# Efficient Execution of Pure Object-Oriented Programs by Follow-up Compilation

Sanjeev Baskiyar, Ph.D.
Assistant Professor
Dept. of Computer Science & Software Engineering
Auburn University
Auburn, AL 36849
baskiyar@eng.auburn.edu

## Abstract

Dynamic type checking and method binding slows pure object-oriented programs such as those written in Smalltalk. Dynamic method lookup is needed to support method overriding and type-method conformance. We have developed a platform independent technique that implements method overriding efficiently for pure object-oriented languages. The technique involves binding non-overridden method calls at compile time. The overridden method calls are also resolved (and bound) statically using simple inferences. Overridden method bindings are corrected dynamically using an Overridden Method Dictionary. Using simulations, we demonstrate that this technique offers 35%-70% reduction in the average method lookup delay over the Berkeley Smalltalk implementation. This technique has very low compile time overhead, memory overhead, does not need specialized hardware and allows shared code pages in concurrent programs. For programs which may be *re-used* it is proper to have a follow-up compilation to generate efficient code after the program development is complete.

Keywords: Dynamic Binding, Object-Oriented, Overriding, Smalltalk, Type Checking.

## 1 Introduction

Object-oriented languages facilitate a framework for developing modular programs by providing classes, inheritance and method overriding. They support incremental program development and an effective interface mechanism for programs developed by more than one programmer. Smalltalk, one of the first pure object-oriented languages, has recently seen a resurgence in interest by many developers because the hardware needed to support the environment is now within the reach of many. The Smalltalk programming environment makes programming simple by supporting dynamic type checking, late message binding and automatic storage management. The environment also enlivens the programmer by supporting incremental compilation. However, these facilities do take a toll–they reduce the speed of execution.

In this paper we show how to enhance the execution speed of Smalltalk programs by a follow-up compilation as suggested in [2]. We address one of the primary speed bottlenecks, run-time type checking and determining the correct methods to execute, which can have a significant impact on the popularity of Smalltalk. We name this technique Compiling using Overridden Methods Dictionary (COMD). This technique can either be used to modify an existing system, or a follow-up compilation may be performed after program development is complete to enhance execution speed. For programs which may be *re-used* it is proper to have a follow-up compilation to generate efficient code after the program development is complete. Although we focus our attention on Smalltalk, the technique is applicable to other pure object-oriented languages, that support dynamic method addition and deletion, as well.

The remainder of this paper is organized as follows. In Section 2 we survey previous approaches to speed object-oriented programs, in Section 3 we outline the COMD technique that involves follow-up compilation using run-time data structures and routines that handle type checking and binding of overridden methods, in Section 4 we outline how the technique can be extended to add/delete methods at run-time, in Section 5 we analyze the COMD technique, present simulation results that demonstrate its performance, and compare it against other techniques and in Section 6 we summarize our effort.

## 2 Background

To support dynamic binding, the system must be able to distinguish data types at run-time[1]. Such a distinction can be facilitated either by having the type/class information in the path of access to the data (i.e. in the descriptor) or by tagging each data item. Smalltalk provides the type information by having every object contain an object pointer to its class description; that pointer could be considered as the data type (tag) of the object. Unlike other fields, the class of an object may be fetched, but not changed. The message "fetchClassOf: Objectpointer" is provided for fetching the class of an object pointer. Also, the pointer for each instance variable contains a pointer to the instance variable's class.

In the Smalltalk implementation "by the book" [12] class descriptors and method dictionaries of all classes are inter-linked in the form of a tree that mirrors the class-superclass relationship. Figure 1 shows such a *Method Dictionary Tree* (MDT). The classes are represented by $A \ldots F$, $B$ is a subclass of $A$ or $B \subset A$. The entries in the method dictionaries are pairs of selectors (method names) and the addresses of the corresponding method bodies. When a message is sent to a receiver, the method dictionary in the receiver's class is searched for a matching selector. If none

[1]Static type distinction is facilitated by explicit type declaration and specification of valid operations on data.

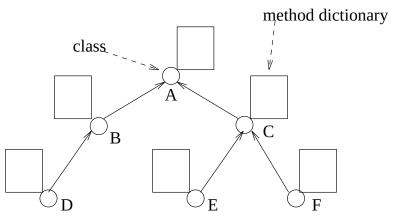

Figure 1: Method Dictionary Tree

is found, the dictionary of the receiver's superclass is searched. The search travels towards the root of the MDT until a matching selector is found. This approach binds *all* calls dynamically following a hierarchical search of method dictionaries to determine the correct method to execute, thereby slowing execution. This search was identified in [16] as extremely time intensive. Also, since portions of the dictionaries are brought into the cache, the cache has to be invalidated whenever any method dictionary is altered. Smalltalk system performance is 5-20% [3] of the performance of optimized *C* programs. Also, the median and maximum overheads due to virtual function table look-ups [8] for $C_{++}$ programs on superscalar processors were measured to be 5% and 29% respectively. These percentages rose to 14% and 47% respectively if all function calls were made virtual. Below we review past attempts to speed the execution of object-oriented programs.

The execution speed of Smalltalk programs was improved in SOAR [22] by placing the target address for any method invocation in the instruction stream once the target address is found; if program control reaches the same point again, the target address does not need to be recomputed [7]. Inline caching speeds execution, at high (95%) hit ratios, however, Driesen et. al. [9] point out that several equally likely receiver types decrease the hit ratio, e.g. SELF spent 25% of its time handling in-line cache misses. Polymorphic in-line caches (PIC) rectify this problem [14] by dynamically saving several look-up results and using a stub to resolve future calls. However, both the above approaches incur run-time delay in determining the target address for the first invocation of any call site. Therefore, such techniques do not help in programs with few repeated calls. Furthermore, such techniques can not be used, when sharing code among concurrently running processes, on many common operating systems (e.g. Unix). These operating systems disallow writes on code pages shared among concurrent processes. Future writes can be allowed by duplicating the code on a page fault due to a first write; however it may involve temporally expensive flushing of the cache to ensure instruction and data cache consistency in a split cache design.

Architectural support for object-oriented languages has been provided in REKURSIV [13], a tightly coupled cluster of processors. It fans separate processors for type checking, index range checking and maintaining a single level store. The main REKURSIV processor supports microcoded instructions. Harland claims that having abstract high level instructions enhances the system, by avoiding packing and moving the same data in two successive instructions.

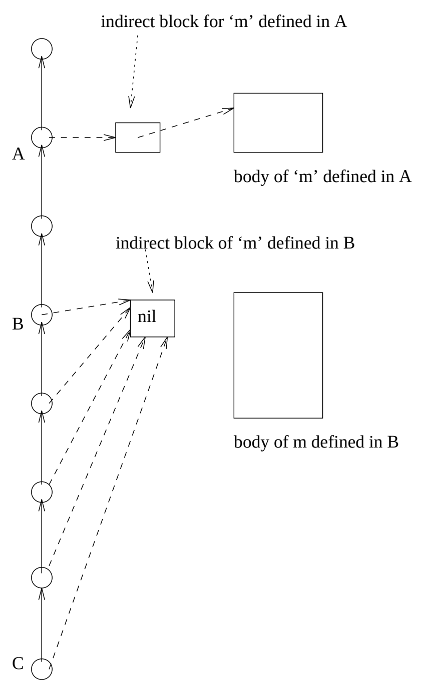
indirect block for 'm' defined in A
A
body of 'm' defined in A
indirect block of 'm' defined in B
B
nil
body of m defined in B
C

Figure 2: Hierarchical search of method dictionaries due to method deletion

REKURSIV uses separate tables to maintain the address of objects in memory, their sizes, types, access ranges[2] and the first word of each object. These tables are accessed in parallel and the corresponding checks are performed. If a check fails, the object is not fetched. This approach needs specialized hardware, furthermore, it does not address the problem of hierarchical search of method dictionaries in the MDT.

In the scheme of inheriting dictionaries [15] the method dictionary for a class contains all inherited method names and references to their method bodies, in addition to those for the proper methods of the class. Therefore, method name resolution involves searching only in the method dictionary of the class of the receiver, alleviating the need for the hierarchical search employed in an implementation "by the book." But maintaining the *direct* addresses of the method bodies in the method dictionaries can not support insertion, deletion or modification of methods. To solve this problem, O'Keefe's scheme stores the addresses of indirect blocks in the method dictionaries; the indirect blocks in turn point to the method bodies. However, this approach suffers the following drawbacks:

- Binding all calls dynamically together with another level of indirection is temporally expensive.

[2]The range checker determines whether indices lie within bounds.

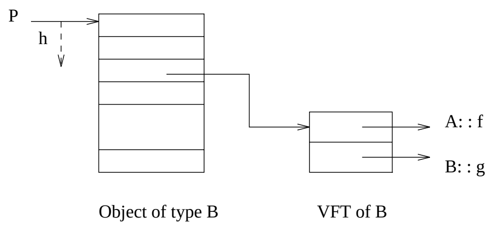

Figure 3: Dynamic Binding in C ++

- The method dictionaries consume more memory than a straightforward implementation [12] by the book because they contain entries for the inherited methods also. Portions of the dictionaries may be brought into the cache reducing the available cache space which in turn may reduce the speed of execution.
- As demonstrated in the following example, deletion of methods may still involve searching the hierarchical method dictionaries, thus *defeating* the proposed solution. In Figure 2 suppose method $m$ first declared in class $A$ ($m \in A$) has been overridden in class $B$, where $B \subset A$. Suppose now that the method $m$ belonging to class $B$ is deleted. Next, consider a message with selector $m$, to an instance of $C$. The search for $m$ now has to travel up the class hierarchy from $C$ to $A$, where $m$ is finally found.

In $C_{++}$ [19] a pointer to an instance of type $t$ may only point to instances whose types are a public subtype of $t$[3]. Calls to virtual functions only are bound dynamically. Associated with each class is a Virtual Function Table (VFT) that has pointers to the bodies of functions declared virtual in that class or any of its parents. The pointers in the VFT appear in an order pre-determined by the compiler. Within the body of any instance of a class, at a compiler determined offset, there exists a pointer to the VFT for that class. The offset is identical for all instances of a class. The implementation of dynamic binding is illustrated in Figure 3 where $p$ is a pointer to an instance of an object $o \in B(\subset A)$. At an offset $h$ in $o$ there is a pointer to the VFT of $B$ which has the virtual functions $f \in A$ and $g \in B$. Run-time binding in $C_{++}$ can be performed in constant time for all instances of all classes. However, this approach suffers the following drawbacks:

- For *each* class, a VFT needs to be maintained which not only holds all virtual functions of that class but also those of its superclasses–consuming cache space.
- All bindings suffer two levels of indirection.
- Methods or classes can not be added at run-time.

[3]It is illegal to cast a pointer to an object of type $t$ to point to an object that is not a subtype of $t$. Also, the type of pointer can only change at run-time if it was statically typed *void ** .

In Java all functions are declared virtual. However, non-overloaded and overloaded function calls are resolved at compile time, only overridden function calls are resolved dynamically in a fashion similar to $C_{++}$. Also, the dynamic binding approach of Eiffel [17] parallels that of $C_{++}$. The COMD technique is relevant to pure object-oriented languages and as such comparisons are more appropriate to such languages.

A survey of dispatch techniques appears in [9]. Selector Table Indexing implements the lookup table by a matrix indexed by class and selector. The size of the table is of the order of $O(c*s)$, where $c$ is the number of classes and $s$ the number of selectors. Because of the large size of the table, none of the systems employ this technique. Techniques such as Selector Coloring, Row Displacement and Compact Selector-Indexed (CT) tables reduce the number of empty entries in the look-up table thus solving the size problem. However, all of the above techniques perform worse than Inline caching and Polymorphic in-line caching on pipelined superscalar processors because of pipeline stalls on unpredictable branches. Notable among the static techniques is CT which uses separate tables for standard and conflict selectors. Standard selectors are those that can be overridden in subclasses whereas conflict selectors are defined in unrelated classes. The conflict selector table is usually sparse unlike that of standard selectors. However the standard and conflict selector tables can be merged for further space savings by trading lookup speed. CT too incurs lookup latency due to subtype test in all method prologues.

Performing bindings at compile time can also reduce the dispatch overheads. The SELF [3, 5] compiler predicts the class of the receiver by a combination of the following techniques: class hierarchy analysis, type feedback from dynamic profiles, message splitting at merge points to retain type information and iterative type analysis of variables in a loop. It also uses customization (code for each possible type of object that a variable can refer to) and defers binding for uncommon calls. The above improve the performance of SELF compiled programs to that of 50% of optimized $C$. Using a compiler called Vortex [5] the effectiveness of above techniques and of others such as interprocedural class analysis and selective recompilation was tested. However, the compile time and code size increase rapidly with program complexity.

## 3 The COMD Technique

The basis of the COMD technique consists of binding both overridden and non-overridden methods statically and correcting at run-time for faulty binding, if any, of overridden methods. The compiler statically determines which method names have been overridden. All calls to non-overridden method names are statically resolved and bound. All calls to overridden method names are also statically bound as follows. If there exists a method matching the selector in the class of the receiver (as inferred by simple static analysis), the call is bound to that method, otherwise to a matching method in the receiver's nearest superclass. At run-time, upon method invocation, a routine called the *Type Checking and Re-direction (TCR)* routine[4] is invoked to

[4]The implementation must prevent *TCR* from being overridden.

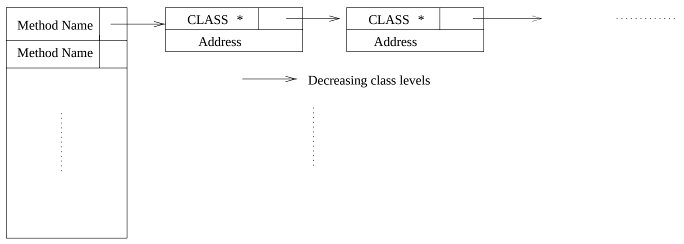

Figure 4: Overridden Method Dictionary

determine whether the static binding was consistent with the class of the receiver; if not *TCR* handles the binding consistently. The following dynamic structures assist *TCR* in achieving the above objective:

- A directed tree called the *ClassTree* to access the class hierarchy information at run-time. Each node of the *ClassTree* is of the type *CLASS* as defined below:

  ```
  struct CLASS {
      int Level;
      CLASS * Parent;
  }
  ```

  The field Level contains the level number of a class. The root class *Object* is assigned the level number *0*. The level of a class is one more than the level of its parent. If multiple inheritance were allowed, the level of a class will be one more than the maximum of the levels of its parents; for simplicity, in this discussion we disallow multiple inheritance.

- A dictionary called the Overridden Method Dictionary (OMD) as shown in Figure 4, that contains the name of each overridden method along with its different classes and corresponding addresses of the method bodies. The OMD is defined as follows:

  ```
  struct MethodNameInfo {
      int MethodName;
      MethodBodyInfo  Ptr;
  }

  struct MethodBodyInfo {
      CLASS * Class;
      int Address;   //Method Body Address
      MethodBodyInfo * Next;
  ```

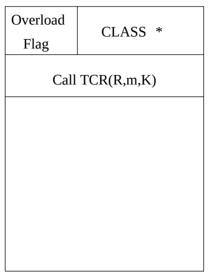

Figure 5: Method Body for Selector $m$.

```
}

MethodNameInfo OMD[N];
// N is the number[5]of overridden methods.
```

The records in the MethodBodyInfo list are organized such that list traversal leads from child to parent classes. The class levels in the *ClassTree* help organize (and search) the MethodBodyInfo list. However, in some instances they are not enough to determine the class-subclass relationships because two classes existing in different branches of the *ClassTree* may have the same level number. For those instances, a traversal of the *ClassTree* may be necessary.

- An overload flag in each method body, as shown in Figure 5, that is set *true* if the corresponding method name has been overridden and *false* otherwise. The pointer to the class to which the method body belongs is also stored in the method body. The first instruction in all method bodies is a call to the *TCR* routine[6].

The procedure *TCR* is defined in Figure 6 and further elaborated below. Consider a message with selector $m$ to receiver $r$. If $m$ is non-overridden, the unique method body for $m$ is statically bound in the call. If $m$ is overridden, the compiler infers the class of $r$ and statically binds the call to a matching method as per Smalltalk rules. Let $K$ represent the class of the statically bound method. $TCR$ corrects bindings for overridden methods at run-time. This gives room for the target inferences to err, thereby keeping the inferencing portion within the compiler simple and fast. At run-time, if $r$ refers to an object, let $R$ refer to its class otherwise, if it refers to the pseudo variable *super*, let $R$ refer to the parent class of the currently executing method.

[5]To handle overriding of methods at run-time by method addition, in an actual implementation, $N$ is empirically set greater than the number of overridden methods found statically; alternatively, one may grow the table dynamically.

[6]To prevent another call to *TCR* the address of a method body in the OMD is actually the address of the instruction following the instruction *Call TCR* in the method body.

```
**inline void** TCR(CLASS * $R$, int $m$, CLASS * $K$)
//Corrects binding for message with selector $m$.
//Inputs: $R$, the class of the receiver, $m$ the selector, $K$ the class to which the method
//has been statically bound.
//Let function $F$ return index $j$ corresponding to an entry for selector $m$ in the OMD.

if $m$ is not overridden
  if SubClass($R, K$)
    return // Binding is correct
  else
    Send message "Does not understand: $m$" to $r$
  endif
else
  $j = F(m)$
  $Address$ = FindMethod($OMD[j].Ptr, R$)
  if ($Address$)
    goto $Address$
  else  //Method not found
    Send message "Does not understand: $m$" to $r$
  endif
endif
end $TCR$
```

Figure 6: Procedure TCR

```
**inline int** FindMethod(MethodBodyInfo * $Ptr$, CLASS * $R$)
// Returns address of the correct method to bind the call
// to a receiver in class $R$ if
// found in the MethodBodyInfo list otherwise, 0
    **while** ($Ptr$)
        **if** ($R \rightarrow Level \geq Ptr \rightarrow Class \rightarrow Level$)
            **if** SubClass($R, Ptr \rightarrow Class$)
                **return** $Ptr \rightarrow Address$;
            **endif**
        **else**
            $Ptr = Ptr \rightarrow Next$;
        **endif**
    **endwhile**
    **return** 0;
**end** FindMethod
```

Figure 7: Procedure FindMethod

Procedure *TCR* first checks whether the compile time binding for the overridden method was correct; if erroneous, it handles the run-time binding. It determines whether the selector $m$ has been overridden by testing the overload flag[7] of the method body to which the call was bound statically. If $m$ has not been overridden and $R \subseteq K$ program control is returned to the method body, otherwise a message "does not understand:$m$" is sent to $r$[8]. In the latter case, $r$ must have been erroneously set to point to an object whose class neither defines nor inherits $m$. If $m$ has been overridden, *TCR* continues with its re-direction part. It finds the list, $L_j$, in OMD that has the addresses of all the method bodies for the selector $m$. It traverses $L_j$ to determine the class whose $m$ will provide the correct binding in accordance with Smalltalk rules. During the traversal, only *few* cases in which the class level in the node is smaller or the same as that of $r$, require a traversal of the *ClassTree* for further verification of the class subclass relationship, for most others, the comparison of the class levels with a much smaller computational overhead, is enough. After determining the correct class, control is transferred to the method body for $m$ in that class. The procedures in Figures 7 and 8 further clarify the traversal in the MethodBodyInfo list. The class-subclass relationship can also be determined without traversing the *ClassTree*–a novel coding technique for determining parent child relationship has been presented in [18].

[7]Procedure *TCR* finds the address of the method body, and hence its class and overload flag, by an offset to its own return address.

[8]In Smalltalk, there is a method in the class *Object* corresponding to the selector "does not understand:" that reports the error to the programmer and suspends the process.

```
inline int SubClass(CLASS * $A$, CLASS * $B$)
// Returns 1 if $A \subseteq B$ otherwise, 0

    while ($A \rightarrow Parent \rightarrow Level$) //not root
        if ($A \rightarrow Parent == B$)
            return 1;
        else $A = A \rightarrow Parent$;
    endwhile
    return 0;
end SubClass
```

Figure 8: Procedure SubClass

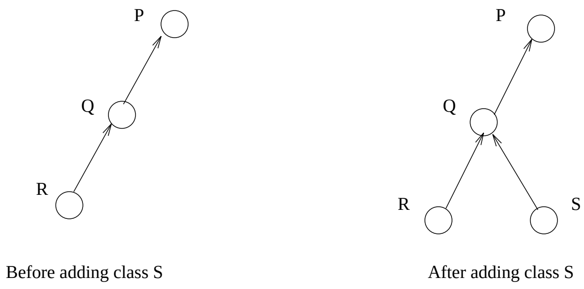

Figure 9: Adding a Class

## 4 Adding Methods Dynamically

Smalltalk environments facilitate incremental method addition. Below we outline how it can be facilitated when the COMD technique is employed. The COMD technique supports method addition/deletion to any class and creation[9] of any sub-class at run-time as shown in Figure 9, where $P \supset Q \supset R$ and $Q \supset S$.

If a method $m$ is added to a class, a run-time method addition routine is invoked that determines whether the addition causes overriding. First, the OMD is searched for $m$. If $m$ is found, the overload flag in the method body being added is set and a corresponding entry is made in the MethodBodyInfo list for $m$; the point of insertion in the list is determined by the level of the class in which $m$ is defined. If an entry for $m$ is not found in the OMD, the MDT is searched for $m$. If found, the overload flags of the method $m$ being added and the one found are set and a corresponding entry made in the OMD. Otherwise, $m$ is added to the appropriate dictionary of the MDT. The procedure for method deletion is symmetric; one may approach the

[9]Dynamic insertion and deletion of classes are best supported by recompilation.

problem using a method counter or using a “once overridden always overridden” approach.

## 5 Performance

In this section we analyze the space and time overheads in the COMD technique against those of the Berkeley Smalltalk implementation [20] and present simulation results to compare the average method lookup delays. Lastly, we compare our technique with those of other researchers. First we introduce notation as a prelude to analysis. Let

| | | |
|---|---|---|
| $S$ | $=$ | Number of non-overridden selectors. |
| $S'$ | $=$ | Number of unique overridden selectors. |
| $k_m$ | $=$ | Number of times selector $m$ has been overridden. |
| $k$ | $=$ | Average number of times any selector has been overridden, or the degree of polymorphism. |
| $D$ | $=$ | Depth of the MDT/*ClassTree*. |
| $d$ | $=$ | Average distance [10] that a method lookup has to traverse in the MDT. |
| $f$ | $=$ | Fraction of all calls that are only to overridden methods. |
| $v$ | $=$ | Probability that, for any node in the MethodBodyInfo list, *FindMethod* calls *SubClass* of the class subclass relationship, after successful class level comparison. |

In the Berkeley Smalltalk implementation space overhead occurs in maintaining the MDT. The non-overridden selectors occur once in the MDT whereas the overridden selectors occur more than once giving a space overhead of $O(S + \sum_m k_m)$, where the summation is over all overridden methods. In the COMD implementation space overhead occurs in maintaining the OMD giving rise to complexity of $O(N + \sum_m k_m)$. Since $N \simeq S'$ and in general $S' \ll S$ (on the average $S' = 0.0007 \cdot S$, [9]), the COMD implementation is spatially more efficient than the Berkeley Smalltalk implementation.

Now we discuss the average case time complexities. In Berkeley Smalltalk binding a call to a method involves search by hashing the method dictionaries at each level of the of the MDT. If there were equal number of selectors, $n$, at each node, the average complexity would be *O(nd)*. In the COMD implementation, for any selector the OMD can be indexed in *O(1)* time (the function $F$ is straightforward since OMD can be pre-sorted and its size, $N$, is small to guarantee such an indexing; also table indices may be computed at compile time). If on the average, $p$ nodes of the MethodBodyInfo list are traversed, $pv$ of such traversals call *SubClass* which gives an average complexity of $O(pvd)$. On the average $p < k$, and since $0 < v \leq 1$, $pv < k$. COMD is faster than Berkeley Smalltalk since on the average $k < n$. If the tree coding technique of

[10]The tables in [20, 10] give the percentage of method lookups that traverse depths $0, 1, 2, \ldots, 10$ of the MDT. Over 70% of lookups are resolved within depth 1.

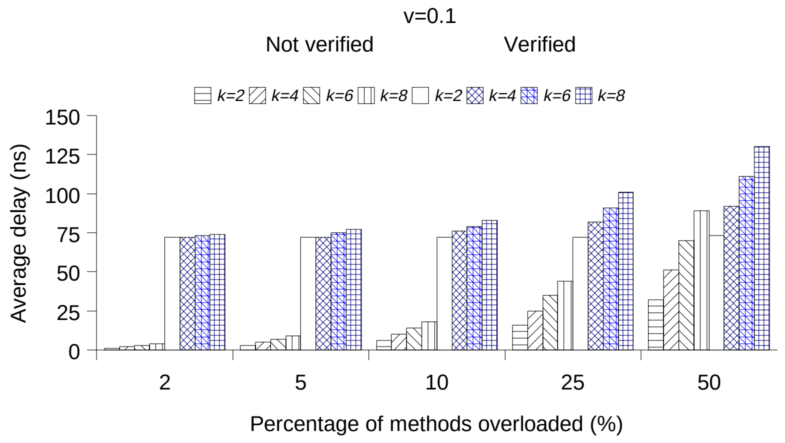

Figure 10: Average Method Lookup Delays for $v = 0.1$

[18] is employed, class-subclass determinations can be performed in *O(1)* time and, the average complexity of the COMD technique further reduces to *O(pv)*.

Next, we discuss the typical average time spent in method lookups. Berkeley Smalltalk uses a *software* method cache to speed method lookups. The hit rate for the *software* method cache of size *1KB* in an interactive session of editing browsing and short arithmetic computations on VAX 11/780 is 94.8% with the average method lookup time [4] being $19\mu$s. Using the SPECbase92/SPECmark suite [11] this time delay translates to 123 ns on a Pentium 133 MHz with 1 MB of cache. Since it is difficult to compare lookup delay for different techniques across processors, languages, applications and run-time systems, we compare against the best possible benchmark, the Berkeley Smalltalk. Simulations were performed on the Pentium 133 MHz to estimate the average delay for method binding when the COMD technique is used and the results were graphed. We hand-coded the dispatch instruction sequences for optimal performance. Figures 10, 11, 12 and 13 show the average delays for different values of $f$ and $k$ for $v = 0.1, 0.2, 0.5$ and 1.0 respectively. We observe that for $(v = 0.1, k \leq 4, f \leq 25\%), (v = 0.2, k \leq 4, f \leq 25\%), (v = 0.5, 1.0, k \leq 4, f \leq 10\%)$, the COMD technique gives 35% reduction in the average delay for method lookup over Berkeley Smalltalk. Note that, in the ParcPlace VisualWorks Smalltalk system, $k = 3.49$ [9]. The function *FindMethod* calls *SubClass* only if the level of the class of the receiver object is more or same as the class of the method, and since the overridden selectors are evenly distributed across different levels, on the average, $v \leq 0.5$.

Furthermore, for non-overridden methods, if the compiler can safely infer that the receiver's class can not fall outside the set of classes containing the class of the method and its descendants, the subclass verification check in lines 2,4 & 5 of TCR is not needed.[11] Doing so, results in even shorter delays for method lookups as shown in the bar graphs with label "Not verified." The

[11]For some cases, which arise from the set of cases in which the object's value is conditionally assigned, where the compiler may be unable to safely predict the class of the receiver a compile time error message may be generated.

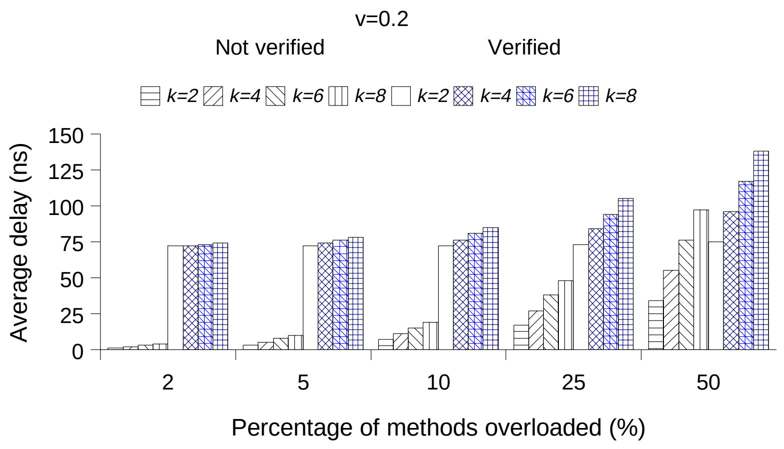

Figure 11: Average Method Lookup Delays for $v = 0.2$

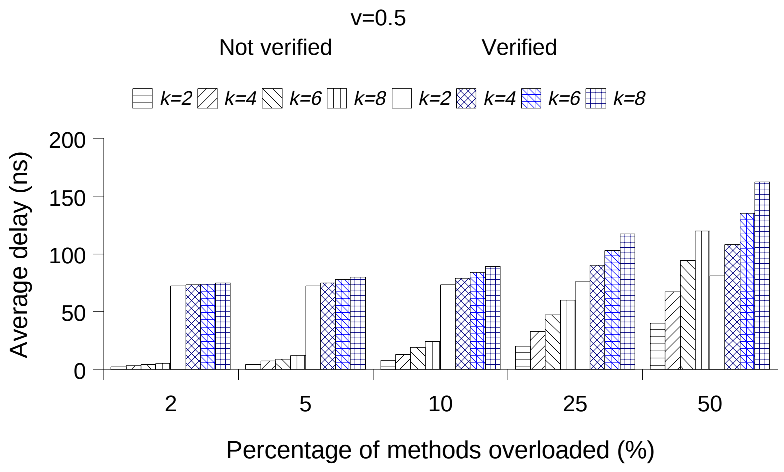

Figure 12: Average Method Lookup Delays for $v = 0.5$

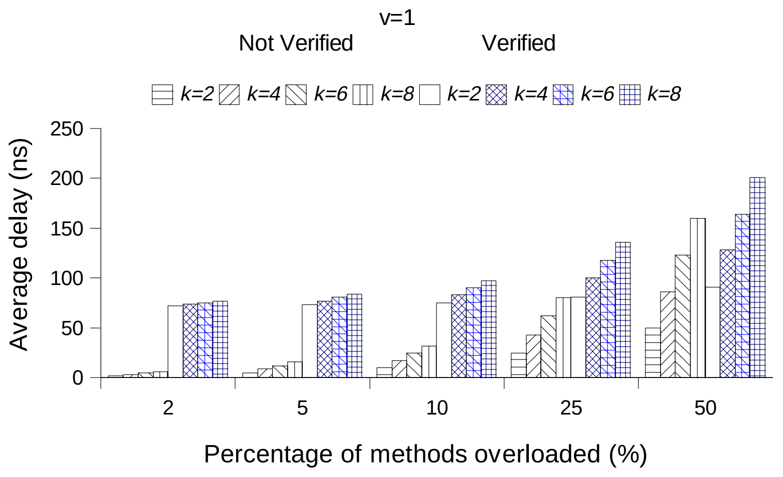

Figure 13: Average Method Lookup Delays for $v = 1.0$

graphs show an improvement of over 70-75% over Berkeley Smalltalk in method lookup delays for $(v = 0.1, 0.2, 0.5, 1.0; f \leq 25\%, k \leq 4)$.

Below we qualitatively compare the COMD technique against those of previous researchers.

**Temporal** In the COMD technique, bindings for only overridden method names need to be resolved at run-time. Resolution for other method names are performed prior to execution whereas in [12] and [15] all bindings are performed at run-time slowing execution. Furthermore, compile time optimizations techniques [1] may be employed to further speed execution.

**Spatial** The OMD is smaller than the MDT of [12] and therefore consumes less space and can be easily maintained in the cache, since usually only a fraction of all method names are overridden. The scheme in [15] entails even more space overhead than in [12], because in the former the method dictionary for *each* class holds entries not only for its proper methods but also for all inherited ones. In $C_{++}$ too the VFT for each class holds entries for all inherited virtual functions.

**Hardware** The COMD technique does not require specialized hardware unlike [13] nor does it require the processor to support dynamically modifiable code unlike [22].

Table 1 qualitatively compares the prominent techniques namely: the Static techniques, Vortex, PIC, Berkeley Smalltalk and COMD. Clearly the COMD technique offers all the features: low method lookup latency, low space overhead, low compile time, and does not require the underlying system to support dynamically modifiable code.

| Technique | Aspect | | | |
|---|---|---|---|---|
| | Method lookup latency | Space overhead | Compilation time | Needs support for dynamic code modification? |
| Static | Low/high if space overhead high/low | High/low if latency low/high | Low | No |
| PIC | Low | Low | Low | Yes |
| Vortex | Low | Low | High | No |
| Berkeley Smalltalk | High | High | Low | No |
| COMD | Low | Low | Low | No |

Table 1: Comparison of the latency reduction techniques

## 6 Conclusion

The COMD technique has dual advantages: on one hand it offers a facility similar to static type checking which can be used to prove the correctness of programs and allocate space for variables, on the other hand it facilitates dynamic binding. It offers the *much needed* speed-up for Smalltalk programs, independent of the platform of execution. This technique has very low compile time overhead, memory overhead, and allows shared code pages in concurrent programs.

## 7 Acknowledgements

The author thanks Prof. Richard Y. Kain, Dr. Gary Elsesser and Dr. Joseph Falcone for making valuable contributions. Several students over years helped in this work in various ways, in particular, the author thanks research assistant N. Meghanathan for assistance in drawing the graphs. A portion of this work was supported by an Auburn University internal grant.

## References

[1] A.V. Aho, J. Ullman and R. Sethi, *Compilers, Principles, Techniques, and Tools*, Addison-Wesley Publishing Co., 1988.

[2] S. Baskiyar, Architectural and Scheduler Support for Object-Oriented Programs, *PhD Thesis*, University of Minnesota, Minneapolis, 1993.

[3] C. Chambers and D. Ungar, "Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs," Proceedings the ACM-SIGPLAN, White Plains, New York, June 1990.

[4] T.J. Conroy and E. Pelegri-Llopart, "An Assessment of Method-Lookup Caches for Smalltalk-80 Implementations," in G. Kranser, *Smalltalk-80 Bits of History, Words of Advice*, Addison-Wesley Publishing Company, 1983.

[5] J. Dean et. al., "Vortex: An Optimzing Compiler for Object-Oriented Languages," *Proceedings of the OOPSLA*, CA, 1996.

[6] J. Dean, D. Grove, C. Chambers, "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis," *Proceedings of 1995 ECOOP*, Aarhus, Denmark, 1995.

[7] L. Deutsch and A. Schiffman, "Efficient Implementation of the Smalltalk-80 System," *Proceedings of the* 11*th* *Annual ACM Symposium on the Principles of Programming Languages*, Utah, 1984.

[8] K. Driesen and U. Holzle, "The Direct Cost of Virtual Function Calls in C++," *Proceedings of OOPSLA*, CA, 1996.

[9] K. Driesen, U. Holzle, and J. Vitek, "Message Dispatch on Pipelined Processors," in *Proceedings of ECOOP, 1995.*

[10] J. R. Falcone, "The Analysis of the Smalltalk 80 System at Hewlett-Packard," Kranser, G., *Smalltalk-80 Bits of History, Words of Advice*, Addison-Wesley Publishing Company, 1983.

[11] J. Falcone, Private Communication. Also in, http://hpline.epfl/ch/bench/SPEC.html and http://www.specbench.org/spec

[12] A. Goldberg and D. Robson, *Smalltalk-80, The Language and its Implementation*, Addison-Wesley Publishing Company, 1984.

[13] D. M. Harland, *REKURSIV: Object-Oriented Computer Architecture*, Ellis Horwood Limited, 1988.

[14] U. Holzle, C. Chambers and D. Ungar, "Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches," *Proceedings of the ECOOP*, 1991.

[15] R. A. O'Keefe, "Finding Smalltalk Methods," in *SIGPLAN Notices*, V20 #6, June 1985.

[16] G. Krasner, *Smalltalk-80 Bits of History, Words of Advice*, Addison-Wesley Publishing Company, 1983.

[17] B. Meyer, *Advances in Object-Oriented Software Engineering*, Prentice Hall Inc., 1992.

[18] N. Meghnathan and S. Baskiyar, "Binary Coding for Fast Determination of Ancestor Descendant Relationships in Trees," *TR CSSE 2001-09*, Dept. of CSSE, Auburn University, 2001.

[19] B. Stroustrup, *The Design and Evolution of* C++ , Addison-Wesley Publishing Company, 1994.

[20] D. Ungar and D. Patterson, "Berkeley Smalltalk: Who Knows Where the Time Goes?" in Kranser, G., *Smalltalk-80 Bits of History, Words of Advice*, Addison-Wesley Publishing Company, 1983.

[21] D. Ungar et. al., "Architecture of SOAR: Smalltalk on a RISC," in *SIGARCH Newsletter 12*, No. 3, June 1984.

[22] D. Ungar and D. Patterson, "What Price Smalltalk," in *Computer*, 1987.

[23] J. Vitek and R. N. Horspool, "Taming Message Passing: Efficient Method Lookup for Dynamically Typed Languages," *Proceedings of the* 8th *ECOOP*, Bologna, Italy,1994.

## Author Biography

Sanjeev Baskiyar received the B.Sc. degree in Physics with honors and distinction in Mathematics from St. Xavier's College, India, the B.E. degree in Electronics and Communication from the Indian Institute of Science, Bangalore, the M.S.E.E. and Ph.D. degrees from the University of Minnesota, Minneapolis. Currently, he is Assistant Professor in the Dept. of Computer Science and Software Engineering at Auburn University, Auburn, Alabama, USA. His experience includes working as an Assistant Professor at Western Michigan University, Kalamazoo, Michigan, as a Senior Software Engineer in the Unisys Corporation, Minneapolis, Minnesota, and as a Computer Engineer in TELCO, India. His publications are in the areas of Task Scheduling in Multiprocessors, Computer Systems Architecture and Real-time and Embedded Computing.