Incremental Algorithms for Dispatching in Dynamically Typed Languages[*]

Yoav Zibin Joseph (Yossi) Gil[†]

Technion—Israel Institute of Technology
`zyoav | yogi @ cs.technion.ac.il`

ABSTRACT

A fundamental problem in the implementation of object-oriented languages is that of a frugal *dispatching data structure*, i.e., support for quick response to dispatching queries combined with compact representation of the type hierarchy and the method families. Previous theoretical algorithms tend to be impractical due to their complexity and large hidden constant. In contrast, successful practical heuristics, including Vitek and Horspool's *compact dispatch tables* (CT) [16] designed for dynamically typed languages, lack theoretical support. In subjecting CT to theoretical analysis, we are not only able to improve and generalize it, but also provide the first non-trivial bounds on the performance of such a heuristic.

Let n, m, ℓ denote the total number of types, messages, and different method implementations, respectively. Then, the dispatching matrix, whose size is nm, can be compressed by a factor of at most $\iota \equiv (nm)/\ell$. Our main variant to CT achieves a compression factor of $\frac{1}{2}\sqrt{\iota}$. More generally, we describe a sequence of algorithms $\text{CT}_1, \text{CT}_2, \text{CT}_3, \ldots$, where CT_d achieves compression by a factor of (at least) $\frac{1}{d}\iota^{1-1/d}$, while using d memory dereferencing operations during dispatch. This tradeoff represents the first bounds on the compression ratio of constant-time dispatching algorithms.

A generalization of these algorithms to a *multiple-inheritance* setting, increases the space by a factor of $\kappa^{1-1/d}$, where κ is a metric of the complexity of the topology of the inheritance hierarchy, which (as indicated by our measurements) is typically small. The most important generalization is an *incremental* variant of the CT_d scheme for a single-inheritance setting. This variant uses at most twice the space of CT_d, and its time of inserting a new type into the hierarchy is optimal. We therefore obtain algorithms for efficient management of dispatching in dynamic-typing, dynamic-loading languages, such as SMALLTALK and even the JAVA `invokeinterface` instruction.

[*]Research supported by the Bar-Nir Bergreen Software Technology Center of Excellence

[†]Research supported in part by the fund for the promotion of research at the Technion.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Run-time environments*; D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Inheritance*; G.4 [**Mathematical Software**]: Algorithm design and analysis

General Terms

Algorithms, Design, Measurement, Performance, Theory

Keywords

CT, Dispatch, Dynamic-typing, Hierarchy, Incremental, Message

1. Introduction

Message dispatching stands at the heart of object-oriented programs, being the only way objects communicate with each other. To implement dynamic binding during dispatch, the runtime system of object-oriented languages uses a *dispatching data structure*, in which a *dispatching query* finds the appropriate implementation of the message to be called, according to the dynamic type of the message receiver. A fundamental problem in the implementation of such languages is then a frugal implementation of this data structure, i.e., simultaneously satisfying (i) compact representation of the type hierarchy and the families of different implementations of each method selector, and (ii) quick response to dispatching queries.

Virtual function tables (VFT) are a simple and well known (see e.g., [14]) incremental technique which achieves dispatching in constant time (two dereferencing operations), and very good compaction rates. The VFT of each type is an array of method addresses. A location in this array represents a message, while its content is the address of an implementing method. The VFT of a subtype is an extension of the VFT of its supertype, and messages are allocated locations at compile time in sequential order. The static type of the receiver uniquely determines the location associated with each message. VFTs rely on *single-inheritance*. *Multiple-inheritance* implementations exist [9], but they are not as elegant or efficient.

The challenge in the dispatching problem is therefore mostly in dealing with *dynamically typed* and/or multiple-inheritance languages. Also very important is the *incremental* version of this problem, in which types (together with their accompanying messages and methods) are added at the bottom of the hierarchy.

Our contribution (described in greater detail in Section 1.3) includes a provable tradeoff between space and dispatching time with extensions to multiple-inheritance hierarchies. The pinnacle of the results is an incremental algorithm for maintaining a compact dispatch table in dynamically typed languages.

1.1 The Problem

We define the dispatching problem in a similar fashion to the *colored-ancestors* abstraction described by Ferragina and Muthukrishnan [7]: a *hierarchy* is a partially ordered set $(\mathcal{T},\preceq)$ where $\mathcal{T}$ is a set of types[1] and $\preceq$ is a reflexive, transitive and anti-symmetric *subtype relation*. The $\min$ operator return the set of smallest types in any given set:

$$\min(X) = \{t \in X \mid \nexists t' \in X : t' \neq t, t' \preceq t\}. \tag{1.1}$$

Let $F \subseteq \mathcal{T}$ denote the *family* of types which have a *method implementation* for the same *message*.[2]

For example, consider the single-inheritance hierarchy in Figure 1.1a. Type names are uppercase and messages are lowercase, e.g., type D implements the messages c, e and f. Then, $\{\mathsf{A}, \mathsf{D}, \mathsf{E}\}$ is the family of method implementations of c.

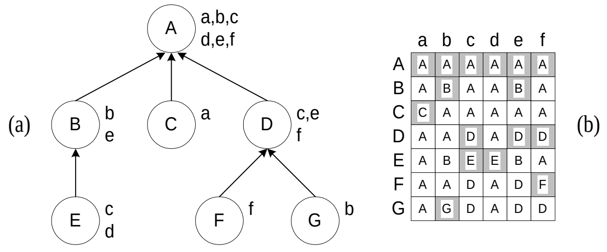


Figure 1.1: (a) A small example of a single-inheritance hierarchy, and (b) its dispatching matrix

Given a family F and a type t, $\mathsf{cand}(F,t)$ is the set of candidates in F, i.e., those ancestors of t in which an implementation of the given message exists:

$$\mathsf{cand}(F,t) \equiv F \cap \mathrm{ancestors}(t). \tag{1.2}$$

In the figure, we have for example $\mathsf{cand}(\{\mathsf{A},\mathsf{D},\mathsf{E}\},\mathsf{G}) = \{\mathsf{A},\mathsf{D}\}$.

A *dispatching query* $\mathsf{dispatch}(F,t)$ returns either *the smallest candidate* or null if no such unique candidate exists. (A null result represents either the *message not understood* or *message ambiguous* error conditions.) Specifically,

$$\mathsf{dispatch}(F,t) \equiv \begin{cases} t' & \text{if } \min(\mathsf{cand}(F,t)) = \{t'\}, \\ \mathrm{null} & \text{otherwise.} \end{cases} \tag{1.3}$$

DEFINITION 1.1. *Given a hierarchy $(\mathcal{T},\preceq)$ and a family collection $\mathcal{F} \subseteq \wp(\mathcal{T})$, the* dispatching problem *is to* encode *the hierarchy in a data structure supporting* $\mathsf{dispatch}(F,t)$ *queries for all $F \in \mathcal{F}, t \in \mathcal{T}$.*

A solution to the dispatching problem is measured by the following three metrics: (i) space, (ii) query time, and (iii) encoding creation time. We would like to express these as a function of the following problem parameters $\langle n, m, \ell\rangle$: the number of types, families, and implementations (or family members). Specifically,

$$\begin{aligned} n &\equiv |\mathcal{T}| \\ m &\equiv |\mathcal{F}| \\ \ell &\equiv \sum_{F\in\mathcal{F}} |F| \end{aligned} \tag{1.4}$$

[1]The distinction between type, class, interface, signature, etc., as it may occur in various languages does not concern us here. We shall refer to all these collectively as *types*.

[2]We abstract away from the nomenclature of different languages, and use the terms *message* (also called selectors or signature) for the unique identifier of a family of *implementation* (also called methods, member functions, operations, features, etc.)

In Figure 1.1 for example, we have $n = 7$, $m = 6$ and $\ell = 16$.

The incremental version of the problem, is to maintain this data structure in the face of additions of types (with their accompanying methods) to the bottom of the hierarchy, as done in languages such as JAVA [1].

1.2 Simple solutions

The most obvious solution is an $n \times m$ *dispatching matrix*, storing the outcomes of all possible dispatching queries. Figure 1.1b shows the dispatching matrix of Figure 1.1a, where the ℓ gray entries correspond to (non-inherited) family members.

In the dispatching matrix representation, queries are answered by a quick indexing operation. However, the space consumption is prohibitively large, e.g., 512MB for the dispatching matrix in the largest hierarchy in our benchmarks (8,793 types and 14,575 families).

Note that an encoding that does not try to compress pointers must use at least ℓ cells for representing the ℓ different method addresses. We would like to get as close as possible to this space requirement while preserving a constant and small query time. The dispatching matrix can be potentially compressed by a factor of

$$\iota \equiv (nm)/\ell. \tag{1.5}$$

We shall refer to ι as the *optimal compression factor*, and to schemes attempting to reach ι as *duplicates-elimination schemes*. In our data-set of 35 large hierarchies (see Section 5), $\iota \approx 725$.

Let w denote the number of non-null entries in the dispatching matrix, i.e.,

$$w \equiv |\{\langle F,t\rangle \mid \mathsf{dispatch}(F,t) \neq \mathrm{null}\}|\,. \tag{1.6}$$

By eliminating null memory cells, the dispatching matrix might be compressed by a factor of $(nm)/w$, which is around 150 in our data-set. Examples of null-elimination schemes are *row displacement* [5, 6], *selector coloring* [4, 12], and *virtual function tables* (VFT) [14]. In single-inheritance and static-typing setting of the problem, the VFT technique uses precisely w memory cells.

In the more general setting, the matrix can also be compressed into $O(w)$ cells (with fairly large constants) by using perfect hashing [8] or one of its variants. Even though dispatching time is constant in perfect hashing, it is complicated by the finite-field arithmetic incurred during the computation of the hash function.

With additional increase to the complexity of dispatching, there are variations to the famous FKS [8] scheme which use $w + o(w)$ cells. There is also a dynamic version of perfect hashing [3] which can support incremental dispatching. The memory toll is even larger, with constants in the range of a thousand.

Notice that even complete null-elimination gives suboptimal compression, since w might be substantially larger than ℓ. In our benchmark of 35 large hierarchies, w/ℓ is on average 8.3, and in one hierarchy it is 122.4!

It is not difficult to come close to complete duplicates-elimination, i.e., a space of $O(\ell)$, with a simple representation of the hierarchy as a graph where types are nodes and immediate inheritance relations are edges. The cost is of course the query time, which becomes $O(n)$, since we must traverse all the ancestors of a receiver in order to find the nearest family member. Sophisticated caching algorithms (as employed in the runtime system of SMALLTALK [2]) make the typical case more tolerable than what the worst case indicates.

1.3 Contribution

There is a large body of research on the dispatching problem (see e.g., [2, 4–6, 10, 12, 15–18]). The focus in these was on "practical" algorithms, which were evaluated empirically, rather than by provable upper bound on memory usage. The main theoretical research on the topic [7, 11] produced algorithms (for the single-inheritance setting) which using minimal space ($O(\ell)$ cells) supported dispatching in doubly logarithmic, $O(\lg\lg n)$, time. However, the hidden constants are large, and the implementation is complicated.

In this paper, we describe a different tradeoff: constant-time dispatching in d steps, while using at most $d\ell\sqrt[d]{\iota}$ cells. Stated differently, our results are that d steps in dispatching (provably) achieve a compression rate of $\frac{\iota}{d\sqrt[d]{\iota}}$. For example, with $d = 2$ the compression is by a factor of at least half of the square root of ι, the optimal compression rate. Also, the compression factor is close to optimal, $\frac{\iota}{2\lg m}$, when the dispatching time is logarithmic, $\lg m$.

An important advantage of these results in comparison to previous theoretical algorithms is that they are simple and straightforward to implement, and bear no hidden constants. In fact, our algorithms are based on a successful practical technique, namely *compact dispatch tables* (CT), which was invented by Vitek and Horspool [16]. Viewed differently, the results presented here give the first proof of a non-trivial upper bound on practical algorithms.

Even though the algorithms carry on to multiple-inheritance with the same time bounds of dispatching, the memory consumption increases by a factor of at most $(2\kappa)^{1-1/d}$, where κ can be thought of as a metric of the complexity of the topology of the inheritance hierarchy. (In a benchmark of 19 multiple-inheritance hierarchies with 34,810 types, we found the median value of an upper bound for κ is 5, the average is 6.4, and the maximum is 18.) Our previous work [18] on dispatching gives an implementation of a dispatching data structure whose space was only $O(\kappa\ell)$, but the dispatching time was logarithmic. The results presented here complete the tradeoff spectrum, giving constant time dispatching with any number of steps. We give empirical evidence that the algorithms perform well in practice, in many cases even better than the theoretically obtained upper bounds.

We also describe an incremental version of the algorithms in a single-inheritance setting, and prove that updates to the dispatching data structures can be made in optimal time. The cost is in a small constant factor increase (e.g., 2) to the memory footprint.

A variant of the this algorithm for the multiple-inheritance setting is not described for lack of space. Readers may also take interest in some proof techniques, including the representation of dispatching as search in a collection of partitionings, the elegant Lemma 4.1, and the amortization analysis of the incremental algorithm.

Outline The remainder of this article is organized as follows. Section 2 presents the generalized CT schemes for single-inheritance hierarchies. Section 3 shows how these schemes can be made incremental. A (non-incremental) version of these schemes for multiple-inheritance hierarchies is described in Section 4. Section 5 presents the experimental results: timing and compression values on a data-set of 35 hierarchies collected from both single and multiple dispatching languages. Open problems and directions for future research are the subject of Section 6.

2. Generalization of Compact Dispatch Tables for Single-Inheritance Hierarchies

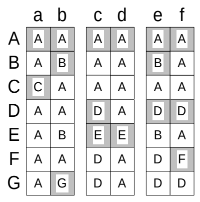
For simplicity, assume w.l.o.g. that the hierarchy is a tree (rather than a forest) rooted at a special node $\top \in \mathcal{T}$. There cannot be a *message ambiguous* in a single-inheritance setting. To avoid the other error situation, namely *message not understood*, we assume that $\top \in F$ for all $F \in \mathcal{F}$. With this assumption, every dispatching query returns a single family member. The cost is in (at most) doubling the number of implementations ℓ. (At the end of this section we will see that the memory toll is in fact much smaller.)

Vitek and Horspool's CT algorithm [16] partitions the family collection $\mathcal{F}$ into k disjoint *slices* $\mathcal{F} = \mathcal{F}_1 \cup \ldots \cup \mathcal{F}_k$. These slices break the dispatching matrix into k sub-matrices, also called *chunks*. The authors' experience was that chunks with 14 columns each give best results, and this number was hard-coded into their algorithm.

Figure 2.1 shows the three chunks of the dispatching matrix of Figure 1.1b for following partitioning:

$$\begin{aligned} \mathcal{F}_1 &= \{F_{\mathsf{a}}, F_{\mathsf{b}}\}, \\ \mathcal{F}_2 &= \{F_{\mathsf{c}}, F_{\mathsf{d}}\}, \\ \mathcal{F}_3 &= \{F_{\mathsf{e}}, F_{\mathsf{f}}\}. \end{aligned} \tag{2.1}$$

As Vitek and Horspool observed, and as can be seen in the figure, there are many identical rows in each chunk. Significant compression can be achieved by merging these rows together, and introducing, in each chunk, an auxiliary array of pointers to map each type to a row specimen.



	a	b	c	d	e	f
A	A	A	A	A	A	A
B	A	B	A	A	B	A
C	C	A	A	A	A	A
D	A	A	D	A	D	D
E	A	B	E	E	B	A
F	A	A	D	A	D	F
G	A	G	D	A	D	D

Figure 2.1: Three chunks of the dispatching matrix of Figure 1.1b

Why should there be many duplicate rows in each chunk? There are two contributing factors: (i) since the slices are small, there are not too many columns in a chunk, and (ii) that the number of distinct values which can occur in any given column is small, since, as empirical data shows, the number of different implementations of a selector is a small constant. Hence, there could not be too many distinct rows.

However, these considerations apply to any random distribution of values in the dispatching matrix. The crucial observation we make is that a much stronger bound on the number of distinct rows can be set relying on the fact that the values in the dispatching matrix are not arbitrary; they are generated from an underlying structured hierarchy.

Consider for example a chunk with two columns, with n_1 and n_2 distinct implementations in these columns. Simple counting considerations show that the number of distinct rows is at most $n_1 n_2$. Relying on the fact that the hierarchy is a tree we can show that the number of distinct rows is at most $n_1 + n_2$.

To demonstrate this observation, consider Figure 2.2a which focuses on the first chunk, corresponding to slice $\mathcal{F}_1 = \{F_{\mathsf{a}}, F_{\mathsf{b}}\}$.

As can be seen in the figure, the rows of types A, D, and F are identical. Figure 2.2b shows the compressed chunk and the auxiliary array. We see that this auxiliary array maps types A, D, and F to the same row.

We call attention to the (perhaps surprising) fact that it is possible to select from the elements of each row in Figure 2.2b a distinguishing representative. These representatives are members of what we call the *master-family*

$$F' = F_{\mathsf{a}} \cup F_{\mathsf{b}} = \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{G}\}.$$

The representatives of the four rows in the first chunk are A, B, C and G, in this order. The figure highlights these in gray. Also note

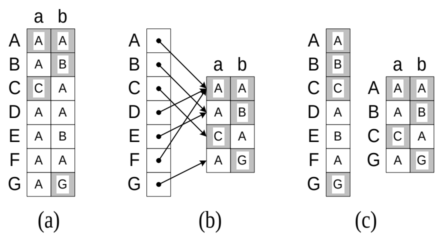


Figure 2.2: (a) The first chunk of Figure 1.1c, (b) the chunk compressed using an auxiliary array of pointers, and (c) the chunk compressed using an array of labels

that each member of the master-family serves as a representative of some row.

Figure 2.2c gives an alternative representation of the chunk, where each row is labeled by its representative. The auxiliary array now contains these labels instead of pointers. For example, the second row is labeled $\mathsf{B} \in F_{\mathsf{b}}$; the second and the fifth entry of the auxiliary array store B rather than the row specimen address.

Our improvement is based on the observation that the distinguishing representatives phenomenon is not a coincidence and on the observation that CT applies a *divide-and-conquer* approach to the dispatching problem: The search first determines the relevant master-family, and then continues to select the appropriate result among its members.

Let A_i denote the compressed i^{th} chunk of the dispatching matrix, and let B be the master dispatching matrix, whose columns are the auxiliary arrays of the chunks. Figure 2.3 shows matrices A_1, A_2, A_3 and B, which constitute the complete CT representation for the hierarchy of Figure 1.1. Note that the first column of B is the auxiliary array depicted in Figure 2.2c.

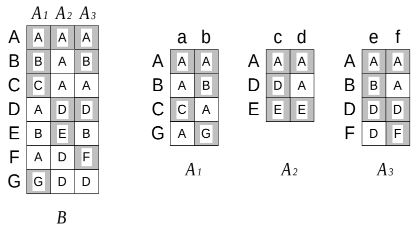


Figure 2.3: CT representation for the hierarchy of Figure 1.1

For each slice $\mathcal{F}_i$ let the *master-family* F'_i be the union of families in that slice, i.e., $F'_i \equiv \bigcup_{F\in\mathcal{F}_i} F$. Then, answering the query $\mathsf{dispatch}(F, t)$ at runtime requires three steps:

1. *Determine the slice of* F. That is, the family collection $\mathcal{F}_s$, such that $F \in \mathcal{F}_s$. If the partitioning into slices and the selector F are known at compile-time, as it is usually the case in dispatching of static-loading languages, then this stage incurs no penalty at runtime.

2. *Fetch the first dispatching result* $t' = \mathsf{dispatch}(F'_s, t)$. This value is found at the row which corresponds to type t and the column which corresponds to the master-family F'_s, i.e., $t' = B[t, s]$.

3. *Fetch the final dispatching result* $t'' = \mathsf{dispatch}(F, t)$. This type is found in the row of t' and the column of F in the compressed chunk A_s, i.e., $t" = A_s[t', F]$.

The algorithm merges together all the different messages in $\mathcal{F}_s$. At step 2, we find $t' \succeq t$, which is the smallest candidate in the merged master-family. Matrix B (of size $n \times k$) is the dispatching matrix of the types $\mathcal{T}$ and the master-family collection $\{F'_1, \ldots, F'_k\}$.

The search then continues with t', to find $t" \succeq t'$, the smallest candidate in F, the original family. Each matrix A_i (of size $|F'_i| \times |\mathcal{F}_i|$) is the dispatching matrix of the types in F'_i and the family collection $\mathcal{F}_i$.

To understand the space saving, consider just two families F_1 and F_2. The naive implementation of dispatch is using *two* arrays, each of size $n = |\mathcal{T}|$, which map each type t to two types $t"_1 \in F_1$ and $t_2" \in F_2$, such that $t_i" = \mathsf{dispatch}(t, F_i)$, $i = 1, 2$. A more compact representation can be obtained by using a *single* array of size n, to dispatch first on the merged master-family $F' = F_1 \cup F_2$. Let $t' \in F'$ be the result of this dispatch. The crucial point is that the smallest candidate for t', in either F_1 or F_2, is the same as for t. Since there are $|F'| \le |F_1| + |F_2|$ different values of t', a continued search from t' (for either F_1 or F_2) can be implemented using two arrays, each of size $|F'|$. The first such array maps F' to F_1; the second to F_2. Total memory used is $n + 2|F'|$ instead of $2n$ cells, while the cost is an additional dereferencing operation.

More generally, given a dispatching problem for a family collection $\mathcal{F}$, the *CT reduction* partitions $\mathcal{F}$ into k disjoint slices

$$\mathcal{F} = \mathcal{F}_1 \cup \ldots \cup \mathcal{F}_k, \tag{2.2}$$

and merges together the families in each slice by defining a master-family

$$F'_i \equiv \bigcup_{F\in\mathcal{F}_i} F, \tag{2.3}$$

for all $i = 1, \ldots, k$. Let A_i be the matrix whose dimensions are

$$|F'_i| \times |\mathcal{F}_i|, \tag{2.4}$$

corresponding to the i^{th} slice. Then, the query $\mathsf{dispatch}(F, t)$ is realized by the fetch

$$A_s[\mathsf{dispatch}(F'_s, t), F], \tag{2.5}$$

where $F \in \mathcal{F}_s$.

Since both steps 2 and 3 in the dispatching are in essence a dispatching operation, better compaction of the dispatching data structure might be achieved by applying the CT technique recursively to either the matrix B, or all the matrices A_i. It is not difficult to see that each of the recursive applications will yield the same dispatching data structure, in which the set of selectors is organized in a three-level hierarchy of partitions: families, master-families, and master-master-families (so to speak). We chose to describe this 3-level system by applying the CT technique to the matrix B. The (potential) saving in space comes at a cost of another dereferencing step during dispatch. Clearly, we could recursively apply the reduction any number of times.

We need the following notation in order to optimize these recursive applications. Let $\mathrm{mem}_d(n, m, \ell)$ denote the memory required for solving the dispatching problem of n types, m families and ℓ method implementations, using d dereferencing operations during dispatch. A simple dispatching matrix representation gives

$$\mathrm{mem}_1(n, m, \ell) = nm. \tag{2.6}$$

Each application of the CT reduction adds another dereferencing, while reducing a dispatching problem with parameters $\langle n, m, \ell\rangle$ to a new dispatching problem with parameters $\langle n, k, \ell'\rangle$, where

$$\ell' = \sum_{i=1}^{k} |F'_i| = \sum_{i=1}^{k} \left| \bigcup_{F\in\mathcal{F}_i} F \right| .$$

Note that $\ell' \leq \ell$. To see this recall that

$$\ell = \sum_{F \in \mathcal{F}} |F| = \sum_{i=1}^{k} \sum_{F \in \mathcal{F}_i} |F|,$$

and apply the fact that the cardinality of the union of sets is at most the sum of cardinalities of these sets

$$\ell' = \sum_{i=1}^{k} \left| \bigcup_{F \in \mathcal{F}_i} F \right| \leq \sum_{i=1}^{k} \sum_{F \in \mathcal{F}_i} |F| = \ell. \tag{2.7}$$

The reduction generates the matrices $A_1, \ldots, A_k$. To estimate their size suppose that all slices are equal in size, i.e., they all have x families. (For simplicity we ignore the case that m is not divisible by x, in which slices are *almost* equal.) Then, the total memory generated by the reduction is

$$\sum_{i=1}^{k} |F_i'| \times |\mathcal{F}_i| = \sum_{i=1}^{k} |F_i'| \times x = x \sum_{i=1}^{k} |F_i'| = x\ell' \leq x\ell.$$

To conclude, the costs of the CT reduction are another dereferencing and an additional space of $x\ell$. In return, a dispatching problem with parameters $\langle n, m, \ell \rangle$ is reduced to a new dispatching problem with parameters $\langle n, k, \ell' \rangle$, where $k = m/x$ and $\ell' \leq \ell$. Formally,

$$\mathrm{mem}_{d+1}(n, m, \ell) \leq \ell x + \mathrm{mem}_d(n, m/x, \ell), \tag{2.8}$$

where x is arbitrary.

Let CT_d be the dispatching data structure and algorithm obtained by applying the CT reduction $d-1$ times to the original dispatching problem. The recursion is ended by applying simple dispatching matrix at the last step. Thus, CT_1 is simply the dispatching matrix, while CT_2 is similar to Vitek and Horspool's algorithm (with $x = 14$). By making $d-1$ substitutions of (2.8) into itself, and then using (2.6), we obtain

$$\mathrm{mem}_d(n, m, \ell) \leq \ell x_1 + \cdots + \ell x_{d-1} + \frac{nm}{x_1 x_2 \cdots x_{d-1}}, \tag{2.9}$$

where x_i is the slice size used during the i^{th} application of the CT reduction. Symmetry considerations indicate that the bound in (2.9) is minimized when all x_i are equal. We have,

$$\mathrm{mem}_d(n, m, \ell) \leq (d-1)\ell x + \frac{nm}{x^{d-1}}, \tag{2.10}$$

which is minimized when $x = (nm/\ell)^{1/d}$.

Table 1 summarizes the space and time requirements of algorithms CT_d, where $\iota \equiv (nm)/\ell$ is the optimal compression factor.

Scheme	Slice size	Time	Space	Compression factor
CT_1	N/A	1	$\ell\iota$	1
CT_2	$\sqrt[2]{\iota}$	2	$2\ell\sqrt[2]{\iota}$	$\frac{\iota}{2\sqrt[2]{\iota}}$
CT_3	$\sqrt[3]{\iota}$	3	$3\ell\sqrt[3]{\iota}$	$\frac{\iota}{3\sqrt[3]{\iota}}$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
CT_d	$\sqrt[d]{\iota}$	d	$d\ell\sqrt[d]{\iota}$	$\frac{\iota}{d\sqrt[d]{\iota}}$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
$\mathrm{CT}_{\log_x m}$	x	$\log_x m$	$(\log_x m)\ell x$	$\frac{\iota}{x \log_x m}$

Table 1: Generalized CT results for single-inheritance hierarchies

The last row in the table is obtained by applying the CT reduction a maximal number of times. In each application the slice size is x (typically, $x = 2$). The collection $\mathcal{F}$ is then organized in a hierarchy of $\log_x m$ levels, which is also the number of dereferencing steps during dispatch. The memory used in each level is ℓx (see (2.8)).

The generalizations (Table 1) of CT_d over Vitek and Horspool's algorithm is in the following directions: (i) a sequence of algorithms which offer a tradeoff between the size of the representation and the dispatching time, and (ii) precise performance analysis, which dictates an optimal slice size, instead of the arbitrary universal recommendation, $x = 14$.

In reflecting on the generalized CT algorithm we see that they are readily adapted to the case where *message not understood* are allowed as is the case in dynamically typed languages. Whenever the search in a master-family F' returns $\top$, we can be certain that the search in every constituent of F' will also return $\top$. Therefore, it is possible to check after each dereferencing operation whether the fetched type is $\top$, and emit the appropriate error message. A more appealing alternative is to continue the search with $\top$, using an array which maps $\top$ into itself for each constituent of F'. Now, since this array does not depend on the identity of F', we can store only one such copy for each application of the CT reduction. The memory toll that CT_d bears for these arrays is $(d-1)x$ cells.

Note also that Vitek and Horspool's idea of using selector coloring [4, 12] in each chunk is still applicable with a slight variation to our generalization. If certain columns in a chunk contain many $\top$ elements, it might be possible to collapse these columns together.

3. Incremental variants for Single-Inheritance hierarchies

This section describes an incremental variant of the CT scheme in the single-inheritance setting, achieving two important properties: (i) the *space* it uses is at most twice that of the static algorithm, and (ii) its total *runtime* is linear in the final encoding size. (We cannot expect an asymptotically better runtime since the algorithm must at least output the final encoding.)

Section 3.1 describes ICT_2, the incremental variant of CT_2. Section 3.2 gives the generalization for CT_d.

The main idea is to *rebuild the entire encoding* whenever the ratio between the current slice size and the optimal one reaches a high- or low-water mark (for example 2 and 1/2). Therefore, some insertions will take longer to process than others. We therefore obtain bounds on the *amortized* time for an insertion.[3] The amortized time of an insertion is asymptotically optimal since the total runtime is linear in the final encoding size. Using techniques of "background copying" [3], it is possible to amend the algorithms so that the *worst case* insertion time is optimal as well.

Note that unlike the static version of the problem, we cannot assume that the families always include the root $\top$. The reason is that this assumption would require $\top$ to include implementation of *all* families, and the initial value of the number of families will jump to m.

3.1 ICT_2 in a single-inheritance setting

The CT_2 scheme applies a single CT reduction and uses a dispatching matrix for the resulting master-families. This process divides the dispatching problem into independent sub-problems: one dispatching matrix, and a set of matrices A_i, $i = 1, \ldots, k$, which (in a single-inheritance setting) are in fact dispatching matrices as well.

We first note that it is relatively easy to maintain a plain, single-level, dispatching matrix subject to type insertions. The cost is in an

[3]We remind the reader that the *amortized time* of an operation is $c(n)$, if a sequence of n such operations requires at most $nc(n)$ time. The worst case time of any single operation can however be much greater than $c(n)$. For more information on amortized complexity see [13].

additional comparison to guard against array overflows. Consider a newly added type t. The row of t in the dispatching matrix is identical to the row of its parent, except for entries corresponding to families in which t is a member. Note that the insertion time of a type is linear in its row size, and the total runtime is therefore linear in the final encoding size.

There is a slight difficulty when t *introduces* new families. (A new type t *introduces* a family F, $t \in F$, iff no other type was a member of F.) Observe that the dispatching result for such a newly introduced family and every other type is always null. Therefore, we extend the row of t and place the entries for these new families at the end. However, instead of extending all the other rows with null entries, we perform a range-check before accessing any given row. In the case of array-overflow we return null, otherwise we proceed as usual.

The space requirement of CT$_2$ in a single-inheritance setting is (see Table 1)

$$\mathrm{mem}(x) = \ell x + nm/x, \tag{3.1}$$

which is minimized when the slice size is

$$x_{\text{OPT}} = \sqrt{nm/\ell}. \tag{3.2}$$

Algorithm ICT$_2$ will maintain the following invariant

$$\boxed{\frac{x_{\text{OPT}}}{2} \le x \le 2x_{\text{OPT}},} \tag{3.3}$$

and will rebuild the encoding whenever this condition is violated. Algorithm 1 shows the procedure to apply whenever a new type is added to the hierarchy.

```
1: Let $x$ be the current slice size.
2: Let $\langle n, m, \ell \rangle$ be the current problem parameters.
3: $x_{\text{OPT}} \leftarrow \sqrt{nm/\ell}$        // The optimal slice size.
4: **If not** $\left(\frac{x_{\text{OPT}}}{2} \le x \le 2x_{\text{OPT}}\right)$ **then**
5:     $x \leftarrow x_{\text{OPT}}$
6:     Rebuild the entire CT$_2$ encoding
7: **end If**
8: Insert $t$ to the CT$_2$ encoding
```

Algorithm 1: Insertion of a new type t in ICT$_2$

Substituting (3.2) in (3.1) we find the optimal encoding size

$$\mathrm{mem}(x_{\text{OPT}}) = 2\sqrt{nm\ell}.$$

Let us write this as a function of the problem parameters,

$$f(n, m, \ell) \equiv \mathrm{mem}(x_{\text{OPT}}) = 2\sqrt{nm\ell}.$$

and study the properties of this function.

FACT 3.1. *Function f is monotonic in all three arguments n, m, ℓ.*

FACT 3.2. *There are constants c_1, c_2, c_3, such that*

$$\begin{aligned} \sum_{i=0}^{\infty} f\left(\frac{n}{2^i}, m, \ell\right) &\le c_1 f(n, m, \ell), \\ \sum_{i=0}^{\infty} f\left(n, \frac{m}{2^i}, \ell\right) &\le c_2 f(n, m, \ell), \\ \sum_{i=0}^{\infty} f\left(n, m, \frac{\ell}{2^i}\right) &\le c_3 f(n, m, \ell). \end{aligned} \tag{3.4}$$

PROOF. Note that

$$\begin{aligned} \sum_{i=0}^{\infty} f\left(\frac{n}{2^i}, m, \ell\right) &= \sum_{i=0}^{\infty} \sqrt{\frac{n}{2^i} m\ell} \\ &= \sqrt{nm\ell} \sum_{i=0}^{\infty} \sqrt{\frac{1}{2^i}} \\ &\le \frac{2}{2-\sqrt{2}} \sqrt{nm\ell} \in O(f(n, m, \ell)). \end{aligned}$$

The proof for parameters m and ℓ is identical. □

LEMMA 3.3. *The* space requirement *of* ICT$_2$ *is at most*

$$2f(n, m, \ell).$$

PROOF. From the algorithm invariant (3.3) it follows that

$$\begin{aligned} \mathrm{mem}(x) &= \ell x + nm/x \\ &\le \ell(2x_{\text{OPT}}) + nm/\left(\frac{x_{\text{OPT}}}{2}\right) \\ &= 2(\ell x_{\text{OPT}} + nm/x_{\text{OPT}}) \\ &= 2\,\mathrm{mem}(x_{\text{OPT}}) = 2f(n, m, \ell). \end{aligned}$$ □

Our next objective is to prove that the total *runtime* of ICT$_2$ is linear in $f(n, m, \ell)$. To do so, we will breakdown the sequence of insertions carried out by the algorithm into *phases*, according to the points in time where rebuilding took place. No rebuilding occurs within a phase, and all that is required is to maintain several plain dispatching matrices. Hence, the total runtime of the insertions in a phase is linear in the encoding size at the end of this phase.

The main observation is that rebuilding happens only when at least one of the problem parameters is doubled. We distinguish between three *kinds* of rebuilds, depending on the parameter which was doubled. We then show that the total runtime of rebuilds of the same kind is linear in $f(n, m, \ell)$.

Formally, phase i begins immediately after phase $i-1$, and ends after the encoding was built for the i^{th} time (the last phase ends when the program terminates). Let $\langle n_i, m_i, \ell_i \rangle$, $i = 1, \ldots, p$, be the problem parameters at the end of phase i. Observe that the problem parameters can only increase, i.e., $n_{i+1} \ge n_i$, $m_{i+1} \ge m_i$, and $\ell_{i+1} \ge \ell_i$, Phase i finishes with an encoding size of at most $2f(n_i, m_i, \ell_i)$, therefore its runtime is linear in $f(n_i, m_i, \ell_i)$. Thus, the total runtime is linear in

$$\sum_{i=1}^{p} f(n_i, m_i, \ell_i). \tag{3.5}$$

We need to show that this sum is linear in $f(n_p, m_p, \ell_p)$.

LEMMA 3.4. *Invariant* (3.3) *is violated only when* at least one *of the problem parameters is doubled, i.e., one of the following holds*

$$\begin{aligned} n_{i+1} &\ge 2n_i, \\ m_{i+1} &\ge 2m_i, \\ \ell_{i+1} &\ge 2\ell_i. \end{aligned} \tag{3.6}$$

PROOF. Let x_j denote the slice size at the beginning of phase j, i.e.,

$$x_j \equiv \sqrt{\frac{n_j m_j}{\ell_j}}. \tag{3.7}$$

At the end of phase i one of the following conditions must hold

$$\begin{aligned} x_{i+1} &\ge 2x_i, \\ x_{i+1} &\le \frac{1}{2} x_i. \end{aligned} \tag{3.8}$$

From (3.7) and (3.8), we have

$$\begin{aligned}\frac{n_{i+1}m_{i+1}}{\ell_{i+1}} &\geq \frac{4n_i m_i}{\ell_i},\\ \frac{n_{i+1}m_{i+1}}{\ell_{i+1}} &\leq \frac{n_i m_i}{4\ell_i}.\end{aligned} \tag{3.9}$$

Since the problem parameters can only increase,

$$\begin{aligned}n_{i+1}m_{i+1} &\geq 4n_i m_i,\\ \ell_{i+1} &\geq 4\ell_i,\end{aligned} \tag{3.10}$$

which implies that at least one of the parameters was doubled. □

LEMMA 3.5. *The total* runtime *of* ICT$_2$ *is linear in*

$$f(n_p, m_p, \ell_p).$$

PROOF. Let $\{(N_1, M_1, L_1), \ldots, (N_q, M_q, L_q)\}$ be the problem parameters of phases where n was doubled, i.e., $N_{i+1} \geq 2N_i$. Therefore,

$$N_q \geq 2N_{q-1} \geq \ldots \geq 2^{q-1}N_1. \tag{3.11}$$

Using Fact 3.2, the total runtime of these phases is linear in

$$\begin{aligned}\sum_{i=1}^{q} f(N_i, M_i, L_i) &\leq \sum_{i=1}^{q} f(N_i, M_q, L_q)\\ &\leq \sum_{i=1}^{q} f\left(\frac{N_q}{2^{q-j}}, M_q, L_q\right)\\ &\in O(f(N_q, M_q, L_q)).\end{aligned} \tag{3.12}$$

The same consideration applies to phases in which the number of methods or the number of families was doubled. So, the runtime of the entire algorithm is the total runtime of the three kinds of phases, which is linear in $f(n_p, m_p, \ell_p)$. □

3.2 ICT$_d$ in a single-inheritance setting

The generalization to $d > 2$ is mostly technical, as outlined next. Function $\mathrm{mem}(x)$, the space requirement of CT$_d$ as defined in (2.10) is minimized when the slice size is

$$x_{\text{OPT}} = \sqrt[d]{nm/\ell}.$$

Let function f_d denote the optimal encoding size

$$f_d(n, m, \ell) \equiv \mathrm{mem}(x_{\text{OPT}}) = d\ell\sqrt[d]{\iota}.$$

Algorithm ICT$_d$ will preserve the following invariant

$$\boxed{\frac{x_{\text{OPT}}}{2^{1/(d-1)}} \leq x \leq 2x_{\text{OPT}}.} \tag{3.13}$$

LEMMA 3.6. *The space requirement of* ICT$_d$ *is at most*

$$2f_d(n, m, \ell).$$

PROOF. Similar to that of Lemma 3.3 □

FACT 3.7. *There are constants* c_1, c_2, c_3*, such that*

$$\begin{aligned}\sum_{i=0}^{\infty} f_d\left(\frac{n}{2^i}, m, \ell\right) &\leq c_1 f_d(n, m, \ell),\\ \sum_{i=0}^{\infty} f_d\left(n, \frac{m}{2^i}, \ell\right) &\leq c_2 f_d(n, m, \ell),\\ \sum_{i=0}^{\infty} f_d\left(n, m, \frac{\ell}{2^i}\right) &\leq c_3 f_d(n, m, \ell).\end{aligned} \tag{3.14}$$

LEMMA 3.8. *Rebuilding only takes place when* at least one *of the problem parameters is doubled.*

PROOF. Similar to that of Lemma 3.4 □

LEMMA 3.9. *The total runtime of* ICT$_d$ *is linear in*

$$f_d(n, m, \ell).$$

PROOF. Similar to Lemma 3.5. □

4. Generalization of Compact Dispatch Tables for Multiple-Inheritance Hierarchies

This section explains how to generalize the CT reduction as described in Section 2 to the multiple-inheritance setting. In a single-inheritance hierarchy, there could never be more than one most specific family member in response to a dispatch query. The fact that this is no longer true in multiple-inheritance hierarchies makes it difficult to apply the CT reduction to such hierarchies. Even if the original families are appropriately augmented to remove all such ambiguities, ambiguities may still occur in the master-families as they are generated by the reduction.

We will therefore use a notion of a *generalized dispatching query*, denoted $\textsf{g-dispatch}(F, t)$, which returns *the entire set* of smallest candidates, rather than null in case that this set is not a singleton. Formally,

$$\textsf{g-dispatch}(F, t) \equiv \min(\textsf{cand}(F, t)). \tag{4.1}$$

Generalized dispatching is a data-structure transaction rather than an actual runtime operation which must result in a single method to execute.

Consider for example the hierarchy of Figure 4.1.

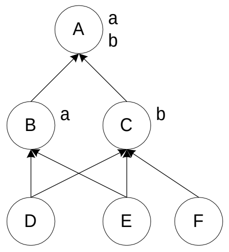


Figure 4.1: A small example of a multiple-inheritance hierarchy with two families

The figure shows two families of methods, F_{a} and F_{b},

$$\begin{aligned}F_{\mathsf{a}} &= \{\mathsf{A}, \mathsf{B}\},\\ F_{\mathsf{b}} &= \{\mathsf{A}, \mathsf{C}\}.\end{aligned} \tag{4.2}$$

The dispatching matrix of these two families is depicted in Figure 4.2a. Note that the results of all dispatching queries on types D and E (for example) are the same. The corresponding rows in the table are identical and can be compressed. Figure 4.2b shows a representation of the dispatching matrix obtained by merging together all identical rows and an auxiliary array of pointers to all different rows specimens.

This compressed representation can be understood in terms of the master-family

$$F' \equiv F_{\mathsf{a}} \cup F_{\mathsf{b}} = \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}.$$

The auxiliary array represents all the possible results of a *generalized* dispatch on this master-family. For example,

$$\textsf{g-dispatch}(F', \mathsf{D}) = \textsf{g-dispatch}(F', \mathsf{E}) = \{\mathsf{B}, \mathsf{C}\}.$$

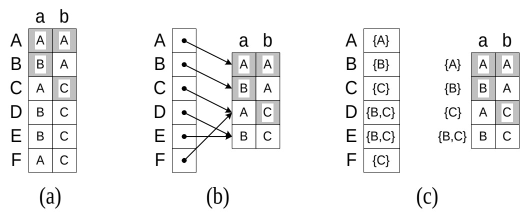


Figure 4.2: (a) The dispatching matrix of Figure 4.1, (b) the matrix compressed using an auxiliary array of pointers, and (c) the matrix compressed using an array of set-labels

Therefore, the D and E entries in the auxiliary array point to the same row specimen whose label is the set $\{B, C\}$.

In total there are four different results of generalized dispatching with respect to F'. Family F' therefore partitions the types in the hierarchy into four sets, as shown in Figure 4.2c. The figure shows the same compressed representation of the dispatching matrix, where the results of generalized dispatch are used to label row specimens instead of pointers in the auxiliary array.

In order to derive bounds on the quality of the CT compression in the multiple-inheritance setting we need to estimate the number of distinct rows in chunks. The difficulty is that the result of a generalized dispatch is a set rather than a singleton, and hence this number might be exponential in the family size. To show that this is not the case, we first define the notion of a partition imposed by a family, and then show the size of this partition is at most 2κ times the size of the family, where $1 \le \kappa \le n$ is a (usually small) metric of the complexity of the hierarchy.

4.1 Family Partitionings

Given a partially ordered set of types $\mathcal{T}$ and a family of implementations $F \subseteq \mathcal{T}$, the *partitioning of* $\mathcal{T}$ *by* F, also called the *family partitioning* due to F, is

$$\nabla F \equiv \{\mathcal{T}_1, \ldots, \mathcal{T}_n\},$$

such that all types in *partition* $\mathcal{T}_i$ have the same generalized dispatch result. In other words, types $a, b \in \mathcal{T}$ are in the same *partition* $\mathcal{T}_i \in \nabla F$ if and only if

$$\text{g-dispatch}(F, a) = \text{g-dispatch}(F, b). \tag{4.3}$$

Figure 4.3 shows the family partitioning of the families F_a, F_b of (4.2) and their master-family $F' \equiv F_a \cup F_b$.

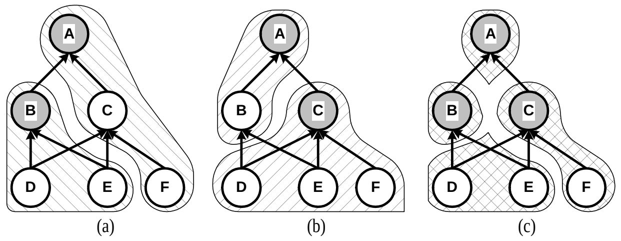


Figure 4.3: The family partitionings of the families F_a, F_b of (4.2) and their master-family $F' \equiv F_a \cup F_b$

Types D and E, for example, are in the same partition in $\nabla F'$ since $\text{g-dispatch}(F', D) = \text{g-dispatch}(F', E) = \{B, C\}$. The partitionings are

$$\begin{aligned} \nabla F_a &\equiv \{\{A, C, F\}, \{B, D, E\}\}, \\ \nabla F_b &\equiv \{\{A, B\}, \{C, D, E, F\}\}, \\ \nabla F' &\equiv \{\{A\}, \{B\}, \{C, F\}, \{D, E\}\}. \end{aligned} \tag{4.4}$$

Figure 4.4 overlays ∇F_a and ∇F_b. The dotted lines are the partitions of ∇F_a, whereas the full lines are the partitions of ∇F_b.

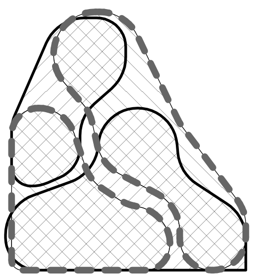


Figure 4.4: The *overlay* of ∇F_a and ∇F_b of Figure 4.3

In comparing Figure 4.3c with Figure 4.4, we see that the partitioning $\nabla F'$ can be obtained by a simple overlay of the two partitionings ∇F_a and ∇F_b. We will next prove that this was no coincidence.

Given two partitionings π, π', their *overlay* $\pi \cdot \pi'$ is the coarsest partitioning consistent with both π and π'. Constructively, the overlay is obtained by intersecting all partitions of π with all partitions of π':

$$\pi \cdot \pi' = \{\mathcal{T}_i \cap \mathcal{T}'_j \mid \mathcal{T}_i \in \pi, \mathcal{T}'_j \in \pi'\}. \tag{4.5}$$

For example, the overlay of ∇F_a and ∇F_b of (4.4) is

$$\begin{aligned} \nabla F_a \cdot \nabla F_b = \{&\{A, C, F\} \cap \{A, B\}, \{A, C, F\} \cap \{C, D, E, F\}, \\ &\{B, D, E\} \cap \{A, B\}, \{B, D, E\} \cap \{C, D, E, F\}\} \\ = \{&\{A\}, \{C, F\}, \{B\}, \{D, E\}\}. \end{aligned} \tag{4.6}$$

LEMMA 4.1. *$\nabla F_1 \cdot \nabla F_2 = \nabla(F_1 \cup F_2)$ for all F_1, F_2.*

PROOF. It is a well known fact that for every partitioning π there is a binary *equivalence relation* whose set of equivalence classes are the same as the partitioning π. Instead of proving that the partitioning $\nabla(F_1 \cup F_2)$ and $\nabla F_1 \cdot \nabla F_2$ are equal, we will prove that their equivalence relations are the same.

On the one hand, types a, b are in the equivalence relation of

$$\nabla(F_1 \cup F_2)$$

iff they have the same generalized dispatching results with respect to $F_1 \cup F_2$ (see (4.3)), i.e.,

$$\text{g-dispatch}(F_1 \cup F_2, a) = \text{g-dispatch}(F_1 \cup F_2, b). \tag{4.7}$$

On the other hand, the overlay partitioning, $\nabla F_1 \cdot \nabla F_2$, is defined by intersecting all partitions of ∇F_1 with those of ∇F_2 (see (4.5)). Therefore, types a, b are in the equivalence relation of $\nabla F_1 \cdot \nabla F_2$ iff the following two conditions hold

$$\begin{aligned} \text{g-dispatch}(F_1, a) &= \text{g-dispatch}(F_1, b), \\ \text{g-dispatch}(F_2, a) &= \text{g-dispatch}(F_2, b). \end{aligned} \tag{4.8}$$

We must show that (4.7) holds iff (4.8) holds. Formally, using the definition of generalized dispatch (4.1), we must show that

$$\begin{gathered} \min(\text{cand}(F_1 \cup F_2, a)) = \min(\text{cand}(F_1 \cup F_2, b)) \\ \Leftrightarrow \\ \min(\text{cand}(F_1, a)) = \min(\text{cand}(F_1, b)) \ \wedge \\ \min(\text{cand}(F_2, a)) = \min(\text{cand}(F_2, b)). \end{gathered} \tag{4.9}$$

Since two sets of candidates (for the same family) have the same smallest elements iff they are equal, our objective is to prove (see the definition of candidates in (1.2))

$$\begin{aligned}(F_1 \cup F_2) \cap \text{ancestors}(a) &= (F_1 \cup F_2) \cap \text{ancestors}(b) \\ &\Leftrightarrow \\ F_1 \cap \text{ancestors}(a) &= F_1 \cap \text{ancestors}(b) \ \wedge \\ F_2 \cap \text{ancestors}(a) &= F_2 \cap \text{ancestors}(b).\end{aligned} \tag{4.10}$$

Given two sets X, Y, their *symmetric difference* is defined as

$$X \triangle Y \equiv (X \cup Y) \setminus (X \cap Y).$$

Observe that

$$Z \cap X = Z \cap Y \Leftrightarrow Z \cap (X \triangle Y) = \emptyset. \tag{4.11}$$

By combining (4.10) and (4.11) we find that we need to prove that

$$\begin{aligned}(F_1 \cup F_2) \cap (\text{ancestors}(a) \triangle \text{ancestors}(b)) &= \emptyset \\ &\Leftrightarrow \\ F_1 \cap (\text{ancestors}(a) \triangle \text{ancestors}(b)) &= \emptyset \ \wedge \\ F_2 \cap (\text{ancestors}(a) \triangle \text{ancestors}(b)) &= \emptyset.\end{aligned} \tag{4.12}$$

The above trivially holds since for all sets X, Y, Z,

$$\begin{aligned}(X \cup Y) \cap Z &= \emptyset \\ &\Leftrightarrow \\ X \cap Z &= \emptyset \ \wedge \\ Y \cap Z &= \emptyset. \qquad \square\end{aligned}$$

4.2 Memory requirements of the reduction

As in the single-inheritance version, the CT reduction partitions $\mathcal{F}$ into disjoint slices $\mathcal{F}_1, \ldots, \mathcal{F}_k$, and generates for the i^{th} slice the master-family F'_i by merging the families in this slice. To answer the *generalized dispatching* query $\textsf{g-dispatch}(F, t)$, where $F \in \mathcal{F}_i$, we first (recursively) answer the query $\textsf{g-dispatch}(F'_i, t)$, in the collection of master-families, $\{F'_1, \ldots, F'_k\}$. This recursive call returns one of the partitions of $\nabla F'_i$. The next step is to find the unique containing partition of ∇F.

To understand this better, recall that $F \subseteq F'_i$. To apply Lemma 4.1 note that there exists a set X such that $F'_i = F \cup X$, and hence

$$\nabla F'_i = \nabla(F \cup X) = \nabla F \cdot \nabla X.$$

Therefore, every partition of $\nabla F'_i$ is contained in a partition of ∇F. A matrix A_i with $|\nabla F'_i|$ rows and $|\mathcal{F}_i|$ columns is used to map each of the partitions of $\nabla F'_i$ to a partition of ∇F, for all $F \in \mathcal{F}_i$. Matrices $A_1, \ldots, A_k$ are nothing other than the dispatching data structure of the CT reduction. (Clearly, there is an additional data structure which the recursive call uses.)

To bound the size of these matrices, we need to estimate $|\nabla F|$. In single-inheritance, $|\nabla F| = |F|$. An easy, but not so useful bound in multiple-inheritance, is $|\nabla F| \leq 2^{|F|}$.

A better bound is given by defining κ, the complexity of a hierarchy, and then showing that

$$|\nabla F| \leq 2\kappa |F|. \tag{4.13}$$

Using slices with x families in each, the total memory of matrices $A_1, \ldots, A_k$ is

$$\sum_{i=1}^{k} |\nabla F'_i| \times |\mathcal{F}_i| = \sum_{i=1}^{k} |\nabla F'_i| \times x \leq x \sum_{i=1}^{k} 2\kappa |F'_i| \leq 2x\kappa\ell.$$

The recursive equations then become

$$\begin{aligned}\text{mem}_1(n, m, \ell) &= nm, \\ \text{mem}_{d+1}(n, m, \ell) &\leq 2\kappa\ell \cdot x + \text{mem}_d(n, m/x, \ell).\end{aligned} \tag{4.14}$$

By using $2\kappa\ell$ instead of ℓ, the analysis of the previous section holds.

COROLLARY 4.2. *Let $\varphi \equiv (nm)/(2\kappa\ell)$. In a hierarchy whose complexity is κ, CT_d performs dispatching in d dereferencing operations, and reaches a compression factor of at least $\frac{1}{d}\varphi^{1-1/d}$ (when using a slice size of $\varphi^{1/d}$).*

In other words, in a hierarchy whose complexity is κ, the space requirements of CT_d in the multiple-inheritance setting is worse than the single-inheritance setting by a factor of at most $(2\kappa)^{1-1/d}$.

4.3 Hierarchy Complexity

DEFINITION 4.3. *The complexity of a hierarchy is the minimal number κ such that there exists partitioning of $\mathcal{T}$ into sets $\mathcal{T}_1, \ldots, \mathcal{T}_\kappa$, and an ordering π_i of $\mathcal{T}_i$, $i = 1, \ldots, \kappa$, such that for every type $t \in \mathcal{T}$, the set $\text{descendants}(t) \cap \mathcal{T}_i$ is an interval in π_i.*

Clearly, the complexity of a hierarchy is 1 if there exists an ordering π of $\mathcal{T}$ in which descendants of any type define an *interval*. All single-inheritance hierarchies have complexity 1 since in a simple preorder the descendants of any type are consecutive.

Figure 4.5 is a multiple-inheritance hierarchy of complexity 1. Within each type we write its position in π.

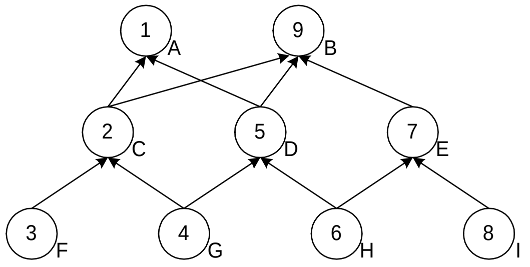


Figure 4.5: An example of a multiple-inheritance hierarchy of complexity 1

Figure 4.6 shows the family partitioning of the family $F = \{\mathsf{A}, \mathsf{B}, \mathsf{E}\}$ in the hierarchy of Figure 4.5. Observe that $|\nabla F| = 5$.

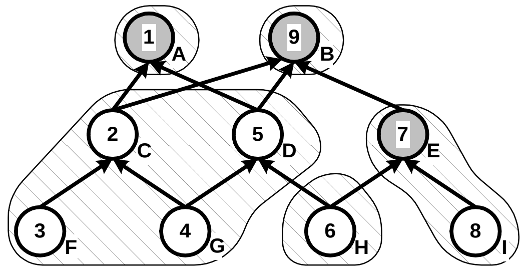


Figure 4.6: The family partitioning of the family $F = \{\mathsf{A}, \mathsf{B}, \mathsf{E}\}$ in the hierarchy of Figure 4.5

Since the complexity of this hierarchy is 1, the descendants of each type define an *interval*. Therefore the family F defines the three intervals depicted in Figure 4.7. Those intervals partition the types into 5 *segments*. (We will show that there are at most $2|F|$ segments.) Types in the same segment have the same set of candidates and therefore belong to the same partition. So we conclude that the number of partitions is at most the number of segments, which in turn is at most $2|F|$. In our example,

$$|\nabla F| = 5 \leq 6 = 2|F|.$$

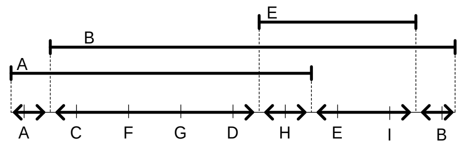


Figure 4.7: The intervals of the family $F = \{\mathsf{A}, \mathsf{B}, \mathsf{E}\}$ in the hierarchy of Figure 4.5

LEMMA 4.4. *$|\nabla F| \leq 2\kappa|F|$ for each family F.*

PROOF. We need the following fact, whose proof is elementary.

> *A set of f intervals partition any consecutive set into at most $2f+1$ segments. Out of these segments at most $2f-1$ are contained in one interval or more.* (See illustration in Figure 4.7.)

Let $f = |F|$. Recall (Definition 4.3) the partitioning of $\mathcal{T}$ into sets $\mathcal{T}_1, \ldots, \mathcal{T}_\kappa$ with their respective ordering. Let i be fixed. We write the list of members of the set $\mathcal{T}_i$, enumerated in its respective order π_i.

Consider a type $t \in \mathcal{T}_i$. The result of $\mathsf{g\text{-}dispatch}(F, t)$ is uniquely determined by the subset of all types $t' \in F$, such that the t is among the descendant of t'. From Definition 4.3, we have that the descendants are consecutive in the list of $\mathcal{T}_i$. Family F defines therefore f intervals (which may be empty) in this list. These intervals partition the list into at most $2f+1$ segments such that the result of $\mathsf{g\text{-}dispatch}(F, t)$ is uniquely determined by the segment of t. These segments give the restriction of ∇F to $\mathcal{T}_i$.

We have thus obtained $|\nabla F| \leq \kappa(2f+1)$. To obtain a tighter bound we need a more careful counting. Let us remove from $\mathcal{T}_i$ all types which are not descendants of any of the members of F. The remaining types are divided by F into $2f-1$ segments. Generalized dispatching on the removed types returns the empty set, irrespective of i. The total number of equivalence classes in ∇F is therefore $\kappa(2f-1)+1 \leq 2\kappa f$. □

We are unaware of any non-exponential method for finding κ. Instead we use a greedy heuristic which gives an *upper-bound* on κ. On a benchmark of 19 large multiple-inheritance hierarchies, the median value on that bound was 5, the average was 6.4, and the maximum was 18.

REMARK 4.1. *The actual partitioning $\mathcal{T}_1, \ldots, \mathcal{T}_\kappa$ is* not *required in order to apply the CT reduction; only the integer k is needed for determining the slice size. We found that in practice the single-inheritance analysis closely models even hierarchies which use multiple-inheritance heavily. (Therefore there is no need even to find k.)*

5. Experimental Results

In this section we compare the theoretical prediction on the algorithms with their empirical performance. Our benchmark comprises 35 hierarchies totaling 63,972 types, 70,680 messages and 418,839 methods. Out of these, there were 16 single-inheritance hierarchies with 29,162 types, 12 multiple-inheritance hierarchies with 27,728 types, and 7 multiple-dispatch hierarchies with 7,082 types.

This data-set includes all hierarchies previously used in the literature in benchmarks of dispatching algorithms. However, prior to running the experiments, all degenerate families, i.e., families of size one, were pruned from the input. The reason for doing so is that sending a message whose family is degenerate requires no dispatching, and is the same as static procedure call. (In dynamically typed languages there is an earlier step, which is *equivalent* to a subtyping test, in which it is made sure that the message is valid for the receiver type.)

We stress that by eliminating degenerate families, compression becomes *more difficult* for the CT schemes. The reason is that this pruning reduces both m and ℓ by the same number. Therefore, the optimal compression factor $\iota \equiv (nm)/\ell$, which we aimed at reaching, becomes smaller. On the other hand, the compression factor of $\mathtt{null}$-*elimination schemes* $(nm)/w$ may or may not decrease.

Table 2 gives the essential properties of the pruned hierarchies. The first two row blocks in the table correspond to single-inheritance (SI) and multiple-inheritance (MI) hierarchies. The last block is for hierarchies drawn from multi-dispatch languages. (We regard each multi-dispatch query as several independent single-dispatch queries on each of the arguments, as done in the first step of the major algorithms for multi-dispatching [18].)

	Hierarchy	n	m	$\frac{(nm)}{10^6}$	$\frac{w}{10^3}$	$\frac{\ell}{10^3}$	$\overline{\kappa}$
Single-Inheritance	Visualworks1	774	1,170	0.91	79.14	4.62	1
	Visualworks2	1,956	3,196	6.25	289.67	13.58	1
	Digitalk2	535	962	0.51	72.27	3.33	1
	Digitalk3	1,357	2,402	3.26	362.11	9.44	1
	IBM Smalltalk 2	2,320	4,335	10.06	204.97	16.29	1
	VisualAge 2	3,241	6,529	21.16	594.98	26.21	1
	NextStep	311	499	0.16	16.24	2.12	1
	ET++	371	296	0.11	12.20	1.41	1
	SI: JDK 1.3.1	6,681	4,392	29.34	128.26	23.82	1
	SI: Corba	1,329	222	0.30	6.94	2.59	1
	SI: HotJava	644	690	0.44	23.86	2.91	1
	SI: IBM SF	6,626	11,664	77.29	287.38	88.28	1
	SI: IBM XML	107	131	0.01	1.30	0.59	1
	SI: Orbacus	1,053	980	1.03	18.66	3.82	1
	SI: Orbacus Test	579	368	0.21	5.67	2.39	1
	SI: Orbix	1,278	535	0.68	10.90	2.90	1
Multiple-Inheritance	Self	1,802	2,459	4.43	234.04	21.75	4
	Unidraw	614	360	0.22	8.11	2.33	2
	LOV	436	663	0.29	14.09	2.84	12
	Geode	1,318	1,413	1.86	122.27	9.52	19
	MI: JDK 1.3.1	7,401	5,724	42.36	140.91	28.68	11
	MI: Corba	1,699	396	0.67	13.58	3.20	7
	MI: HotJava	736	829	0.61	24.90	3.40	8
	MI: IBM SF	8,793	14,575	128.16	390.35	116.15	13
	MI: IBM XML	145	271	0.04	2.33	0.95	3
	MI: Orbacus	1,379	1,261	1.74	24.82	5.00	5
	MI: Orbacus Test	689	379	0.26	7.49	2.75	5
	MI: Orbix	2,716	786	2.13	22.44	3.70	4
Multiple-Dispatching	Cecil	932	1,009	0.94	72.89	4.21	7
	Dylan	925	428	0.40	70.38	1.78	3
	Cecil-	473	592	0.28	16.06	2.36	5
	Cecil2	472	131	0.06	17.17	0.56	6
	Harlequin	666	229	0.15	23.11	1.02	8
	Vor3	1,660	328	0.54	15.44	1.86	7
	Vortex3	1,954	476	0.93	305.50	2.50	7

Table 2: Essential parameters of the pruned hierarchies in our data-set

The first two data columns in the table give the values of n and m for each of the hierarchies in the data-set. We see that the hierarchies span a range of sizes: the number of types is between 107 and 8,793 while the number of messages is between 131 and 14,575. A more detailed description of the data-set, including the source of the hierarchies and their respective programming languages is available elsewhere [18].

The column entitled $\frac{nm}{10^6}$ gives the memory requirement of the dispatching matrix, measured in millions of cells. We see that this matrix can be huge. Suppose that each cell uses four bytes (an assumption we make henceforth), then this matrix consumes about 160MB of memory in the MI: JDK 1.3.1 hierarchy and about 500MB in the MI: IBM SF hierarchy.

The next column in the table, entitled $\frac{w}{10^3}$, gives the number of non-null entries in the dispatching matrix, measured in thousands. The column indicates that this matrix is sparse: In most cases, 90% or more of its cells are empty. We shall use this column as a *baseline* for comparison of the CT algorithms, since it shows the memory requirement of an *optimal* null-elimination scheme such as VFT on single-inheritance hierarchies. Note that in hierarchies such as MI: JDK 1.3.1 and MI: IBM SF the potential compression is by a factor of 300 or more. But still, the VFTs may consume a lot of space: 1–2MB on some single-inheritance hierarchies.

The column entitled $\frac{\ell}{10^3}$ gives the number of method implementations, which ranges between 562 and 116,152. This column also sets a lower bound on the memory used by an *optimal* duplicates-elimination compression scheme. Comparing this column to the previous one, we learn that duplicates-elimination is potentially much better than null-elimination. However, it is much more difficult to come close to optimal duplicates-elimination than to optimal null-elimination. We shall use this column as another comparison standard for the performance of the CT algorithms.

The final column entitled $\overline{\kappa}$ shows an upper bound on κ which was found by our greedy heuristic. (Recall that we do not have an efficient algorithm for computing κ.) In single-inheritance hierarchies, $\kappa = \overline{\kappa} = 1$. The median of $\overline{\kappa}$ in the remaining hierarchies is 7. The hierarchy whose topology seems to be the most complex is Geode, followed by MI: IBM SF , LOV and then JDK 1.3.1.

The implementation of the various CT schemes was run on 900Mhz Pentium III computer, equipped with 256MB internal memory and controlled by a Windows 2000 operating system. On this machine, the runtimes for generating the encoding (without actually copying the values into matrices) of the first four schemes (CT_2 through CT_5) were 0.7 Sec, 1.4 Sec, 2.1 Sec and 2.9 Sec. Since our data-set included in total 418,839 methods we find that the time per implementation is measured in microseconds. For example, we found that the creation time *per implementation* ranged between 0.3 and 1.7 μSec in CT_2 in single-inheritance hierarchies (the median being 0.6 μSec). These times increase in multiple-inheritance hierarchies: the range being 1.1 to 6.7 μSec; the median being 2.4 μSec.

Figure 5.1 shows the memory used by the first four CT schemes relative to the w baseline in the 35 hierarchies in the data-set. Memory usage of the CT schemes were obtained using the empirically found *best* slice size (which may be different than the prescription of column 2 of Table 1).

The figure shows that compared to the *optimal* null-elimination, CT_2 is better in 6 hierarchies, CT_3 in 13 hierarchies, CT_4 in 15 hierarchies, and CT_5 in 16 hierarchies. In a few cases, the improvement is by an order of magnitude from the baseline. We also see that CT_2 is at most one order of magnitude worse than this idealized baseline.

We can also learn from Figure 5.1 that the incremental improvement by the series of CT schemes is diminishing. In fact, examining the actual memory requirements, we find that the median incremental improvements are: CT_3 over CT_2: 44%, CT_4 over CT_3: 18%, and CT_5 over CT_4: 8%. This finding is consistent with the theoretical prediction.

The figure also plots another idealized algorithm, i.e., the optimal duplicates-elimination scheme, which uses ℓ cells. We see that this ideal is about one order of magnitude better than the various CT schemes. Finally, we see a certain correlation between ℓ and the series of CT schemes, as predicted by the theoretical analysis. When $\ell \ll w$ the CT schemes outperform even an optimal null-elimination scheme.

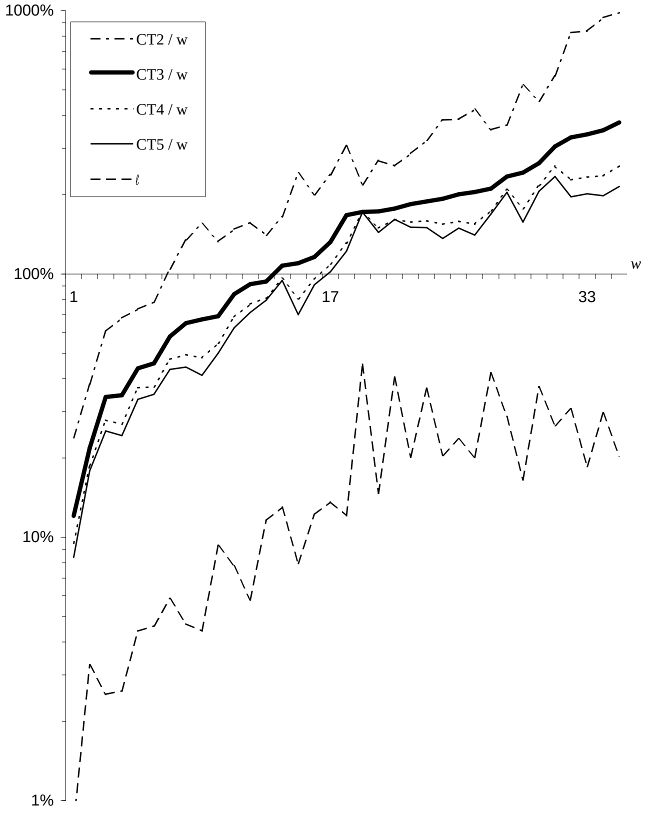


Figure 5.1: Memory used by CT_2, CT_3, CT_4, CT_5 and optimal duplicates-elimination (ℓ) relative to optimal null-elimination (w – marked as the 100%); hierarchies are sorted in ascending memory used by CT_3

We now turn to comparing the actual performance of the various CT schemes with the theoretically obtained bounds.

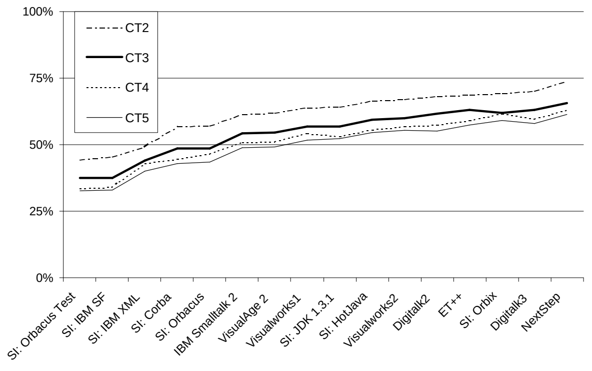
In single-inheritance hierarchies, the upper bound on the memory requirement are given by the fourth column of Table 1. Figure 5.2a shows the memory requirement relative to these values. We see that in all schemes and in all hierarchies, the memory requirement is significantly smaller than the upper bounds. Also, the extent of improvement of CT_d over the upper bound increases with d.

Corollary 4.2 provides the upper bounds in multiple-inheritance hierarchies depending on their complexity κ. Figure 5.2b shows the memory, relative to these upper bounds, of the actual CT performance. Again, we see that the extent of improvement of CT_d over the upper bound increases with d. Interestingly, in comparing Figure 5.2b with Figure 5.2a, we see that the improvement of the implementation upon the upper bound is much greater in multiple-inheritance vs. single-inheritance hierarchies.

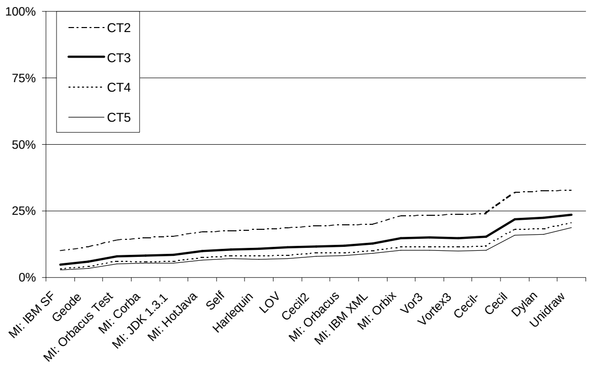
A possible explanation for this seemingly better performance in multiple-inheritance hierarchies is exaggerated upper bounds. Examining Corollary 4.2, we see that the upper bounds increase with κ. Since our heuristics only finds an upper approximation of κ, it could be that the true upper bounds are actually smaller, and hence the improvement upon the upper bound is not as great.

Figure 5.2c tries to test this hypothesis, by comparing the performance on multiple-inheritance hierarchies with the upper bounds obtained by assuming $\kappa = 1$ (as in single-inheritance hierarchies).[4] We see that the improvement upon the upper bounds computed thus is almost the same as in single-inheritance hierarchies (Figure 5.2a). Such a similarity could not be explained by an overestimation of κ.

(a)



(b)



(c)

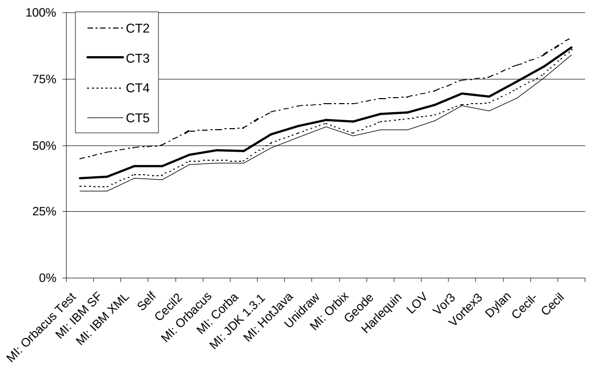


Figure 5.2: The memory requirement of CT_2, CT_3, CT_4 and CT_5 relative to the theoretically obtained upper bounds in single-inheritance hierarchies (a), multiple-inheritance hierarchies where the upper bound was computed using $\overline{\kappa}$ (b), and multiple-inheritance hierarchies, where the upper bound is computed as in single-inheritance hierarchies ($\kappa = 1$) (c)

The reason that algorithms perform better than the theoretically obtained is that the analysis of the CT reduction bounded the size of a master-family by the sum of sizes of its constituents, i.e.,

$$|F'_i| = \left| \bigcup_{F \in \mathcal{F}_i} F \right| \leq \sum_{F \in \mathcal{F}_i} |F|.$$

In fact, especially when the families are large, the probability of finding shared elements may be significant, and the master family is likely to be smaller. As a result, ℓ', the number of implementations after the reduction, may be much smaller than the original value ℓ. For example, with $x = 29$ for CT_2 in Digitalk3, the CT reduction transforms the problem $\langle n, m, \ell \rangle = \langle 1357, 2402, 9444 \rangle$ to $\langle 1357, 83, 4616 \rangle$, i.e., the number of implementations decreased by a factor of more than 2. Our analysis assumed (see (2.7)) however that $\ell' = \ell$.

This effect increases also with slice size, which is the reason that choosing a slice size greater than the theoretical prescription may improve the performance of the reduction. In IBM SF, for example, the theoretical analysis suggested that $x_{\text{OPT}} = 30$ as optimal slice size for CT_2. However, by using instead a slice size $x = 70$, we were able to further reduce the number of cells from 3.3M to about 2.4M.

Figure 5.3 compares the actual memory used by the CT_2 scheme with the theoretical prediction (2.10) in the Digitalk3 hierarchy. (The graphs of other hierarchies and higher order schemes are similar.) We see that the extent by which the empirical performance is superior to the theoretically obtained bound increases with the slice size.

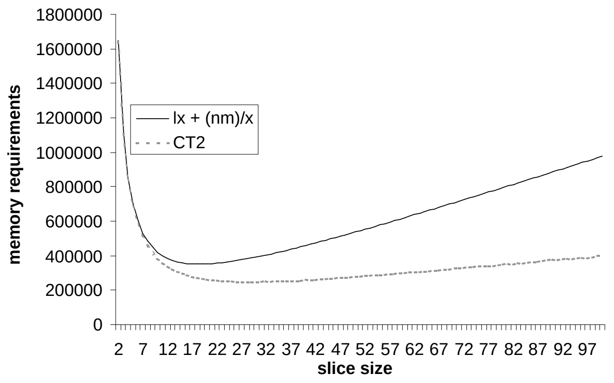


Figure 5.3: Space requirements vs. slice size in the single-inheritance hierarchy of Digitalk3 for CT_2 and its theoretical upper bound (2.10)

6. Conclusions and Open Problems

The incremental algorithm described in Section 3 is in many ways the pinnacle of this paper. This algorithm assumes the single-inheritance, dynamically typed, and dynamic loading model, denoted SDTDL. A prime example for the model is the SMALLTALK programming language. Note that the VFT method is unsuitable in an SDTDL model.

Curiously, even though JAVA is in essence a statically typed language, the implementation of the `invokeinterface` bytecode instruction is a very close match of this model. To see this recall that all implementations of a method defined in an **`interface`** must reside in **`class`**es, and that these classes take a tree topology. The locations of these implementations in this tree are however totally unrelated, and additional implementations can be introduced as a result of dynamic class loading. Even though there is a

[4] In fact, we used the bound for single-inheritance in Table 1, which is smaller by a factor of $2^{1-1/d}$ than the bound for multiple-inheritance in Corollary 4.2.

possibility of using static information of the interface type, many implementations of the `invokeinterface` bytecode instruction assume an SDTDL model.

Incorporating the algorithm into a runtime system requires careful attention to details, including selecting a heuristic of determining the optimal slice size, which might perform better than the theoretical value, a wise strategy for background copy to avoid stagnation, tweaking and fine tuning of the partitioning algorithm, etc. We leave this empirical evaluation to continuing research.

Also, the incremental algorithm can be generalized to the multiple-inheritance setting, but there are subtle issues in the theoretical analysis of the performance of this generalization.

Observe that the static algorithm for multiple-inheritance hierarchies, achieves $2\kappa\ell \lg m$ space when $d = \lg m$. Type slicing [18] however uses only $O(\kappa\ell)$ cells, while achieving $O(\lg \lg n)$ dispatching time. There is therefore a reason to believe that the trade-off offered by our technique can be improved, especially for higher values of d.

7. REFERENCES

[1] K. Arnold and J. Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, Reading, Massachusetts, 1996.

[2] P. Deutsch and A. Schiffman. Efficient implementation of the Smalltalk-80 system. In *11th Symposium on Principles of Programming Languages, POPL'84*, pages 297–302, Salt Lake City, Utah, Jan. 1984. ACM SIGPLAN — SIGACT, ACM Press.

[3] M. Dietzfelbinger, A. R. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, Aug. 1994.

[4] R. Dixon, T. McKee, M. Vaughan, and P. Schweizer. A fast method dispatcher for compiled languages with multiple inheritance. In *Proceedings of the 4th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 211–214, New Orleans, Louisiana, Oct. 1-6 1989. OOPSLA'89, ACM SIGPLAN Notices 24(10) Oct. 1989.

[5] K. Driesen. Selector table indexing & sparse arrays. In *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 259–270, Washington, DC, USA, Sept. 26 - Oct. 1 1993. OOPSLA'93, ACM SIGPLAN Notices 28(10) Oct. 1993.

[6] K. Driesen and U. Hölzle. Minimizing row displacement dispatch tables. In *Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 141–155, Austin, Texas, USA, Oct. 15-19 1995. OOPSLA'95, ACM SIGPLAN Notices 30(10) Oct. 1995.

[7] P. Ferragina and S. Muthukrishnan. Efficient dynamic method-lookup for object oriented languages. In J. Díaz and M. Serna, editors, *Algorithms—ESA '96, Fourth Annual European Symposium*, volume 1136 of *Lecture Notes in Computer Science*, pages 107–120, Barcelona, Spain, 25–27 Sept. 1996. Springer.

[8] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31(3):538–544, July 1984.

[9] J. Y. Gil and P. Sweeney. Space- and time-efficient memory layout for multiple inheritance. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 256–275, Denver, Colorado, Nov.1–5 1999. OOPSLA'99, ACM SIGPLAN Notices 34(10) Nov. 1999.

[10] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the 5th European Conference on Object-Oriented Programming*, number 512 in Lecture Notes in Computer Science, Geneva, Switzerland, July15–19 1991. ECOOP'91, Springer Verlag.

[11] S. Muthukrishnan and M. Müller. Time and space efficient method-lookup for object-oriented programs. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 42–51, New York/Philadelphia, Jan. 28–30 1996. ACM/SIAM.

[12] A. Royer. Optimizing Method Search with Lookup Caches and Incremental Coloring. In *Proceedings of the 7th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 110–126, Vancouver, British Columbia, Canada, Oct.18-22 1992. OOPSLA'92, ACM SIGPLAN Notices 27(10) Oct. 1992.

[13] D. Sleator and R. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, July 1985.

[14] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, Reading, Massachusetts, Mar. 1994.

[15] J. Vitek and R. N. Horspool. Taming message passing: Efficient method lookup for dynamically typed object-oriented languages. In *Proceedings of the 8th European Conference on Object-Oriented Programming*, number 821 in Lecture Notes in Computer Science, Bologna, Italy, July 4-8 1994. ECOOP'94, Springer Verlag.

[16] J. Vitek and R. N. Horspool. Compact dispatch tables for dynamically typed object oriented languages. In T. Gyimothy, editor, *Compiler Construction, 6th International Conference*, volume 1060 of *Lecture Notes in Computer Science*, pages 309–325, Linköping, Sweden, 24–26 Apr. 1996. Springer.

[17] O. Zendra, C. Colnet, and S. Collin. Efficient dynamic dispatch without virtual function tables: The SmallEiffel compiler. In *Proceedings of the 12th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 125–141, Atlanta, Georgia, Oct. 5-9 1997. OOPSLA'97, ACM SIGPLAN Notices 32(10) Oct. 1997.

[18] Y. Zibin and J. Y. Gil. Fast algorithm for creating space efficient dispatching tables with application to multi-dispatching. In *Proceedings of the 17th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 142–160, Seattle, Washington, Nov. 4–8 2002. OOPSLA'02, ACM SIGPLAN Notices 37(10) Nov. 2002.