Time and Space Efficient Multi-Method Dispatching

Stephen Alstrup[1], Gerth Stølting Brodal[2]*, Inge Li Gørtz[1], and Theis Rauhe[1]

[1] The IT University of Copenhagen, Glentevej 67, DK-2400 Copenhagen NV,
Denmark. E-mail: `{stephen,inge,theis}@it-c.dk`
[2] BRICS (Basic Research in Computer Science), Center of the Danish National
Research Foundation, Department of Computer Science, University of Aarhus, Ny
Munkegade, DK-8000 Århus C, Denmark. E-mail: `gerth@brics.dk`.

Abstract. The *dispatching problem* for object oriented languages is the problem of determining the most specialized method to invoke for calls at run-time. This can be a critical component of execution performance. A number of recent results, including [Muthukrishnan and Müller SODA'96, Ferragina and Muthukrishnan ESA'96, Alstrup et al. FOCS'98], have studied this problem and in particular provided various efficient data structures for the *mono-method* dispatching problem. A recent paper of Ferragina, Muthukrishnan and de Berg [STOC'99] addresses the *multi-method* dispatching problem.

Our main result is a linear space data structure for *binary* dispatching that supports dispatching in logarithmic time. Using the same query time as Ferragina et al., this result improves the space bound with a logarithmic factor.

1 Introduction

The *dispatching problem* for object oriented languages is the problem of determining the most specialized method to invoke for a method call. This specialization depends on the actual arguments of the method call at run-time and can be a critical component of execution performance in object oriented languages. Most of the commercial object oriented languages rely on dispatching of methods with only one argument, the so-called *mono-method* or *unary dispatching problem*. A number of papers, see e.g.,[10, 15] (for an extensive list see [11]), have studied the unary dispatching problem, and Ferragina and Muthukrishnan [10] provide a linear space data structure that supports unary dispatching in log-logarithmic time. However, the techniques in these papers do not apply to the more general *multi-method dispatching problem* in which more than one method argument are used for the dispatching. Multi-method dispatching has been identified as a powerful feature in object oriented languages supporting multi-methods such

---

* Supported by the Carlsberg Foundation (contract number ANS-0257/20). Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

as Cecil [3], CLOS [2], and Dylan [4]. Several recent results have attempted to deal with d-ary dispatching in practice (see [11] for an extensive list). Ferragina et al. [11] provided the first non-trivial data structures, and, quoting this paper, several experimental object oriented languages' "ultimately success and impact in practice depends, among other things, on whether multi-method dispatching can be supported efficiently".

Our result is a *linear space* data structure for the *binary dispatching* problem, i.e., multi-method dispatching for methods with at most two arguments. Our data structure uses *linear space* and supports dispatching in logarithmic time. Using the same query time as Ferragina et al. [11], this result improves the space bound with a logarithmic factor. Before we provide a precise formulation of our result, we will formalize the general d-ary dispatching problem.

Let T be a rooted tree with N nodes. The tree represents a class hierarchy with nodes representing the classes. T defines a partial order $\prec$ on the set of classes: $A \prec B \Leftrightarrow A$ is a descendant of B (not necessarily a proper descendant). Let $\mathcal{M}$ be the set of methods and let m denote the number of methods and M the number of distinct method names in $\mathcal{M}$. Each method takes a number of classes as arguments. A method invocation is a query of the form $s(A_1, \ldots, A_d)$ where s is the name of a method in $\mathcal{M}$ and $A_1, \ldots, A_d$ are class instances. A method $s(B_1, \ldots, B_d)$ is *applicable* for $s(A_1, \ldots, A_d)$ if and only if $A_i \prec B_i$ for all i. The *most specialized method* is the method $s(B_1, \ldots, B_d)$ such that for every other applicative method $s(C_1, \ldots, C_d)$ we have $B_i \prec C_i$ for all i. This might be ambiguous, i.e., we might have two applicative methods $s(B_1, \ldots, B_d)$ and $s(C_1, \ldots, C_d)$ where $B_i \neq C_i$, $B_j \neq C_j$, $B_i \prec C_i$, and $C_j \prec B_j$ for some indices $1 \leq i, j \leq d$. That is, neither method is more specific than the other. Multi-method dispatching is to find the most specialized applicable method in $\mathcal{M}$ if it exists. If it does not exist or in case of ambiguity, "no applicable method" resp. "ambiguity" is reported instead.

The *d-ary dispatching problem* is to construct a data structure that supports multi-method dispatching with methods having up to d arguments, where $\mathcal{M}$ is static but queries are online. The cases $d = 1$ and $d = 2$ are the *unary* and *binary dispatching* problems respectively. In this paper we focus on the binary dispatching problem which is of "particular interest" quoting Ferragina et al. [11].

The input is the tree T and the set of methods. We assume that the size of T is $O(m)$, where m is the number of methods. This is not a necessary restriction but due to lack of space we will not show how to remove it here.

Results Our main result is a data structure for the binary dispatching problem using $O(m)$ space and query time $O(\log m)$ on a unit-cost RAM with word size logarithmic in N with $O(N + m\,(\mathrm{loglog}\, m)^2)$ time for preprocessing. By the use of a reduction to a geometric problem, Ferragina et al. [11], obtain similar time bounds within space $O(m \log m)$. Furthermore they show how the case $d = 2$ can be generalized for $d > 2$ at the cost of factor $\log^{d-2} m$ in the time and space bounds.

Our result is obtained by a very different approach in which we employ a dynamic to static transformation technique. To solve the binary dispatching problem we turn it into a unary dispatching problem — a variant of the marked ancestor problem as defined in [1], in which we maintain a dynamic set of methods. The unary problem is then solved persistently. We solve the persistent unary problem combining the technique by Dietz [5] to make a data structure fully persistent and the technique from [1] to solve the marked ancestor problem. The technique of using a persistent dynamic one-dimensional data structure to solve a static two-dimensional problem is a standard technique [17]. What is new in our technique is that we use the class hierarchy tree to denote the time (give the order on the versions) to get a fully persistent data structure. This gives a "branching" notion for time, which is the same as what one has in a fully persistent data structure where it is called the version tree. This technique is different from the plane sweep technique where a plane-sweep is used to give a partially persistent data structure. A top-down tour of the tree corresponds to a plane-sweep in the partially persistent data structures.

Related and Previous Work For the unary dispatching problem the best known bound is $O(N+m)$ space and $O(\text{loglog}\, N)$ query time [15, 10]. For the d-ary dispatching, $d \geq 2$, the result of Ferragina et al. [11] is a data structure using space $O(m\ (t\ \log\ m/\log\ t)^{d-1})$ and query time $O((\log\ m/\log\ t)^{d-1}\text{loglog}\, N)$, where t is a parameter $2 \leq t \leq m$. For the case $t = 2$ they are able to improve the query time to $O(\log^{d-1} m)$ using fractional cascading. They obtain their results by reducing the dispatching problem to a point-enclosure problem in d dimensions: Given a point q, check whether there is a smallest rectangle containing q. In the context of the geometric problem, Ferragina et al. also present applications to approximate dictionary matching.

In [9] Eppstein and Muthukrishnan look at a similar problem which they call *packet classification*. Here there is a database of m filters available for preprocessing. Each query is a packet P, and the goal is to *classify* it, that is, to determine the filter of highest priority that applies to P. This is essentially the same as the multiple dispatching problem. For $d = 2$ they give an algorithm using space $O(m^{1+o(1)})$ and query time $O(\text{loglog} m)$, or $O(m^{1+\varepsilon})$ and query time $O(1)$. They reduce the problem to a geometric problem, very similar to the one in [11]. To solve the problem they use a standard plane-sweep approach to turn the static two-dimensional rectangle query problem into a dynamic one-dimensional problem,which is solved persistently such that previous versions can be queried after the plane sweep has occurred.

2 Preliminaries

In this section we give some basic concepts which are used throughout the paper.

Definition 1 (*Trees*). *Let T be a rooted tree. The set of all nodes in T is denoted $V(T)$. The nodes on the unique path from a node v to the root are denoted $\pi(v)$, which includes v and the root. The nodes $\pi(v)$ are called the* ancestors *of v. The* descendants *of a node v are all the nodes u for which $v \in \pi(u)$. If $v \neq u$ we say that u is a* proper *descendant of v. The distance dist(v,w) between two* *nodes in T is the number of edges on the unique path between v and w. In the rest of the paper all trees are rooted trees.*

Let C be a set of colors. A labeling $l(v)$ of a node $v \in V(T)$ is a subset of C, i.e., $l(v) \subseteq C$. A labeling $l : V(T) \rightarrow 2^C$ of a tree T is a set of labelings for the nodes in T.

Definition 2 (***Persistent data structures***)**.** *The concept of persistent data structures was introduced by Driscoll et al. in [8]. A data structure is* partially persistent *if all previous versions remain available for queries but only the newest version can be modified. A data structure is* fully persistent *if it allows both queries and updates of previous versions. An update may operate only on a single version at a time, that is, combining two or more versions of the data structure to form a new one is not allowed. The versions of a fully persistent data structure form a tree called the* version tree*. Each node in the version tree represents the result of one update operation on a version of the data structure. A persistent update or query take as an extra argument the version of the data structure to which the query or update refers.*

Known results. Dietz [5] showed how to make any data structure fully persistent on a unit-cost RAM. A data structure with worst case query time $O(Q(n))$ and update time $O(F(n))$ making worst case $O(U(n))$ memory modifications can be made fully persistent using $O(Q(n) \operatorname{loglog} n)$ worst case time per query and $O(F(n) \operatorname{loglog} n)$ expected amortized time per update using $O(U(n) \operatorname{loglog} n)$ space.

Definition 3 (***Tree color problem***)**.**

Let T be a rooted tree with n nodes, where we associate a set of colors with each node of T. The tree color problem *is to maintain a data structure with the following operations:*

color(v,c): add c to v's set of colors, i.e., $l(v) \leftarrow l(v) \cup \{c\}$,
uncolor(v,c): remove c from v's set of colors, i.e., $l(v) \leftarrow l(v) \setminus \{c\}$,
findfirstcolor(v,c): find the first ancestor of v with color c (this may be v itself).

The incremental *version of this problem does not support uncolor, the* decremental *problem does not support color, and the* fully dynamic *problem supports both update operations.*

Known results. In [1] it is showed how to solve the tree color problem on a RAM with logarithmic word size in expected update time $O(\operatorname{loglog} n)$ for both *color* and *uncolor*, query time $O(\log n/\operatorname{loglog} n)$, using linear space and preprocessing time. The expected update time is due to hashing. Thus the expectation can be removed at the cost of using more space. We need worst case time when we make the data structure persistent because data structures with amortized/expected time may perform poorly when made fully persistent, since expensive operations might be performed many times.

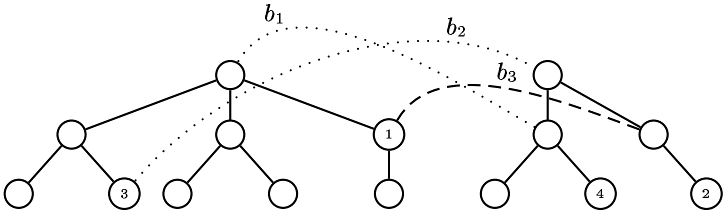

Fig. 1. The solid lines are tree edges and the dashed and dotted lines are bridges of color c and c', respectively. *firstcolorbridge*(c,v_1,v_2) returns b_3. *firstcolorbridge*(c',v_3,v_4) returns ambiguity since neither b_1 or b_2 is closer than the other.

Dietz [5] showed how to solve the incremental tree color problem in $O(\text{loglog}n)$ amortized time per operation using linear space, when the nodes are colored top-down and each node has at most one color.

The unary dispatching problem is the same as the *tree color problem* if we let each color represent a method name.

Definition 4. *We need a data structure to support insert and predecessor queries on a set of integers from* $\{1, \ldots, n\}$*. This can be solved in worst case* $O(\text{loglog}\, n)$ *time per operation on a RAM using the data structure of van Emde Boas [18] (VEB). We show how to do modify this data structure such that it only uses* $O(1)$ *memory modifications per update.*

3 The Bridge Color Problem

The binary dispatching problem ($d = 2$) can be formulated as the following tree problem, which we call the *bridge color problem.*

Definition 5 (*Bridge Color Problem*). *Let* T_1 *and* T_2 *be two rooted trees. Between* T_1 *and* T_2 *there are a number of bridges of different colors. Let* C *be the set of colors. A bridge is a triple* $(c, v_1, v_2) \in C \times V(T_1) \times V(T_2)$ *and is denoted by* $c(v_1, v_2)$*. If* $v_1 \in \pi(u_1)$ *and* $v_2 \in \pi(u_2)$ *we say that* $c(v_1, v_2)$ *is a bridge* over (u_1, u_2)*. The* bridge color problem *is to construct a data structure which supports the query firstcolorbridge(c,*v_1*,*v_2*). Formally, let* B *be the subset of bridges* $c(w_1, w_2)$ *of color* c *where* w_1 *is an ancestor of* v_1*, and* w_2 *an ancestor of* v_2*. If* $B = \emptyset$ *then firstcolorbridge(c,*v_1*,*v_2*) = NIL. Otherwise, let* $b_1 = c(w_1, w_1') \in B$*, such that dist(*v_1*,*w_1*) is minimal and* $b_2 = c(w_2', w_2) \in B$*, such that dist(*v_2, w_2*) is minimal. If* $b_1 = b_2$ *then firstcolorbridge(c,*v_1*,*v_2*)=* b_1 *and we say that* b_1 *is the* first bridge *over* (v_1, v_2)*, otherwise firstcolorbridge(c,*v_1*,*v_2*) = "ambiguity". See Fig. 1.*

The binary dispatching problem can be reduced to the bridge color problem the following way. Let T_1 and T_2 be copies of the tree T in the binary dispatching problem. For every method $s(v_1, v_2) \in \mathcal{M}$ make a bridge of color s between $v_1 \in V(T_1)$ and $v_2 \in V(T_2)$.

The problem is now to construct a data structure that supports *firstcolorbridge.* The object of the remaining of this paper is show the following theorem:

Theorem 1. *Using expected* $O(m\ (\text{loglog}m)^2)$ *time for preprocessing and* $O(m)$ *space, firstcolorbridge can be supported in worst case time* $O(\log\ m)$ *per operation, where* m *is the number of bridges.*

4 A Data Structure for the Bridge Color Problem

Let B be a set of bridges ($\mid B \mid= m$) for which we want to construct a data structure for the bridge color problem. As mentioned in the introduction we can assume that the number of nodes in the trees involved in the bridge color problem is $O(m)$, i.e., $\mid V(T_1) \mid + \mid V(T_2) \mid= O(m)$. In this section we present a data structure that supports *firstcolorbridge* in $O(\log\ m)$ time per query using $O(m)$ space for the bridge color problem.

For each node $v \in V(T_1)$ we define the labeling l_v of T_2 as follows. The labeling of a node $w \in V(T_2)$ contains color c if w is the endpoint of a bridge of color c with the other endpoint among ancestors of v. Formally, $c \in l_v(w)$ if and only if there exists a node $u \in \pi(v)$ such that $c(u, w) \in B$. Similar define the symmetric labelings for T_1. In addition to each labeling l_v, we need to keep the following extra information stored in a sparse array $H(v)$: For each pair $(w, c) \in V(T_2) \times C$, where $l_v(w)$ contains color c, we keep the node v' of maximal depth in $\pi(v)$ from which there is a bridge $c(v', w)$ in B. Note that this set is sparse, i.e., we can use a sparse array to store it.

For each labeling l_v of T_2, where $v \in V(T_1)$, we will construct a data structure for the static tree color problem. That is, a data structure that supports the query *findfirstcolor*(u,c) which returns the first ancestor of u with color c. Using this data structure we can find the first bridge over $(u, w) \in V(T_1) \times V(T_2)$ of color c by the following queries.

In the data structure for the labeling l_u of the tree T_2 we perform the query *findfirstcolor*(w,c). If this query reports NIL there is no bridge to report, and we can simply return NIL. Otherwise let w' be the reported node. We make a lookup in $H(u)$ to determine the bridge b such that $b = c(u', w') \in B$. By definition b is the bridge over (u, w') with minimal distance between w and w'. But it is possible that there is a bridge (u'', w'') over (u, w) where *dist*$(u,u'') <$ *dist*(u,u'). By a symmetric computation with the data structure for the labeling $l(w)$ of T_1 we can detect this in which case we return “ambiguity”. Otherwise we simply return the unique first bridge b.

Explicit representation of the tree color data structures for each of the labelings l_v for nodes v in T_1 and T_2 would take up space $O(m^2)$. Fortunately, the data structures overlap a lot: Let $v, w \in V(T_1)$, $u \in V(T_2)$, and let $v \in \pi(w)$. Then $l_v(u) \in l_w(u)$. We take advantage of this in a simple way. We make a fully persistent version of the *dynamic* tree color data structure using the technique of Dietz [5]. The idea is that the above set of $O(m)$ tree color data structures corresponds to a persistent, survived version, each created by one of $O(m)$ updates in total.

Formally, suppose we have generated the data structure for the labeling l_v, for v in T_1. Let w be the child of node v in T_1. We can then construct the data structure for the labeling l_w by simply updating the persistent structure for l_v by inserting the color marks corresponding to all bridges with endpoint w (including updating $H(v)$). Since the data structure is fully persistent, we can repeat this for each child of v, and hence obtain data structures for all the labelings for children of v. In other words, we can form all the data structures for the labeling l_v for nodes $v \in V(T_1)$, by updates in the persistent structures according to a top-down traversal of T_1. Another way to see this, is that T_1 is denoting the time (give the order of the versions). That is, the version tree has the same structure as T_1.

Similar we can construct the labelings for T_1 by a similar traversal of T_2. We conclude this discussion by the following lemma.

Lemma 1. *A static data structure for the bridge color problem can be constructed by $O(m)$ updates to a fully persistent version of the dynamic tree color problem.*

4.1 Reducing the Memory Modifications in the Tree Color Problem

The paper [1] gives the following upper bounds for the tree color problem for a tree of size m. Update time expected $O(\text{loglog}\, m)$ for both *color* and *uncolor*, and query time $O(\log\, m/\text{loglog}\, m)$, with linear space and preprocessing time.

For our purposes we need a slightly stronger result, i.e., updates that only make worst case $O(1)$ memory modifications. By inspection of the dynamic tree color algorithm, the bottle-neck in order to achieve this, is the use of the van Emde Boas predecessor data structure [18] (VEB). Using a standard technique by Dietz and Raman [6] to implement a fast predecessor structure we get the following result.

Theorem 2. *Insert and predecessor queries on a set of integers from $\{1,\ldots,n\}$ can be performed in $O(\text{loglog}\, n)$ worst case time per operation using worst case $O(1)$ memory modifications per update.*

To prove the theorem we first show an amortized result[3]. The elements in our predecessor data structure is grouped into buckets $S_1,\ldots,S_k$, where we maintain the following invariants:

$$\begin{array}{ll} (1)\ \max\, S_i < \min\, S_{i+1} & \text{for } i = 1,\ldots k-1, \text{ and} \\ (2)\ 1/2\ \log\, n < |S_i| \le 2\ \log\, n & \text{for all } i. \end{array}$$

We have $k \in O(n/\ \log\, n)$. Each S_i is represented by a balanced search tree with $O(1)$ worst case update time once the position of the inserted or deleted element is known and query time $O(\log\, m)$, where m is the number of nodes in the tree [12, 13]. This gives us update time $O(\text{loglog}\, n)$ in a bucket, but only $O(1)$ memory modifications per update. The minimum element s_i of each bucket S_i is stored in a VEB.

When a new element x is inserted it is placed in the bucket S_i such that $s_i < x < s_{i+1}$, or in S_1 if no such bucket exists. Finding the correct bucket is done by a predecessor query in the VEB. This takes $O(\text{loglog}\, n)$ time. Inserting the element in the bucket also takes $O(\text{loglog}\, n)$ time, but only $O(1)$ memory

---

[3] The amortized result (Lemma 2) was already shown in [14], bur in order to make the deamortization we give another implementation here.

modifications. When a bucket S_i becomes to large it is split into two buckets of half size. This causes a new element to be inserted into the VEB and the binary trees for the two new buckets have to be build. An insertion into the VEB takes $O(\text{loglog } n)$ time and uses the same number of memory modifications. Building the binary search trees uses $O(\log\ n)$ time and the same number of memory modifications. When a bucket is split there must have been at least $\log\ n$ insertions into this bucket since it last was involved in a split. That is, splitting and inserting uses $O(1)$ amortized memory modifications per insertion.

Lemma 2. *Insert and predecessor queries on a set of integers from $\{1, \ldots, n\}$ can be performed in $O(\text{loglog}\, n)$ worst case time for predecessor and $O(\text{loglog}\, n)$ amortized time for insert using $O(1)$ amortized number of memory modifications per update.*

We can remove this amortization at the cost of making the bucket sizes $\Theta(\log^2 n)$ by the following technique by Raman [16] called thinning.

Let $\alpha > 0$ be a sufficiently small constant. Define the *criticality* of a bucket to be: $\rho(b) = \frac{1}{\alpha\ \log\ n}\max\{0, \text{size}(b) - 1.8\ \log^2\ n\}$. A bucket b is called *critical* if $\rho(b) > 0$. We want to ensure that $\text{size}(b) \leq 2\ \log^2\ n$. To maintain the size of the buckets every $\alpha\ \log\ n$ updates take the most critical bucket (if there is any) and move $\log\ n$ elements to a newly created empty adjacent bucket. A bucket rebalancing uses $O(\log\ n)$ memory modifications and we can thus perform it with $O(1)$ memory modifications per update spread over no more than $\alpha\ \log\ n$ updates.

We now show that the buckets never get too big. The criticality of all buckets can only increase by 1 between bucket rebalancings. We see that the criticality of the bucket being rebalanced is decreased, and no other bucket has its criticality increased by the rebalancing operations. We make use of the following lemma due to Raman:

Lemma 3 (*Raman*). *Let $x_1, \ldots, x_n$ be real-valued variables, all initially zero. Repeatedly do the following:*

(1) Choose n non-negative real numbers $a_1, \ldots, a_n$ such that $\sum_{i=1}^{n} a_i = 1$, and set $x_i \leftarrow x_i + a_i$ for $1 \leq i \leq n$.
(2) Choose an x_i such that $x_i = max_j\{x_j\}$, and set $x_i \leftarrow max\{x_i - c, 0\}$ for some constant $c \geq 1$.

Then each x_i will always be less than $ln\ n + 1$, even when $c = 1$.

Apply the lemma as follows: Let the variables of Lemma 3 be the criticalities of the buckets. The reals a_i are the increases in the criticalities between rebalancings and $c = 1/\alpha$. We see that if $\alpha \leq 1$ the criticality of a bucket will never exceed $\ln + 1 = O(\log\ n)$. Thus for sufficiently small α the size of the buckets will never exceed $2\ \log^2\ n$. This completes the proof of Theorem 2.

We need worst case update time for *color* in the tree color problem in order to make it persistent. The expected update time is due to hashing. The expectation can be removed at the cost of using more space. We now use Theorem 2 to get the following lemma.

Lemma 4. *Using linear time for preprocessing, we can maintain a tree with complexity* $O(\text{loglog}\, n)$ *for color and complexity* $O(\log\, n/\text{loglog}\, n)$ *for findfirstcolor, using* $O(1)$ *memory modifications per update, where* n *is the number of nodes in the tree.*

4.2 Reducing the Space

Using Dietz' method [5] to make a data structure fully persistent on the data structure from Lemma 4, we can construct a fully persistent version of the tree color data structure with complexity $O((\text{loglog}\, m)^2)$ for *color* and *uncolor*, and complexity $O((\log\, m/\text{loglog}\, m) \cdot \text{loglog}\, m) = O(\log\, m)$ for *findfirstcolor*, using $O(m)$ memory modifications, where m is the number of nodes in the tree.

According to Lemma 1 a data structure for the first bridge problem can be constructed by $O(m)$ updates to a fully persistent version of the dynamic tree color problem. We can thus construct a data structure for the bridge color problem in time $O(m\ (\text{loglog}\, m)^2)$, which has query time $O(\log\, m)$, where m is the number of bridges.

This data structure might use $O(c{\cdot}m)$ space, where c is the number of method names. We can reduce this space usage using the following lemma.

Lemma 5. *If there exists an algorithm* A *constructing a* static *data structure* D *using expected* $t(n)$ *time for preprocessing and expected* $m(n)$ *memory modifications and has query time* $q(n)$*, then there exists an algorithm constructing a data structure* D' *with query time* $O(q(n))$*, using expected* $O(t(n))$ *time for preprocessing and space* $O(m(n))$*.*

Proof. The data structure D' can be constructed the same way as D using dynamic perfect hashing [7] to reduce the space. □

Since we only use $O(m)$ memory modifications to construct the data structure for the bridge color problem, we can construct a data structure with the same query time using only $O(m)$ space. This completes the proof of Theorem 1.

If we use $O(N)$ time to reduce the class hierarchy tree to size $O(m)$ as mentioned in the introduction, we get the following corollary to Theorem 1.

Corollary 1. *Using* $O(N + m\ (\text{loglog}\, m)^2)$ *time for preprocessing and* $O(m)$ *space, the* multiple dispatching problem *can be solved in worst case time* $O(\log m)$ *per query. Here* N *is the number of classes and* m *is the number of methods.*

References

1. S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems (extended abstract). In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 534–543, 1998.
2. D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common LISP object system specification X3J13 document 88-002R. *ACM SIGPLAN Notices*, 23, 1988. Special Issue, September 1988.

3. Craig Chambers. Object-oriented multi-methods in Cecil. In Ole Lehrmann Madsen, editor, *ECOOP '92, European Conference on Object-Oriented Programming, Utrecht, The Netherlands*, volume 615 of *Lecture Notes in Computer Science*, pages 33–56. Springer-Verlag, New York, NY, 1992.
4. Inc. Apple Computer. Dylan interim reference manual, 1994.
5. P. F. Dietz. Fully persistent arrays. In F. Dehne, J.-R. Sack, and N. Santoro, editors, *Proceedings of the Workshop on Algorithms and Data Structures*, volume 382 of *Lecture Notes in Computer Science*, pages 67–74, Berlin, August 1989. Springer-Verlag.
6. Paul F. Dietz and Rajeev Raman. Persistence, amortization and randomization. In *Proc. 2nd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 78–88, 1991.
7. M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. In *29th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 524–531. IEEE Computer Society Press, 1988.
8. J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. Computer Systems Sci.*, 38(1):86–124, 1989.
9. David Eppstein and S. Muthukrishnan. Internet packet filter manegement and rectangle geometry. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2001.
10. P. Ferragina and S. Muthukrishnan. Efficient dynamic method-lookup for object oriented languages. In *European Symposium on Algorithms*, volume 1136 of *Lecture Notes in Computer Science*, pages 107–120, 1996.
11. P. Ferragina, S. Muthukrishnan, and M. de Berg. Multi-method dispatching: A geometric approach with applications to string matching problems. In *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing*, pages 483–491, New York, May 1–4 1999. ACM Press.
12. R. Fleischer. A simple balanced search tree with O(1) worst-case update time. *International Journal of Foundations of Computer Science*, 7:137–149, 1996.
13. C. Levcopoulos and M. Overmars. A balanced search tree with O(1) worstcase update time. *Acta Informatica*, 26:269–277, 1988.
14. K. Mehlhorn and S. Näher. Bounded ordered dictionaries in $O(\log\log n)$ time and $O(n)$ space. *Information Processing Letters*, 35:183–189, 1990.
15. S. Muthukrishnan and Martin Müller. Time and space efficient method-lookup for object-oriented programs (extended abstract). In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 42–51, Atlanta, Georgia, January 28–30 1996.
16. R. Raman. *Eliminating Amortization: On Data Structures with Guaranteed Response Time.* PhD thesis, University of Rochester, Computer Science Department, October 1992. Technical Report TR439.
17. N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29:669–679, 1986.
18. P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6:80–82, 1978.