheritance hierarchy $\mathcal{H}$ and its induced 2-arity product-type graph, $\mathcal{H}^2$. To this end, 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 [2]. Note that the product-type hierarchies, $\mathcal{H}^2$, $\mathcal{H}^3$, ..., 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 Expression 2.

We next define the concept of a *behavior*. Behaviors are denoted by $\mathcal{B}_\sigma^k$, and have a name, $\sigma =$ *name*$(\mathcal{B}_\sigma^k)$, and an arity, $k = arity(\mathcal{B}_\sigma^k)$. The maximum arity for all behaviors in the system is denoted as K. Multiple methods can be defined for each behavior. A method for a behavior named

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

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 be thought of 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 [3].

2.2 Inheritance Conflicts

In single-receiver languages with multiple inheritance, the concept of *inheritance conflicts* 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 so that an inheritance conflict is not considered to have occurred if $T_i \prec T_j$ or $T_j \prec T_i$, since one of the types is "closer" to T.

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, $\mathcal{H}^k$. 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, 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. In multi-method languages, it is especially important to use the more relaxed definition of inheritance conflict (where a conflict does not occur if one of the conflicting product-types is a child of the other). Otherwise, a large number of inheritance conflicts would be generated for almost every method definition.

3 Single-Receiver Row Displacement Table Dispatch (RD)

S	A_0	B_1	C_2	D_3
α	A::α	--	C::α	--
β	A:: β	--	B::β	B::β

Figure 3: Selector Table, S

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 shows the selector table for the type hierarchy and method definitions in Figure 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 ([Cox87]). This algorithm is known as STI in the literature [DHV95].

[3]In single-receiver languages, the first argument is called as receiver. However, in multi-method languages the first argument is only one of the arguments.

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 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 ([DH95]), so the table can be compressed.

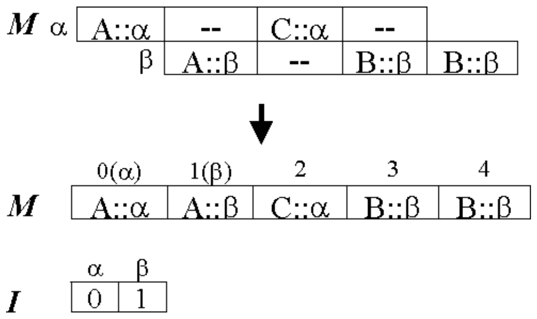


Figure 4: Compressing A Selector Table By Row Displacement

Row displacement (RD) reduces the number of empty entries by compressing the two-dimensional selector table into a one-dimensional array ([DH95, Dri93]). As illustrated in Figure 4, 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 β with D as the dynamic type of the receiver, the shift index for β 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 B::β. Compared with other single-receiver table dispatch techniques, row displacement is highly space and time efficient ([HS97]). We will show how this single-receiver technique can be generalized to multi-method languages in Section 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* ([AGS94, DAS96]) 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. Although this approach provides efficient dispatch, the type-to-group arrays tend to take up a substantial amount of space.

2. *LUA: Lookup Automata* ([CTK94, Che95]) 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). This technique is O(k) but its overall speed performance is inversely proportional to the memory used, based on an implementation parameter (the smaller this parameter, the better the dispatch performance, but the more memory used).

3. *SRP: Single-Receiver Projections* ([HSLP98]) 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 left-most on-bit and returning the method associated with this index. It is much more space efficient than CNT, but has slower dispatch time.

4. *Extended Cache-Based Techniques* are used in Cecil ([Cha92]). The cache-based techniques from single-receiver languages ([DHV95]) are extended to work for product-types instead of just simple types.

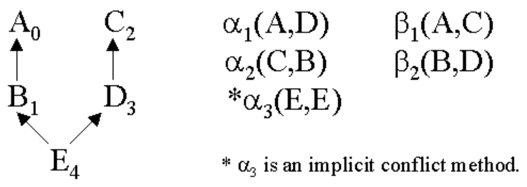
5 Multiple Row Displacement (MRD)

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 idea of the selector table in single-receiver dispatch can be extended to handle multi-method dispatch. In multi-method dispatch, each k-arity behavior, $\mathcal{B}^k_\sigma$, will have a k-dimensional dispatch table, D^k_σ, with type numbers as indices for each dimension. Therefore, each k-dimensional dispatch table will have size $|\mathcal{H}|^k$. 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 $T^i = type(o_i)$.

For example, the 2-dimensional dispatch tables for the type hierarchy and method definitions in Figure 5 (a) are shown in Figure 5 (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, we define an implicit conflict method α_3, and insert it into the table at $E \times E$.



(a)

D^2_α — 2nd Argument (rows: 1st Argument)

	A_0	B_1	C_2	D_3	E_4
A_0	--	--	--	α_1	α_1
B_1	--	--	--	α_1	α_1
C_2	--	α_2	--	--	α_2
D_3	--	α_2	--	--	α_2
E_4	--	α_2	--	α_1	α_3

D^2_β — 2nd Argument (rows: 1st Argument)

	A_0	B_1	C_2	D_3	E_4
A_0	--	--	β_1	β_1	β_1
B_1	--	--	β_1	β_2	β_2
C_2	--	--	--	--	--
D_3	--	--	--	--	--
E_4	--	--	β_1	β_2	β_2

(b)

Figure 5: N-Dimensional Dispatch Tables

N-dimensional table dispatch is very time efficient. 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, the space required for all 2 and 3-dimensional tables in the Cecil hierarchy would be more than 9 Gbytes.

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 first example uses the type hierarchy and 2-arity method definitions from Figure 5 (a).

Instead of the single k-dimensional array shown in Figure 5 (b), each table can be represented as an array of arrays as shown in Figure 6 (a). The arrays indexed by the first arguments are called $level$-0 arrays, L_0. There should be 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 6 (a) that some of the $level$-1 arrays are exactly the same. Those arrays are combined as shown in Figure 6 (b). In general, there will be many identical rows in an n-dimensional dispatch table. As well, there will be many rows for which all entries are empty (i.e. do not correspond to any method). These observations are the basis for the CNT dispatch technique mentioned in Section 4, and are also one of the underlying reasons for the compression provided

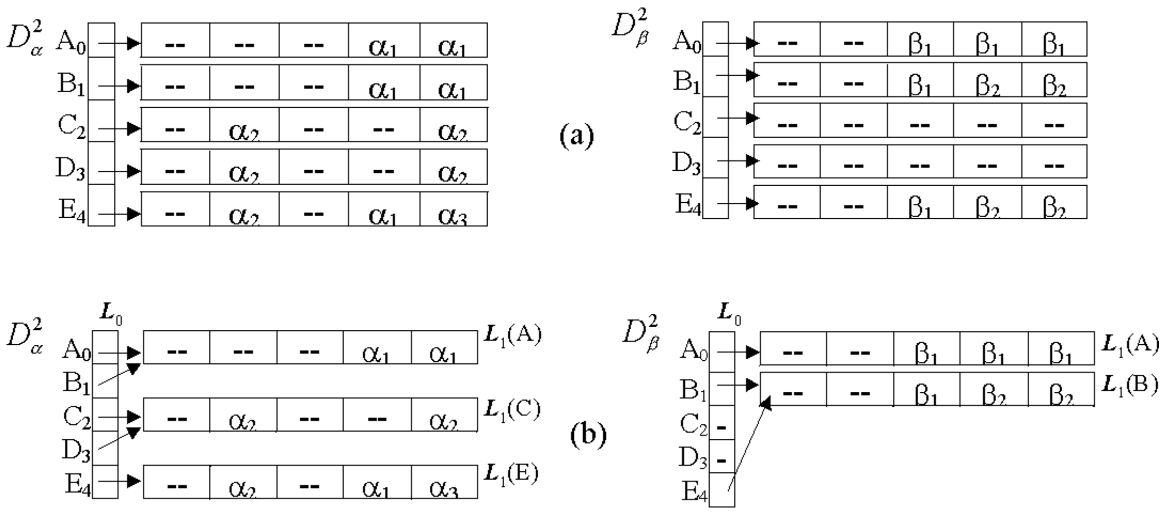


Figure 6: Data Structure for Multiple Row Displacement

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 invoked the same methods for all types, and although languages like Tigukat ([OPS$^+$95]) 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) ([VH96]).

We have one data structure per behavior, D^k_σ, 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^2_α in Figure 6 (b), the $level$-1 array $L_1(A)$ is first shifted into the Global Master Array, M, by row displacement, as shown in Figure 7 (a). 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. Figure 7 (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. Finally, as shown in Figure 7 (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 8 shows how to compress the behavior data structure, D^2_β,

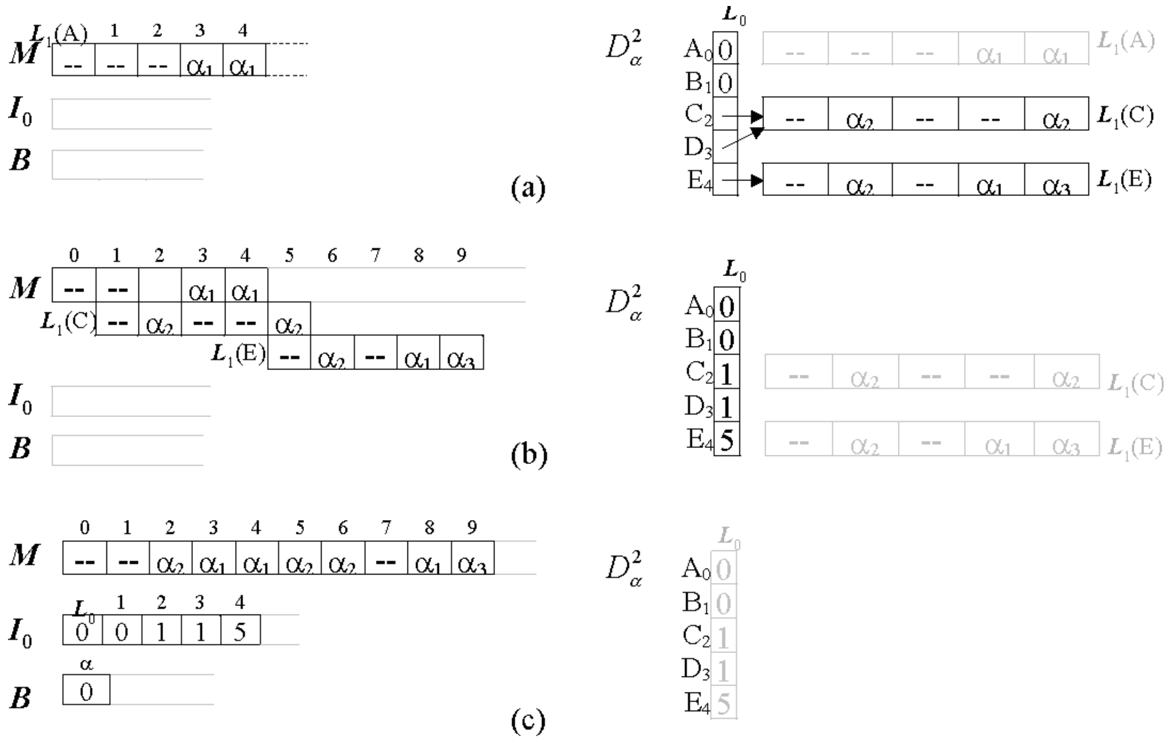


Figure 7: Compressing The Data Structure for α

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 8 (a). The compression of the $level$-0 array, L_0, is shown in Figure 8 (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.

As an example of dispatch, we will demonstrate how to dispatch a call-site $\beta(anE, aD)$ using the data structures in Figure 8 (b). The method dispatch starts by obtaining the shift index of the behavior, β, from the Global Behavior Array, B. From Figure 8 (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 9 (a) shows the method definitions of a 3-arity behavior, δ, and Figure 9 (b) shows its preliminary behavior data structure, D^3_δ. Figures 9 (c) to (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 9 (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 9 (b) that pointed to

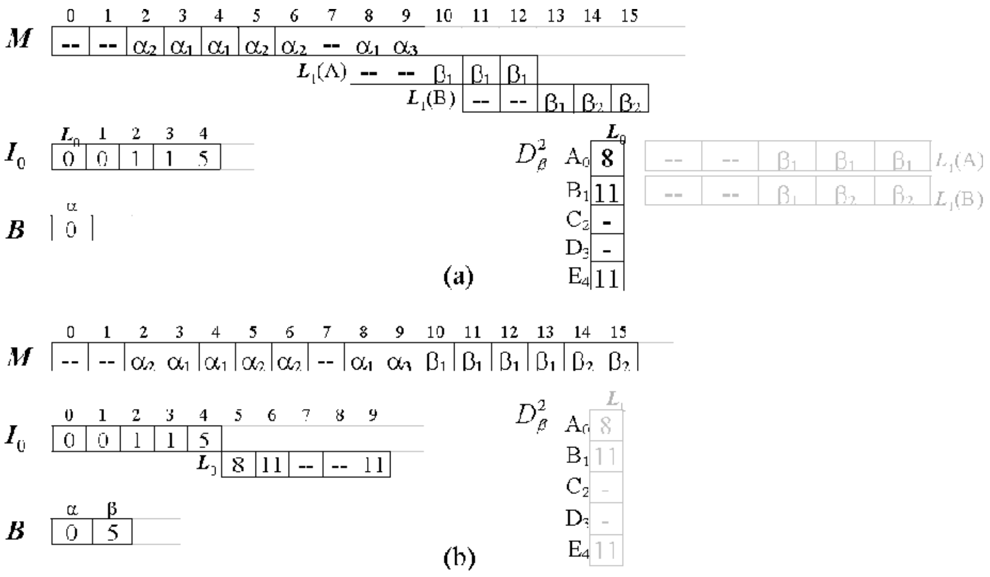


Figure 8: Compressing The Data Structure For β

$L_2(B \times D)$ 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 9 (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 9 (e).

5.3 The Multiple Row-Displacement Dispatch 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^k_σ, is created for each behavior. D^k_σ is a data structure for a k-arity behavior named σ, as shown in Figure 9 (b). It is actually an n-dimensional dispatch table, which is an array of pointers to arrays. Each array in D^k_σ 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 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^k_σ 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

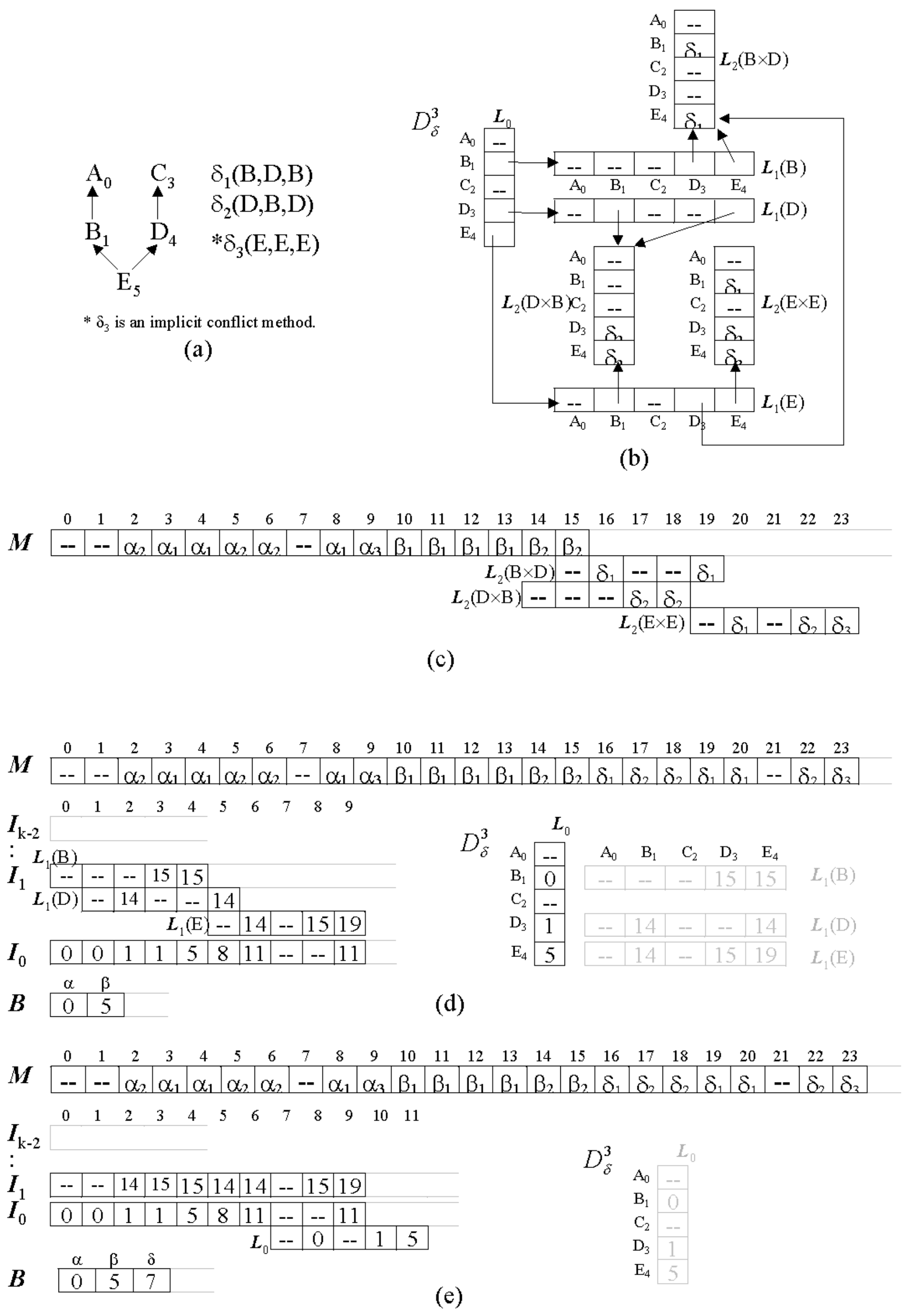


Figure 9: Compressing The Data Structure For δ

$B[\sigma]$. The whole process is repeated for each behavior. The algorithm to compress all behavior data structures is shown in Appendix A.

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

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

As an example of dispatch with Expression 3, we will demonstrate how to dispatch a call-site $\delta(anE, aD, aB)$ using the data structures in Figure 9 (e). Since δ is a 3-arity behavior, Expression 3 becomes Expression 4.

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

Substituting the data from Figure 9 (e) into Expression 4 yields the method δ_1, as shown in Expression 5.

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

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 we have ignored this final compression.

6 Performance Results

Here we present memory and execution results for the new technique, MRD, and three other published techniques, CNT, LUA and SRP. When analyzing dispatch techniques, 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. To our knowledge, this is the first time such a comparison of the techniques has appeared in the literature.

The rest of this section is organized into four 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. The fourth subsection presents some observations on how MRD can be made even more space efficient. The source code for each technique was compiled with 'g++ -O2' and run on an UltraSparc1 with 128 Mbytes of memory using Solaris 2.6.

6.1 Data Structures and Dispatch Code

This section provides a brief description of the data-structures that each of the four dispatch techniques requires 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 $\sigma(\ o_1,\ o_2,\ ...,\ o_k\ )$.

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 type structure that contains a 'typeNumber' field.

6.1.1 MRD

MRD has an M array that stores function addresses, a B array that stores behavior shift indices, and $K-1$ index arrays, I_0, ..., I_{K-2}.

The dispatch sequence is given in Expression 6.

$$\begin{aligned}&(\ *(M[\ I_{k-2}[\ ...I_1[\ I_0[b^\sigma + N(o_1)\ ]\\ &\qquad +N(o_2)\ ] + ...\ ] + N(o_{k-1})\ ] + N(o_k)\ ])\ )(o_1, o_2, ..., o_k)\end{aligned} \tag{6}$$

Note that the Global Behavior Array, B, from Expression 3, is known at compile-time, so $B[\sigma]$ can be precomputed. The resulting constant that is denoted as b^σ.

6.1.2 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_σ^{CNT}.

In addition to the behavior-specific information, CNT requires arrays that map types to type-groups. These group arrays can be compressed by merging multiple groups into a single group using 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, ..., 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 σ has dimensions $n_1, n_2, ..., n_k$, where the n_i values are behavior specific, and that the group arrays for these dimensions are $G_{i_1}, G_{i_2}, ..., G_{i_k}$ then the call-site dispatch code is given in Expression 7.

$$\begin{aligned}(*\,(D_\sigma^{CNT}[\quad & G_{i_1}[N(o_1)] \\ + \;& G_{i_2}[N(o_2)] \times n_1 \\ + \;& ... \\ + \;& G_{i_k}[N(o_k)] \times n_1 \times n_2 \times ... \times n_{k-1}\;\;]\;)\;)\;(o_1, o_2, ...o_k)\end{aligned} \qquad (7)$$

Note that since the n_i are known constants, the products of the form , $n_1 \times ... \times n_j$, can be precomputed. Thus, only $k-1$ multiplications are required as run-time.

6.1.3 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 techniques, 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_σ^{SRP}.

The dispatch code for SRP is given in Expression 8, where FirstBit() is some macro or function that implements the operation of finding the position of the first '1' bit in a bit-vector. [HSLP98] discusses this in some detail. Our timing and space results assume that this is a hardware-supported operation with the same performance as modulo.

$$\begin{aligned}(\;*(D_\sigma^{SRP}[\;FirstBit(& S_1[N(o_1) + b_1^\sigma]\;\& \\ & S_2[N(o_2) + b_2^\sigma]\;\& \\ & ...\;\& \\ & S_k[N(o_k) + b_k^\sigma])\;]\;)\;)\;(o_1, o_2, ..., o_k)\end{aligned} \qquad (8)$$

Note that b_i^σ is the shift index assigned to behavior σ in argument-table i and is a compile-time constant.

6.1.4 LUA

LUA is, in some ways, the most difficult technique to evaluate accurately. First, [Che95] does not provide any explicit description of what the code at a particular call-site would look like. Second, there are a number of variations possible during implementation, that have vastly different space vs. time performance results. The most naive implementation (referred to here as simply LUA) is highly space efficient, but has a dispatch time that is proportional to the method count (number of methods defined for the behavior). The performances of MRD, CNT and SRP are independent of method count. A more complex implementation, which we will denote LUA/c, reduces this dependency on method count and improves dispatch performance at the expense of additional memory. The space-time tradeoff is controlled by a parameter, c. When $c = 1$, LUA provides constant time dispatch with respect to method count, but uses prohibitive amounts of memory. For higher values of c, the technique varies between the extremes of LUA and LUA/c.

For the purposes of the comparisons here, we have chosen to use the naive approach, for a number of reasons. First, as mentioned previously, no explicit discussion of call-site code is presented, so the discussion here represents our own implementation of data structures described in [Che95]. Second, describing static data-structures and dispatch code for LUA/c would require an entire paper. Third, for low method counts, LUA and LUA/c have similar dispatch performance and extremely different memory utilization, favoring LUA over LUA/c.

Conceptually, a lookup automaton consists of a root node with a set of edges to other nodes, which in turn have edges to other nodes (or to method addresses). Edges between nodes are labeled with types. If a dynamic type is a subtype of the type associated with an edge, then the edge can be chosen. As described in [Che95], we can order the edges of a node by their associated types in a bottom-up fashion (any ordering in which a subtype is guaranteed to occur before any of its supertypes). In this way, each node can be represented as an array of integer-pointer pairs, where the integer represents a type number, and the pointer refers to either a method address, or another array of integer-pointer pairs.

Thus, for a behavior, we must maintain a collection of arrays of integer-pointer pairs. Each behavior has an array of such pairs, identified by some behavior-specific name, D_{σ}^{LUA}. This array can be fully initialized at compile-time. In order to avoid using the heap, the subarrays are also allocated statically on the stack.

In addition to the behavior-specific data, LUA requires data-structures for performing sub-type testing. Although techniques like [KVH97] provide extremely space efficient implementations, we decided to optimize time instead of space, so we implemented the sub-type testing in a naive fashion. Each type, T, has an associated array, T_s, that is indexed by type numbers. The i^{th} element of T_s stores a boolean indication of whether type i is a subtype of T_s. A few simple variations were tested for their impact on space and time. Using bitvectors is space efficient, but requires that for type i we perform several additional logical operations[4]. Using an array of bytes avoids these operations at the cost of 8 times more space. We have presented results here for an implementation using bitvectors, since it dramatically improves memory usage and has very little impact on dispatch efficiency on the architectures we tested.

Given the D_{σ}^{LUA} and T_s arrays, the dispatch code for LUA is:

[4]The actual code becomes $T[i >> 3] \& (1 << (i\%8))$ instead of just $T[i]$.

```
typedef struct {
      LUAPairOrAddr * next;
      int                        type;
} LUAPairOrAddr;

register int i;
register LUAPairOrAddr * P;

// Follow appropriate edge from root
i = 0; label1: if (!(($T_s$[ $D_\sigma^{LUA}$[i].type ])[ $N(o_1)$ ])) ++i; goto label1;

// Obtain the level-1 node
P = $D_\sigma^{LUA}$[i].next;
i = 0; label2: if (!(($T_s$[ P[i].type ])[ $N(o_2)$ ])) ++i; goto label2;

...

// Obtain the level-(k-1) node
P = P[i].next;
i = 0; labelk: if (!(($T_s$[ P[i].type ])[ $N(o_k)$ ])) ++i; goto labelk;

// Call the method
( *(P[i].next) )($o_1, o_2, ..., o_k$);
```

As mentioned previously, the original LUA paper did not give specific optimizing implementation details, and we have provided only one possible solution. We mention in passing that LUA provides information that is required by a much more efficient mechanism for dispatch, but this information can be derived more easily than by the mechanisms suggested by LUA. This new technique will be the subject of a forthcoming paper.

6.2 Timings Results

The most critical aspect of a method dispatch technique is the amount of time it takes to compute the address of the method to invoke at a particular call-site.

In this paper, we use a simple mechanism for obtaining the timing results for a particular dispatch technique. We wrap the call-site code in a loop and execute it ten million times. We do this in order to obtain enough computation to exceed the resolution of the timing facilities, which in our case was *getrusage* [5]. After obtaining a total time, we subtract from it the amount of time needed to execute an empty loop 10 million times, and divide by 10 million to get a measure of the time to dispatch a single call-site.

Although MRD, SRP and CNT always require the same amount of time for behaviors of a given arity [6], the performance of LUA is dramatically affected by the average number of comparisons that

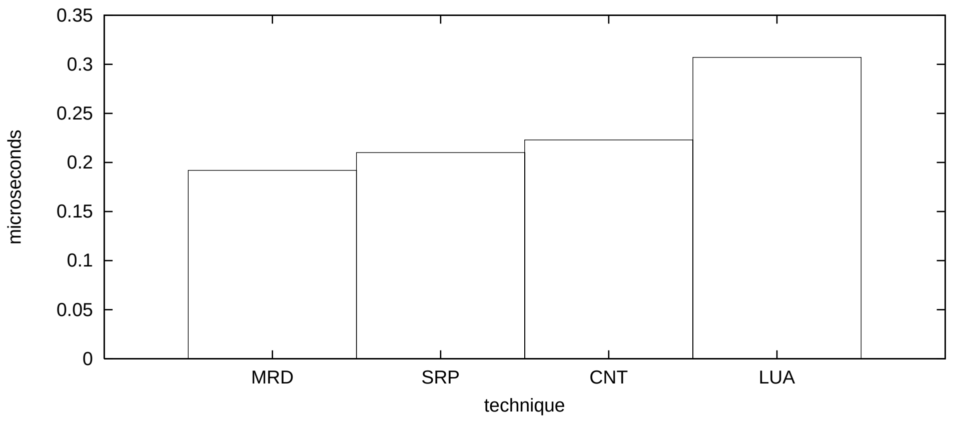


Figure 10: Number of microseconds required to compute a method at a call-site

occur before successful matches occur at each level of search. Our results here are for the fastest possible case, when the match occurs on the first comparison at each level.

From Figure 10, MRD provides the fastest call-site dispatch, at 0.19 μs, followed by SRP at 0.21 μs, then CNT at 0.22 μs, and finally LUA at 0.30 μs.

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.

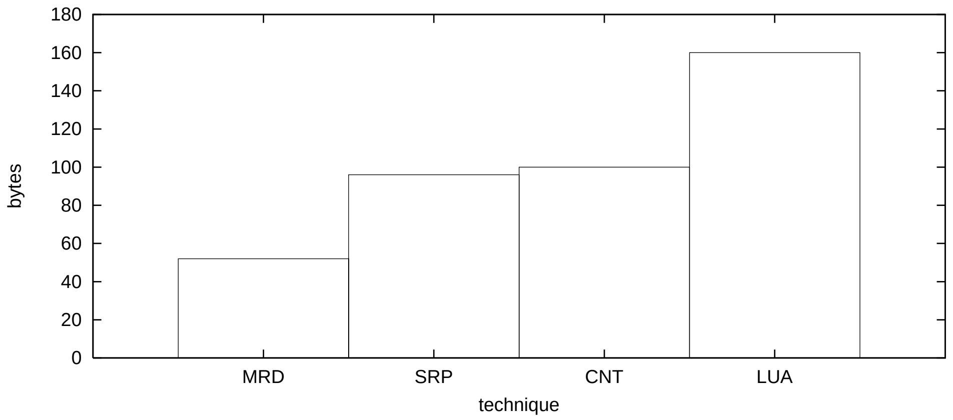


Figure 11: Call-Site Memory Usage

Figure 11 shows the number of bytes required by the call-site dispatch code. MRD requires 52 bytes, SRP requires 96 bytes, CNT requires 100 bytes and LUA requires 160 bytes.

[5]The *getrusage* routine has a resolution threshold of 10,000 microseconds on Solaris machines.

[6]In making this observation we ignore fortuitous optimizations like the ability to change the multiply operation in CNT

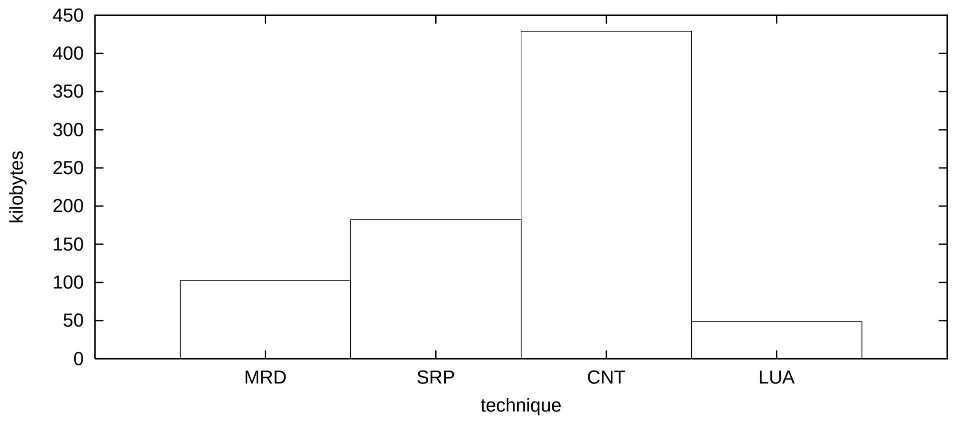


Figure 12: Static Data Structure Memory Usage for all 2-arity Cecil Methods

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 2-arity behaviors in the basic Cecil library ([Cha92]). It consists of 472 classes and 741 2-arity behaviors. The results are shown in Figure 12. LUA requires the least amount of space, at 48 Kbytes, as compared to MRD at 102 Kbytes, SRP at 182 Kbytes and CNT at 428 Kbytes.

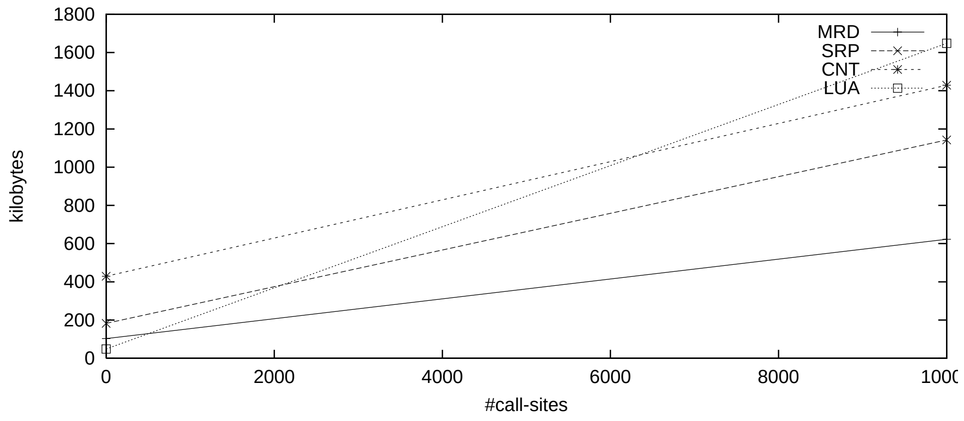


Figure 13: Total Memory for Cecil 2-Arity Methods

The total memory needed for method dispatch at run-time is the memory used by the static data structures plus the memory used by each call-site times the number of call-sites. Figure 13 shows the impact of the number of call-sites on overall memory consumption. Although LUA has very small data-structure requirements, its large call-site code size quickly makes it less efficient than MRD. For any application with more than 498 call-sites, MRD is more space-efficient than LUA.

Given the dominating impact of call-site code size on memory usage, it is natural to ask whether it is wise to generate such code at every call-site. An alternative approach would be to generate a function that represents the dispatch code for each behavior, so that the call-site code becomes

dispatch to a shift operation when the constant is a power of two simply a call to the appropriate dispatch function. This is a viable alternative but does incur a time penalty due to the extra function call. Note that MRD provides efficient memory utilization and fast dispatch even when call-site code is generated for every call-site.

6.4 Optimizations to MRD to further improve space utilization

The row-shifting mechanism used in our implementation of row displacement is not the most space-efficient technique possible. If we replace our row-shifting algorithm by a more general algorithm based on string matching, we will get a higher compression rate. For example, using row-shifting to compress rows R1 and R2 in Figure 14 (a) produces a master array with 9 elements as shown in Figure 14 (b). However, using the improved algorithm to compress R1 and R2 produces a master array with only 6 elements as shown in Figure 14 (c). Our improved algorithm cannot be used in single-receiver row displacement, since different rows contain different selectors. The memory results in this paper do not include the improved algorithm.

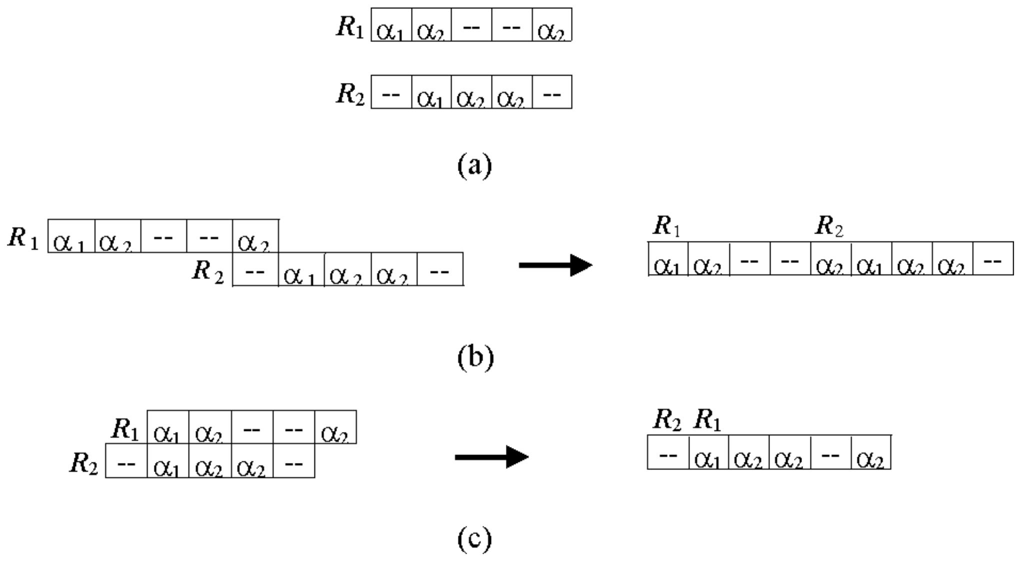


Figure 14: Different Row Displacement Techniques

7 Future Work

The research into 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. In particular, LUA has a great deal of room for different 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. Rather than placing such code at every call-site, it is possible to define a single dispatch function for each behavior. Since the number of behaviors is much less than the number of call-sites, this will provide a substantial savings in code size, at the expense of an extra function call.

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

This paper has 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 smallest per-call-site code size.

In addition to presenting the new technique, we have provided the first performance comparison of the existing table-based multi-method dispatch techniques.

References

[AGS94] Eric Amiel, Olivier Gruber, and Eric Simon. Optimizing multi-method dispatch using compressed dispatch tables. In *OOPSLA'94 Conference Proceedings*, 1994.

[App94] Apple Computer, Inc. *Dylan Interim Reference Manual*, 1994.

[BDG$^+$88] 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.

[Cha92] Craig Chambers. Object-oriented multi-methods in Cecil. In *ECOOP'92 Conference Proceedings*, 1992.

[Che95] Weimin Chen. Efficient multiple dispatching based on automata. Master's thesis, Darmstadt, Germany, 1995.

[Cox87] Brad Cox. *Object-Oriented Programming, An Evolutionary Approach*. Addison-Wesley, 1987.

[CTK94] Weimin Chen, Volker Turau, and Wolfgang Klas. Efficient dynamic look-up strategy for multimethods. In *ECOOP'94 Conference Proceedings*, 1994.

[DAS96] Eric Dujardin, Eric Amiel, and Eric Simon. Fast algorithms for compressed multimethod dispatch table generation. In *Transactions on Programming Languages and Systems*, 1996.

[DH95] K. Driesen and U. Holzle. Minimizing row displacement dispatch tables. In *OOPSLA'95 Conference Proceedings*, 1995.

[DHV95] K. Driesen, U. Holzle, and J. Vitek. Message dispatch on pipelined processors. In *ECOOP'95 Conference Proceedings*, 1995.

[Dri93] Karel Driesen. Selector table indexing and sparse arrays. In *OOPSLA'93 Conference Proceedings*, 1993.

[HS97] Wade Holst and Duane Szafron. A general framework for inheritance management and method dispatch in object-oriented languages. In *ECOOP'97 Conference Proceedings*, 1997.

[HSLP98] 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.

[KVH97] Andreas Krall, Jan Vitek, and R. Nigel Horspool. Near optimal hierarchical encoding of types. In *ECOOP'97 Conference Proceedings*, 1997.

[OPS$^+$95] 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.

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

A Algorithm To Compress Behavior Data Structure

```
// Shift the given array into this array by row displacement, and return the shift index.
// The shift index will also be stored in the given array.
shiftIndex Array::add( Array );

// Get the shift index of this array. If this array has never been shifted
// into any other array, return -1.
shiftIndex Array::getShiftIndex();

// Create a n-dimensional table for the behavior.
BehaviorStructure Behavior::createStructure();

Array $M$, $I_0$, ..., $I_{K-2}$, $B$;

createGlobalDataStructure() begin
    for( each behavior $B^k_\sigma$ ) do
        BehaviorStructure $D^k_\sigma$ = $B^k_\sigma$.createStructure();
        createRecursiveStructure( $D^k_\sigma.L_0$, 0 );
        $B[\sigma] = I_0.add(D^k_\sigma.L_0)$;
    endfor
end

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

    else

        for( int i=0, i<L.size(); i++ ) do
            if( L[i] == null ) then
                continue;
            elseif( L[i].getShiftIndex() == -1 ) then
                createRecursiveStructure(L[i], level+1 );
                L[i] = $I_{level+1}$.add( L[i] );
            else
                L[i] = L[i].getShiftIndex();
            endif
        endfor
    endif
end
```