RC 20330 (01/11/96)
Computer Science

# IBM Research Report

## Evaluating Virtual Dispatch Mechanisms for C++

Harini Srinivasan, Peter F. Sweeney

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York



IBM Research Division
Almaden · T.J. Watson · Tokyo · Zurich

# Evaluating Virtual Dispatch Mechanisms for C++

Harini Srinivasan and Peter F. Sweeney
IBM Thomas J. Watson Research Center
harini@watson.ibm.com pfs@watson.ibm.com

## 1 Introduction

Polymorphism, supported in C++ via *virtual functions*, is the ability to call a variety of functions with the same signature from the same call site [1]. This feature improves the flexibility of C++ programs, and helps programmers write extensible software. Nevertheless, these benefits come with a performance price.

This paper explores the space and time overhead of virtual dispatch mechanisms that implement polymorphism in C++. How polymorphism is implemented depends on the object model; that is, the specific structures and mechanisms used to support the object-oriented features of a language. In particular, this study holds the object layout fixed, and varies the VFT layout to determine the time and space overhead of the virtual dispatch mechanisms for four different VFT layouts. The object model that is used is generally known [10] and is used in a production compiler, and the four VFT layouts that are evaluated are either generally known [10] or else used in a production compiler. This paper is the first study, known to the authors, that quantitatively compares VFT layouts.

This paper presents a framework, to evaluate virtual dispatch mechanisms, that analytically models time and space overhead. Space overhead is modeled by counting the number of words of storage and the number of instructions that are required to implement different aspects of virtual dispatch mechanisms. Time overhead is modeled by counting the number of instructions that would be executed for an execution of a benchmark. The time overhead is an upper bound for a simple RISC-like target architecture that weighs the cycle time of all instructions uniformly as one. Alternative instruction sequences and instruction cycle times can be collated with our results to draw conclusions about alternative architectures.

Our results, for a set of three C++ benchmarks, demonstrate that VFT layout has a significant impact on virtual dispatch overhead. Alternative VFT layouts varied by as much as two times for space overhead and by as much as 67% for time overhead on the benchmarks that we used. Our results also demonstrate that virtual dispatch performance may be impacted by the source code's use of inheritance (single, multiple, and/or virtual), and where virtual functions are defined in the class hierarchy.

### 1.1 Related Work

Recent research has focused on resolving virtual function calls to direct calls either with analysis that detects false polymorphism (a virtual function call site that can only call one target routine) [11, 5, 6, 16, 17, 4]; or, in the case of true polymorphism (a virtual function call site that can call multiple targets), has focused on specializing the call site by performing explicit run-time type checks [2, 5]. In contrast, our research is interested in evaluating virtual dispatch overhead for different object models when polymorphism is required and a polymorphic call is retained.

An interesting extension of this work would be to evaluate virtual dispatch overhead after false polymorphism is detected and removed.

Past research focused on finding an efficient object model and virtual dispatch mechanism for multiple inheritance [20, 19, 7, 10]. The virtual dispatch mechanisms that we study benefit from this previous research. Our work differs in that we explore in greater detail the impact of VFT layout and class inheritance on virtual dispatch overhead.

Driesen *et al* [9] have done a broad study of various dispatch techniques for implementing virtual function calls. They analytically model the time overhead (in cycles) and space overhead of each of the dispatch techniques for three different processor implementations. In a later paper [8], Driesen and Hölzle study the *Row Displacement* technique in more detail. This paper studies, in more detail, the *Virtual Function Table* technique that is suggested in [20, 10]. We believe that our study is warranted since this approach is used in current C++ compilers. Our paper analytically models the space and time overheads of four different Virtual Function Table layout alternatives. Although we measure the number of instructions while measuring the time costs, we believe that the evaluation framework presented in this paper can be extended, in a straightforward manner, to compute the cycle times as well.

Recently, Myers [14] has proposed a new object model, that is based on earlier work by Pugh et. al [18], that reduces VPTR sharing in subobjects at the expense of requiring indirection to access a subobject's data members. We view this work as an alternative object and VFT layout. This paper does not address the effect of different object layouts on virtual dispatch overhead. In future work, we plan to also model and measure different object layouts.

### 1.2 Organization of the Paper

The rest of the paper is organized as follows: Section 2 discusses background material that is needed for discussing virtual dispatch mechanisms. Section 3 presents the four VFT layouts and their virtual dispatch mechanisms that this paper studies. Section 4 describes a framework to evaluate the time and space tradeoffs and our experimental methodology including the software tools that we used in the empirical study. Section 5 tabulates and discusses empirical results on a set of C++ benchmark programs. Section 6 discusses future directions of research in this area and Section 7 presents conclusions.

## 2 Background

A virtual function is a special member function invoked through a public base class reference or pointer [13]. Such a function is bound dynamically at run-time. This run-time mechanism that resolves the virtual function is transparent to the user and is called *virtual dispatch*. Virtual dispatch is efficiently implemented as an indirect function call through a table of function addresses, called a virtual function table [10]. The virtual function table is usually implemented as an array of function addresses. We refer to this table as the VFT. Conceptually, the compiler generates one or more VFTs for any class that defines or inherits a virtual function [1]. If a class has at least one VFT, then any object of that class has a pointer to one of the class's VFTs, and this pointer provides the mechanism for accessing the VFT. We refer to this pointer as the VPTR.

What are the steps that are required for virtual dispatch? Consider the *static* (or declared) type, $S$, and the *dynamic* (or run-time) type, $D$, of the object pointed to by $p$ in the virtual function

[1] A class may require a different VFT for each of its derived classes. The algorithm for computing VFTs is outside the scope of this paper.

call, $p \rightarrow f()$. $D$ may be $S$ or a derived class of $S$, and the type of the *fully derived* object is $D$. The actual function invoked at this call site is the function $f$ that is inherited or overridden by D. Suppose this function is $E :: f()$; that is, $f$ is defined in class $E$. The following steps may be necessary during virtual dispatch to $E :: f()$ at run-time:

1. Adjust the THIS pointer to the subobject, $X$, in the $D$ object whose VFT contains the VFT entry that has the address of $E :: f()$. Such a THIS adjustment is called an *early cast.*

2. Access the VFT entry.

3. If the $E$ and $X$ subobjects are at different locations within the $D$ object, adjust the THIS pointer to point to the $E$ subobject. Such a THIS adjustment is called a *late cast.* A late cast may be required independently of whether an early cast is required.

4. Branch to the address of $E :: f()$ that is available in the VFT entry.

Steps 2 and 4 are common to all the virtual dispatch mechanisms that this paper explores and are collectively called the *mandatory dispatch code.* The early cast (Step 1) and late cast (Step 3) may vary between virtual dispatch mechanisms. THIS adjustments (early and/or late casts) may be required when multiple and/or virtual inheritance occurs in a program. A *virtual dispatch mechanism* is the manner in which virtual dispatch is implemented for a particular object model. An *object model* is defined in terms of how the following three questions are answered: How are objects laid out in memory; How are VFTs laid out in memory; and What code sequences are required to implement object-oriented language features for a particular object and VFT layout? The object-oriented language feature that is the subject of this paper is polymorphism. Virtual dispatch mechanisms, that implement polymorphism, are affected not only by the object model, but also by class inheritance properties. The rest of this section discusses the rudiments of class inheritance (Section 2.1) and its affect on sharing VPTRs between subobjects (Section 2.2) for a particular object layout. The discussion of VPTR sharing's role in virtual dispatch is deferred until Section 3.4, after the VFT layout alternatives are introduced.

## 2.1 Inheritance

In C++, a class may inherit another class. In C++ parlance, the inherited class is known as the *base class* and the inheriting class is known as the *derived class.* Inheritance is a powerful language abstraction that provides a mechanism for reuse by inheriting method implementations from base classes and a mechanism for extensibility by specializing an inherited class.

There are different types of inheritance in C++, and we are interested in all of them for our investigation of virtual dispatch mechanisms. In particular, a class may either inherit only one other class (*single inheritance*) or inherit multiple classes (*multiple inheritance*), and the inheritance may be specified as virtual (*virtual inheritance*).

Figure 1 illustrates the three types of inheritance. In Figure 1(a), single inheritance is illustrated as a graph of two nodes $B$ and $D$ with an arrow going from $B$ to $D$. This inheritance graph is interpreted as class $D$ inherits class $B$, $B$ is a *direct base class* of $D$ and $D$ is a derived class of $B$. In Figure 1(b), multiple inheritance and virtual inheritance are illustrated as a diamond shaped graph. $D$ is an example of a class that has multiple inheritance as it has more than one direct base class: $C1$ and $C2$. The edges that are labeled with V, from $V$ to $C1$ and $V$ to $C2$, are examples of virtual inheritance. Classes $D$, $C1$ and $C2$ virtually inherit class $V$ and $V$ is a *virtual base* of $C1$, $C2$, and $D$, and a *direct virtual base* of $C1$ and $C2$.

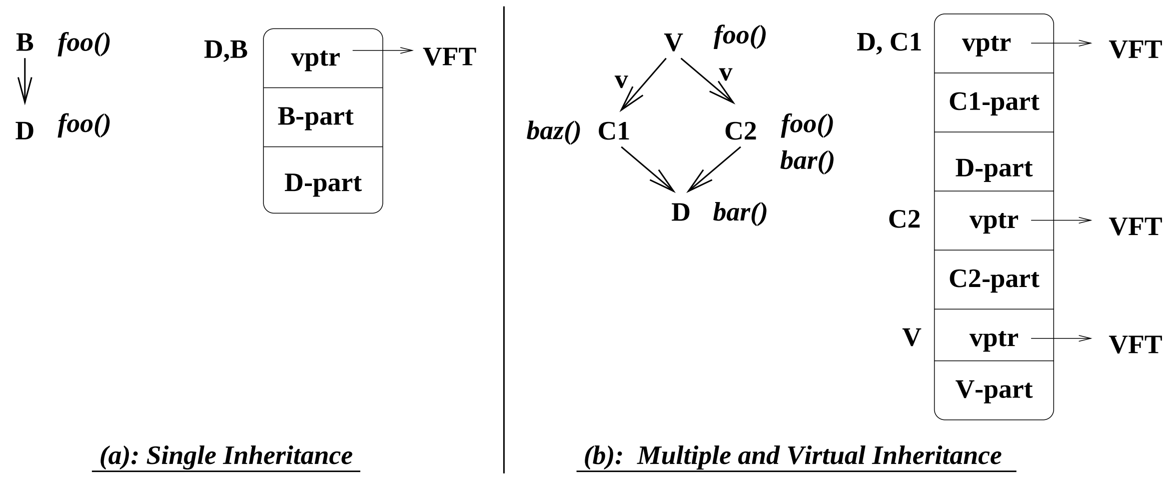


**Figure 1**: Examples of inheritance and of object layout for the most derived class $C$. Edges with V denote virtual inheritance. Virtual functions label the classes that define them.

## 2.2 Object Layout

This section presents how inheritance affects an object's layout. In particular, class inheritance in conjunction with an object layout algorithm determine what VPTRs are shared between subobjects. (Section 3.4 discusses how VPTR sharing helps determine whether or not an early or a late cast is required.) For all the VFT layout alternatives that we explore, we assume that the same object layout is used, namely the object layout presented in *The Annotated C++ Reference Manual* [10] (or ARM for short). The rest of this section talks about this object layout.

ARM describes an object layout scheme where an object of class type $D$ is comprised of subobjects, one for $D$ and one for each of its bases. In this paper, the figures that illustrate object layout hide most of the detailed content of subobjects. A subobject of a class, say $C$, will be represented by a VPTR and a `C-part` that consists of, among other things, $C$'s data members.

Figure 1 shows the object layouts, as proposed in the ARM, for the most derived class in the class hierarchies that are also shown (and have been discussed above in Section 2.1). Figure 1(a) illustrates the object layout for $D$ in the presence of single inheritance. Both $B$ and $D$'s subobject share a VPTR. In general, if there is only single inheritance, all subobjects of an object can share a single VPTR.

Figure 1(b) illustrates the object layout in the presence of multiple inheritance and virtual inheritance. The $D$ object has four subobjects, one corresponding to each class in the class hierarchy. Note that although there are multiple paths from the virtual base $V$ to the derived class $D$, the semantics of virtual inheritance requires that a $D$ object have only one copy of the virtual base subobject $V$ [10]. In the object layout for $D$ there are three VPTRs: one for the virtual base $V$, one for $C2$ and one that is shared by $D$ and $C1$ [2]. In general, a derived class can share its VPTR with at most one of its base classes, and it can not share its VPTR with a virtual base [10].

In addition, if a class has a virtual base, then any subobject of that class will have a *virtual base pointer* to that virtual base, i.e. a pointer to the unique virtual base subobject in the fully derived object [10]. We refer to this pointer as VBPTR. VBPTRs are essential for implementing virtual bases; that is, they provide the mechanism by which a virtual base subobject and its data members may

[2] $D$ could have shared its VPTR with $C2$ instead of $C1$. The algorithm for choosing which direct base class to share a VPTR is outside the scope of this paper.

be accessed. The object layout in Figure 1 does not explicitly show the VBPTRs. We assume the VBPTRs for the $D$, $C1$ and $C2$ subobjects are in `D-part`, `C1-part` and `C2-part` respectively.

# 3 Virtual Function Table Layout

This section presents VFT layout alternatives and discusses how the VFT layout affects a virtual dispatch mechanism. A VFT layout can be described in terms of its shape; that is, its *height*, the number of entries, and its *width*, the contents of each entry. Sections 3.1 and 3.2 present two alternatives each for the height and width of a VFT. Section 3.3 presents the four different virtual dispatch mechanisms that result from combining these alternatives. Section 3.4 discusses how VPTR sharing in an object's layout and where virtual function definitions occur in a class hierarchy affect the early and late casts that are required by a virtual dispatch mechanism.

Before discussing VFT layouts in detail, an explanatory remark is necessary. All code sequences in this section are presented in a psuedo assembly language for a generic RISC architecture. A code sequence's intent is to provide, without getting bogged down in a specific target architecture, a flavor of the assembly code that would be required to implement a particular event. For example, in the code sequences that follow, we do not distinguish between data that resides in memory and data that resides in a register. In the psuedo assembler language, an operand enclosed in parenthesis and followed by a displacement specifies the data whose location is computed from the sum of the contents of the operand and the displacement. For example (VPTR)X is the data located at displacement X from the beginning of the VFT pointed to by the VPTR.

## 3.1 VFT Height

The height of a VFT is determined by the number of entries that the VFT contains. This paper explores two alternatives for determining what VFT entries are contained in a VFT: the VISIBLE and the INTRODUCED alternatives.

VISIBLE: ARM proposes that the entries in a VFT for a class consist of all the virtual functions that are *visible* in that class; that is, all the virtual functions that are either declared at that class or inheritted by that class. When VPTRs are shared by subobjects of different classes, the entries in the resulting VFT are the *union* of the virtual functions that are visible in these classes.

INTRODUCED: In contrast, a VFT for a class may have entries only for those virtual functions that are *introduced* by that class; that is, all the virtual functions that are declared by that class but are not declared by any base of that class. When VPTRs are shared by subobjects of different classes, the entries in the resulting VFT are the *union* of all the virtual functions that are introduced by these classes. A virtual function is introduced by only one of these classes. This approach is adopted for "native" C++ classes in the current IBM XL C++ compiler that runs on the AIX operating system [3].

In both the VISIBLE and INTRODUCED alternatives, when virtual functions are overridden, the virtual function definition that is in a VFT entry is the definition that is in the most derived class of a fully derived object. In the VISIBLE alternative, if multiple or virtual inheritance occurs, the number of VFT entries that are duplicated in multiple VFTs of an object may increase significantly as the depth of inheritance increases. In contrast, VFT entries for the INTRODUCED alternative

[3] In the case of contravariant functions, this compiler creates extra VFT entries.

are not duplicated. The total number of VFT entries for the INTRODUCED is equal to the total number of virtual functions in the C++ program. In general, the total number of VFT entries for the INTRODUCED alternative will be no more than the total number of VFT entries for the VISIBLE alternative, and is expected to be less if multiple or virtual inheritance occurs.

The virtual dispatch mechanism that is needed for the INTRODUCED alternative may require additional space and time overhead to find the VFT entry in the introducing base class's subobject; that is, an early cast may be required. An early cast is always a cast to a base class's subobject. The space overhead is the extra instructions at each virtual function call site that requires an early cast, and the time overhead is the execution of these instructions. In contrast, the VISIBLE alternative never requires an early cast.

For architectures, such as the PowerPC [12], that have a compound load and add instruction, an early cast may not require additional space or time overhead. In particular, if an early cast does not require the use of a VBPTR to access a base class (**NVCast**), the cast can be combined with loading the VPTR. On the other hand, if an early cast requires the use of a VBPTR (**VCast**), the code sequence for an early cast is:

```
load  vbptr, (this)X       // load vbptr from this pointer
add   vbptr, this          // adjust this pointer with vbptr
```

The code sequence first loads the VBPTR from the THIS pointer, and then adds the VBPTR offset to the THIS pointer [4].

### 3.2 VFT Width

The width of a VFT is determined by the number of items in each VFT entry. This paper explores two alternatives for the contents of a VFT entry: the *2-column* and *thunk* alternatives.

2-Column: Each VFT entry contains two items: the address of the function to be invoked at run-time, VF_addr, and the THIS adjustment value, delta, that is used to perform a late cast. During virtual dispatch, the VFT entry is load from the VPTR, and the late cast is performed by loading delta and then adding it to the THIS pointer. The following code sequence is used at run-time to invoke the proper virtual function by loading the VF_addr and delta from a VFT pointed to by a VPTR.

```
load  vptr, this           // load vptr from this pointer
load  VF_addr, (vptr)X     // load VF_addr from VFT entry
load  delta, (vptr)Y       // late cast: load delta from VFT entry
add   delta, this          // late cast: adjust this pointer by delta
b     VF_addr              // branch to VF_addr
```

Thunk: Each VFT entry contains exactly one item: either the address of the function when no late cast is required (i.e. delta is zero), or the address of a snippet of code, called *thunk* (or trampoline) code, when a late cast is required (i.e. delta is non-zero). The thunk code simply performs the late cast and then transfers control to the function address. When no late cast is required, the code sequence during virtual dispatch is:

[4] Some object models implement VBPTRs as pointers, not offsets, and the second instruction in the code sequence would be a load to adjust the THIS pointer. Without loss of generality, we assume, for the sake of this study, that VBPTR really are offsets.

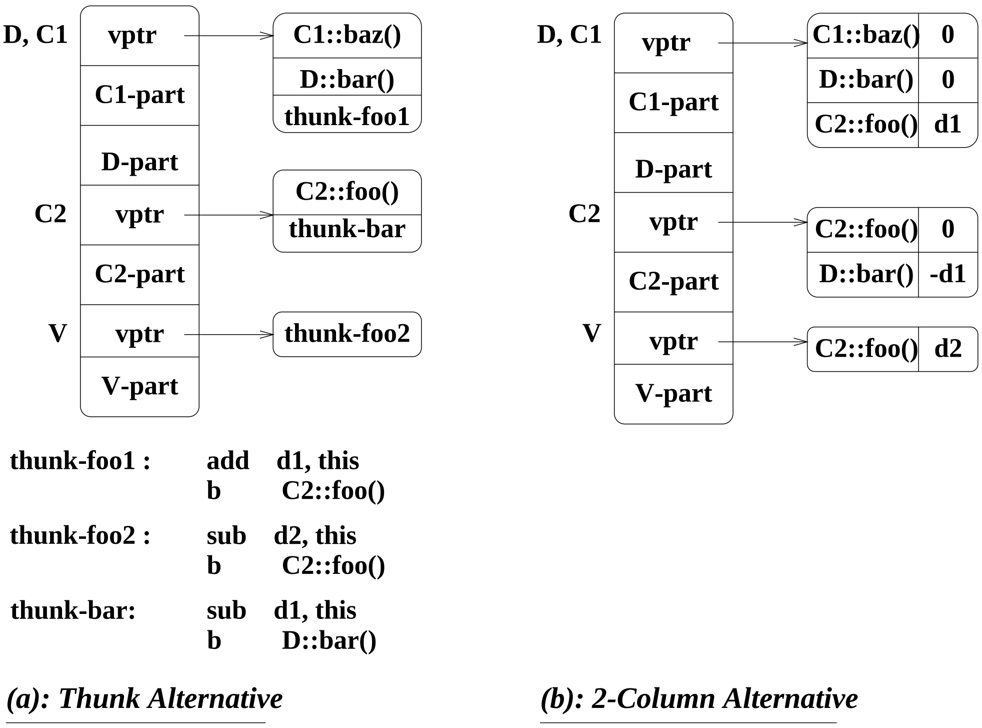


**Figure 2**: Object and VFT Layout using the VISIBLE alternative for the $D$ object that is illustrated in the class inheritance graph in Figure 1(b).

```
load  vptr, this           // load vptr from this pointer
load  VF_addr, (vptr)X     // load VF_addr from VFT entry
b     VF_addr              // branch to VF_addr
```

When a late cast is required, the code sequence during virtual dispatch is:

```
load  vptr, this           // load vptr from this pointer
load  thunk_addr, (vptr)X  // load thunk address from VFT entry
b     thunk_addr           // branch to thunk

add   delta, this          // thunk: add delta to this pointer
b     VF_addr              // thunk: branch to VF_addr
```

Both the 2-COLUMN and the THUNK alternatives are widely known and have been suggested in ARM [10]. All three virtual dispatch code sequences that are presented above require the same three mandatory dispatch code instructions that correspond to the code sequence for THUNK code when no late cast is performed. The other two virtual dispatch code sequences include, in addition to mandatory dispatch code, a late cast. Although both alternatives affect code size, the THUNK alternative allows thunk code to be shared between virtual function call sites when a late cast is required and the late cast values are the same, while the 2-COLUMN alternative requires late cast code at every call site.

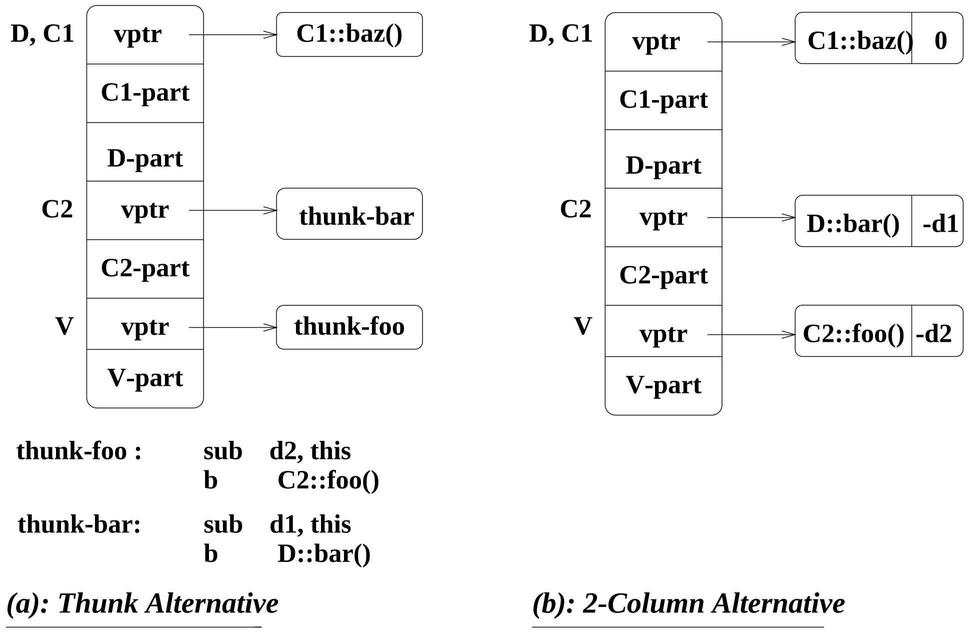


**Figure 3**: Object and VFT Layout using the INTRODUCED alternative for the $D$ object that is illustrated in the class inheritance graph in Figure 1(b).

### 3.3 Four VFT layouts

When the width and height alternatives are combined, there are four possible VFT layouts and the corresponding virtual dispatch mechanisms: 2-COLUMN VISIBLE, 2-COLUMN INTRODUCED, THUNK VISIBLE, and THUNK INTRODUCED. This section presents the four VFT layouts for the most derived class, $D$, in a simple class hierarchy, illustrated in Figure 1(b), that has both multiple and virtual inheritance.

Figure 2 shows the VFT layouts and thunk code for $D$ using the VISIBLE alternative, and Figure 3 shows the VFT layouts and thunk code for $D$ using the INTRODUCED alternative. In both figures, `d1` and `d2` are THIS adjustments: `d1` is the difference between the offsets of $D$ and $C2$ in a $D$ object, and `d2` is the difference between the offsets of $C2$ and $V$ in a $D$ object. The figure does not show any code at a virtual function call site; that is, mandatory dispatch code; early cast code, if required, in the INTRODUCED alternative; and late cast code in the 2-COLUMN alternative are not shown.

For the VISIBLE alternative, Figure 2(a) shows the VFT layout and thunk code using the THUNK alternative, and Figure 2(b) shows the VFT layouts for $D$ using the 2-COLUMN alternative. The THUNK alternative reduces space overhead of VFTs, since no THIS adjustment values are needed. In addition, this alternative also reduces the space overhead of virtual dispatch code size, since only half of the entries require late cast code.

For the INTRODUCED alternative, Figure 3(a) shows the VFT layout and thunk codes using the THUNK alternative, and Figure 3(b) shows the VFT layout using the 2-COLUMN alternative. The THUNK alternative, again, reduces the space overhead for both VFTs and virtual dispatch code; however, the reduction is not as great as in the VISIBLE alternative.

As Figures 2 and 3 illustrate, the number of VFT entries required for the VISIBLE alternative is significantly more, twice as much as the number of VFT entries required by the INTRODUCED alternative. In addition, the number of thunks that is required by the VISIBLE alternative is also increased. Section 3.4 examines this issue more closely. One could surmise that in general, the THUNK alternative will be more space efficient than the 2-COLUMN alternative. In particular, in the absence of multiple or virtual inheritance, since no late casts are required, no thunk code is generated. Nevertheless, it is hard to draw conclusions about time overhead without dynamic information, in particular how often a virtual function call site executes. In Section 4, we develop analytical models for space and time costs. Section 5 analyzes the space and time overhead for the four VFT layouts on a set of C++ benchmarks.

## 3.4 When is a Virtual Dispatch Mechanism Required to Perform an Early and/or a Late Cast?

This section demonstrates that the decision of when a virtual dispatch mechanism is required to perform an early and/or a late cast is a function of where virtual function definitions are placed in a class hierarchy and how VPTRs are shared in an object's layout [5]. The thunk alternative will perform a late cast only when required. The 2-COLUMN alternative is inefficient in that it always performs a late cast whether or not a late cast is required. Our results, in Section 5 provide a quantitative study of this effect.

Consider, the virtual function call site $p \rightarrow f()$; where $D$ is the dynamic type of the object that is pointed to by $p$, $S$ is the static type of $p$, and $I$ is the introducing class of $f$, i.e., $I$ defines $f$ and no base of $I$ defines $f$. $I$ can be $S$ or a base of $S$. Let $E :: f()$ be the actual function that will be invoked; that is, $f$ is defined in classed $E$. We now analyze when $f$ requires an early or a late cast for each of the four VFT layouts.

### 3.4.1 Single Inheritance without Virtual Inheritance

Assume that $D$ and all of its base classes only use single inheritance; that is, there is no multiple or virtual inheritance. As shown in Figure 1(a), only one VPTR per object is required. In this case, no early or late casts are required. One would expect that the THUNK alternative will be the best choice for single inheritance with no virtual inheritance since no late casts are required. In addition, the VISIBLE and INTRODUCED alternatives combined with the THUNK alternative and 2-COLUMN have the same the number of VFT entries, and do not require early casts.

### 3.4.2 Multiple Inheritance without Virtual Inheritance

Assume that $D$ and its base classes have no virtual inheritance, but some of these classes have multiple inheritance. Multiple VPTRs are required in a fully derived $D$ object. Each VPTR is located at a different offset from the start of a $D$ object. It is possible to identify when an early and a late cast is required based on the relative position of the classes $I$, $E$ and $S$ in the class hierarchy and on an object layout algorithm that determines VPTR sharing between subobjects.

For the INTRODUCED alternative, an early cast is required if $S \neq I$, and $S$ and $I$ do not share VPTRs, and a late cast is required if $I \neq E$, and $I$ and $E$ do not share VPTRs. For the VISIBLE alternative, a late cast is required if $S \neq E$, and $S$ and $E$ do not share VPTRs.

### 3.4.3 Virtual Inheritance

Assume that class $D$ has at least one virtual base class. Multiple VPTRs are required in a fully derived $D$ object. It is possible to identify when an early and a late cast is required based on the

[5] The object layout algorithm is not important to this discussion, only what suboject's share VPTRs.

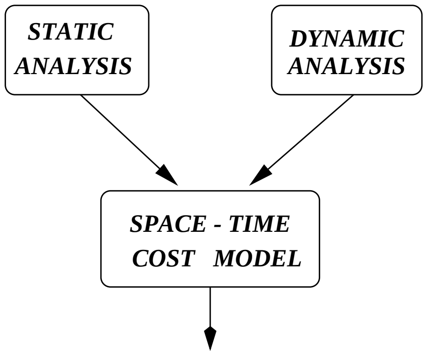


**Figure 4**: Framework for evaluating Virtual Dispatch Mechanisms

relative position of the classes $I$, $E$ and $S$ in the class hierarchy and on an object layout algorithm that determines VPTR sharing between subobjects.

For the INTRODUCED alternative, an early cast is required if $I$ is a virtual base of $S$, and a late cast is required if $I \neq E$, and $I$ and $E$ do not share VPTRs. For the VISIBLE alternative, a late cast is required if $S \neq E$, and $S$ and $E$ do not share VPTRs.

# 4 Framework for Evaluating Virtual Dispatch Mechanisms

This section presents a framework to evaluate virtual dispatch mechanisms. The framework's architecture is illustrated in Figure 4 and consists of three modules: the *static analysis module* collects, at a program's compile-time, static measurement information; the *dynamic analysis module* collects, at a program's execution-time, dynamic measurement information; and *space and time cost module* analytically models the space and time overhead of different virtual dispatch mechanisms.

For a C++ program's execution, a particular virtual dispatch mechanism's space and time overhead can be estimated from the static measurement information of the program, the dynamic measurement information of an execution of this program, and the space and time cost models for that virtual dispatch mechanism. The rest of this section describes each of these modules in the context of the virtual dispatch mechanisms discussed in Section 3.3, and discusses how the measurement information and space and time overheads are computed. The framework is extensible: it can accommodate other virtual dispatch mechanisms as well as other object and VFT layouts that are not discussed in this paper. Extensions may be incorporated into the framework by adding new cost equations and algorithms to the space and time cost module, and by extending the static and dynamic analysis modules to compute any additional information that is required by the new cost equations and algorithms.

## 4.1 Static Analysis Module

The static measurement information that is computed by the static analysis module and that is used in computing space and time overhead for a virtual dispatch mechanism is:

- **Define**$(f, C)$: the class whose definition of the virtual function $f$ is executed when a subobject, of a fully derived $C$ object, invokes $f$,
- **Introduced**$(C)$: the set of virtual functions that are introduced in class $C$,

- **Introduced**($f$, $C$): the class $X$ that is either $C$ or a base class of $C$, such that $X$ declares $f$, and no base of $X$ declares $f$,
- **NumberOfVirtualSites**: the number of virtual function call sites in a program,
- **NumberOfNVCastSites**: the number of virtual function call sites in a program that require an early cast that does not access a VBPTR,
- **NumberOfVCastSites**: the number of virtual function call sites in a program that require an early cast that accesses a VBPTR,
- **NVBase**($C$): the set of direct, nonvirtual base classes of a class $C$,
- **Offset**($C$, $B$, $E$): the offset of a $B$ subobject from an $E$ subobject in a fully derived $C$ object,
- **ShareVPTR**($B$, $E$): a boolean function that returns true if $B$ and $E$ share VPTRs,
- **VBase**($C$): the set of virtual base classes of a class $C$, and
- **Visible**(C): the set of virtual functions that are visible in class $C$.

This information is computed by using the *Class Hierarchy Graph* (CHG) data structure [3]. The CHG is a graph with nodes representing C++ classes and edges representing inheritance relationship between classes. By traversing this graph, the values for the functions **Define**($f$, $C$), **Introduced**($B$), **Introduced**($f$, $C$), **NVBase**($C$), **VBase**($C$), and **Visible**($B$), are computed. In addition, object layout information is extracted from the compiler and stored in each class node $C$. The object layout information stored in $C$ includes what subobject share VPTRs, and information on the list of subobjects and their offsets from the beginning of a fully derived $C$ object. From the object layout information, the return values for the functions **Offset**($C$, $B$, $E$), and **ShareVPTR**($B$, $E$) are computed.

In addition to the CHG, the *Program Virtual-call Graph* (PVG) [4] is used to compute call site information. The PVG records the set of virtual function call sites, $\mathcal{V}(S,I)$, where $S$ and $I$ are the static type and the introducing class at the call site. From the PVG, the **NumberOfVirtualSites** is computed, and from the CHG and the PVG the **NumberOfNVCastSites** and **NumberOfVCastSites** are computed.

### 4.2 Dynamic Analysis Module

The dynamic measurement information that is computed by the dynamic analysis module and that is used in computing time overhead is:

- **TotalNumberOfVirtualCalls**: the total number of virtual function calls that occur in the program's execution,
- **NumberOfCalls**($\mathcal{V}$): the number of times a virtual function call site, $\mathcal{V}$, is called in a program's execution,
- **DynamicTypes**($\mathcal{V}$): the set of dynamic types that occur in the invocation of the virtual function call site $\mathcal{V}$, and
- **DynamicCount**($\mathcal{V}$, $D$): the number of times that dynamic type $D$ occurs in an invocation of the virtual function call site $\mathcal{V}$.

This information is computed by simulating a program's dynamic call graph from an address trace of the program's execution. The address trace is obtained from the tracing tool *xtrace* [15]. The trick is to determine the dynamic type of the expression that invokes a virtual function, since this information is not available from an address trace. The dynamic type is obtained by recording the dynamic type of a subobject and its VPTR address at construction-time in the trace when the subobject's VPTR is initialized, determining the VPTR address at virtual function invocation time when the VPTR address is accessed, and looking up the VPTR address to find the dynamic type at the virtual function call site. From the simulation, all the dynamic measurement information can be computed.

## 4.3 Space and Time Cost Module

This section presents the cost formulas and algorithms that are used to model the space and time overhead of the virtual dispatch mechanisms for the four different VFT layouts that are presented in Section 3.3. The space cost model (Section 4.3.1) uses fixed costs, and only static measurement information. The time cost model (Section 4.3.2) use fixed costs, static measurement information, and also dynamic measurement information for a particular execution of a program. The *fixed costs* for both space and time are a function of the target architecture that determines the code sequences that are required to implement object-oriented language features and the cycle times of different instructions. The space and time cost overhead can be customized for different architectures by setting the fixed costs to the appropriate values for that architecture.

### 4.3.1 Space Cost Module

The space overhead of a virtual dispatch mechanism includes the space occupied by VFTs, by thunk code, and by THIS adjustment and mandatory dispatch code that is generated statically at virtual function call sites. The THIS adjustment can occur either for early casts to base subobjects (in the INTRODUCED alternative), or for late casts (in the 2-COLUMN alternative). The code sequences for early cast code, mandatory dispatch code, thunk code, and late cast code are presented in Section 3. The following equations are used to compute the space overhead:

1. The space overhead of VFTs, **SpaceForVFTs**, is the total number of VFT entries for a program, **NumberOfEntries**, times the size of a VFT entry, **SizeOfVFT_Entry**. **SizeOfVFT_Entry** is either the size of an address for the THUNK alternative, or else the size of an address and a THIS adjustment value for the 2-COLUMN alternative:

$$\mathbf{SpaceForVFTs} = \mathbf{NumberOfEntries} * \mathbf{SizeOfVFT\_Entry}$$

2. The INTRODUCED alternative space overhead for early casts that access a VBPTR is the total number of virtual function call sites that require an early cast that accesses a VBPTR, **NumberOfVCastSites**, times the code size of such an early cast, **SizeOfVCastCode**. The space overhead for those early casts that do not access a VBPTR is the total number of virtual function call sites where such an early cast is required, **NumberOfNVCastSites**, times the code size of such an early cast, **SizeOfNVCastCode**. The total space overhead for early casts, **SpaceForEarlyCasts**, is:

$$\begin{aligned}\mathbf{SpaceForEarlyCasts} = {} & (\mathbf{NumberOfVCastSites} * \mathbf{SizeOfVCastCode}) + \\ & (\mathbf{NumberOfNVCastSites} * \mathbf{SizeOfNVCastCode})\end{aligned}$$

3. The space overhead for mandatory dispatch code, **SpaceForMandatoryDispatch**, is the total number of virtual function call sites, **NumberOfVirtualSites**, times the size of mandatory dispatch code, **SizeOfMandatoryDispatchCode**.

$$\begin{aligned}\textbf{SpaceForMandatoryDispatch} &= \textbf{NumberOfVirtualSites} * \\ &\quad \textbf{SizeOfMandatoryDispatchCode}\end{aligned}$$

4. The THUNK alternative space overhead for the thunk code, **SpaceForThunks**, is the total number of thunks generated for a program, **NumberOfThunks**, times the size of the thunk code, **SizeOfThunkCode**:

$$\textbf{SpaceForThunks} = \textbf{NumberOfThunks} * \textbf{SizeOfThunkCode}$$

5. The 2-COLUMN alternative space overhead for late casts, **SpaceForLateCasts**, is the total number of virtual function call sites, **NumberOfVirtualSites**, times the size of late cast code, **SizeOfLateCastCode**:

$$\textbf{SpaceForLateCasts} = \textbf{NumberOfVirtualSites} * \textbf{SizeOfLateCastCode}$$

The quantities **NumberOfVCastSites**, **NumberOfNVCastSites** and **NumberOfVirtualSites** are obtained from the static measurement information. The quantities **SizeOfVFT_Entry**, **SizeOfNVCastCode**, **SizeOfVCastCode**, **SizeOfMandatoryDispatchCode**, **SizeOfThunkCode**, and **SizeOfLateCastCode** are fixed costs whose values are determined by the target architecture. The computation of the quantities, **NumberOfEntries** and **NumberOfThunks**, is described below.

### Computing NumberOfEntries

The number of VFT entries, for both the VISIBLE and the INTRODUCED alternatives, is computed as the sum of the VFT entries for each class $C$, **Entries**($C$). **Entries**($C$) accounts for the entries in all the VFTs that VPTRs point to in a fully derived object of type $C$. In the presence of virtual bases, the VPTRs in the virtual base subobjects and the corresponding VFT entries are referred to as the *virtual part* of the object. All other VPTRs and VFT entries are referred to as the *nonvirtual part* of the object. **Entries**($C$) is recursively computed as the sum of the entries in the nonvirtual part, **Entries**($C$, **nonvirtual**), plus the entries in the virtual part, **Entries**($C$, **virtual**).

$$\textbf{Entries}(C) = \textbf{Entries}(C, \textbf{nonvirtual}) + \textbf{Entries}(C, \textbf{virtual})$$

$$\textbf{Entries}(C, \textbf{nonvirtual}) = X + \sum_{B \in \textbf{NVBase}(C)} \textbf{Entries}(B)$$

$$\textbf{Entries}(C, \textbf{virtual}) = \sum_{V \in \textbf{VBase}(C)} \textbf{Entries}(V)$$

where **NVBase**($C$) and **VBase**($C$) are obtained from the static measurement information, and $X$ is either the number of virtual functions visible in class $C$ less the number of virtual functions visible in all classes whose subobjects share a VPTR with $C$'s subobject (VISIBLE alternative), or the number of virtual functions that are introduced in $C$ (INTRODUCED alternative).

```
ComputeNumberOfThunks(alternative)
  NumberOfThunks = 0
  for all classes C in the program do
    for all bases B of C in the class hierarchy do
      for each f ∈ VFTEntries(B, alternative) do
        E = Define(f, C)
        if B ≠ E and NOT ShareVPTR(B, E) then
            delta = Offset(C, B, E)
            if thunk not generated for f with delta then
              generate thunk code
              record thunk code generated for f with delta
              NumberOfThunks += 1
  return NumberOfThunks

VFTEntries(B, alternative)
  if alternative = VISIBLE then
    return Visible(B)
  if alternative = INTRODUCED then
    return Introduced(B)
```

**Figure 5**: Algorithm to Compute, for both the VISIBLE and the INTRODUCED alternatives, the number of thunks that are generated by a program.

### Computing NumberOfThunks

The algorithm in Figure 5 computes, for both the VISIBLE and the INTRODUCED alternatives, the number of thunks that are generated for a program. The algorithm works by determining if a late cast is required to call a virtual function from a subobject in an object. Although duplicate VFT entries (that is, VFT entries with the same function address and THIS adjustment) are possible in the 2-COLUMN VFT layout, duplicate thunk codes can be avoided in the THUNK alternative by generating a single thunk code for every function address and this adjustment pair. Algorithm **ComputeNumberOfThunks** avoids duplicate thunk code by recording when thunk code is generated for each function and THIS adjustment pair, and then by generating thunk code only if it has not already been generated. The functions **Define**($f$, $C$), **Introduced**($B$), **Offset**($C$, $B$, $E$), **ShareVPTR**($B$, $E$), and **Visible**($B$) are all defined in the static measurement information.

#### 4.3.2 Time Cost Model

The time overhead of a virtual dispatch mechanism consists of the time to execute early cast code, mandatory dispatch code, and late cast code for a virtual dispatched call. The following equations are used to compute time overhead:

1. The INTRODUCED alternative time overhead for an early cast, **TimeForEarlyCasts**, is the sum of the early cast time at each virtual function call site. Consider a virtual function call site, $\mathcal{V}(S, I) = p \rightarrow f()$, where $S$ is the static type of the object that $p$ points to and $I$ is the class introducing function $f$. The early cast time for $\mathcal{V}$ is the total number of times $\mathcal{V}$ is

```
ComputeThunkLateCastTime(alternative)
  TimeForThunks = 0
  for each virtual function call site, V(S,I) = p → f() do
    for each D ∈ DynamicTypes(V) do
      E = Define(f, D)
      if (ThunkRequired(E,I,S,alternative)) then
        TimeForThunks += DynamicCount(V, D) * CostOfThunk
  return TimeForThunks

ThunkRequired(E,I,S, alternative)
  if alternative = VISIBLE then
    return(E ≠ S ∧ NOT ShareVPTR(E, S))
  if alternative = INTRODUCED then
    return(E ≠ I ∧ NOT ShareVPTR(E, I)
```

**Figure 6**: Algorithm to compute, for both the VISIBLE and the INTRODUCED alternatives, the late cast cost for a particular execution of a program.

invoked, **NumberOfCalls**($\mathcal{V}$), times the cost of an early cast:

$$\sum_{\mathcal{V}(S,I)} \mathbf{NumberOfCalls}(\mathcal{V}) * \begin{cases} 0 & \text{No early cast} \\ \mathbf{CostOfVCast} & \text{Early cast uses VBPTR} \\ \mathbf{CostOfNVCast} & \text{Otherwise} \end{cases}$$

The **NumberOfCalls**($\mathcal{V}$) quantity is obtained from the dynamic measurement information.

2. The time overhead for mandatory dispatch code, **TimeForMandatoryDispatch**, is the total number of virtual function calls that occur in a program's execution, **TotalNumberOfVirtualCalls**, times the cost to execute mandatory dispatch code, **CostOfMandatoryDispatch**.

$$\begin{aligned} \mathbf{TimeForMandatoryDispatch} &= \mathbf{TotalNumberOfVirtualCalls} * \\ &\quad \mathbf{CostOfMandatoryDispatch} \end{aligned}$$

The **TotalNumberOfVirtualCalls** quantity is obtained from the dynamic measurement information.

3. The THUNK alternative time overhead for late casts, **TimeForThunks**, is the total number of virtual function calls to thunk code times the time to execute the thunk code, **CostOfThunk**. The algorithm **ComputeThunkLateCastTime**, presented in Figure 6, computes **TimeForThunks** for a specific execution of a C++ program. It is parameterized by the VISIBLE or the INTRODUCED alternative. The algorithm works by examining every virtual function call site. The dynamic measurement information is consulted to determine if a virtual function call site is invoked, and if it is, to provide the set of dynamic types that occurred at that call site. For each dynamic type in the set, the analysis determines if a thunk is required, and if it is, **TimeForThunks** is incremented by **CostOfThunk** times the number of times this call site is executed with this dynamic type. The quantities **Define**($f$, $D$), and **ShareVPTR**($B$, $E$) are obtained from the static measurements information, and the quantities **DynamicTypes**($\mathcal{V}$), and **DynamicCount**($\mathcal{V}$, $D$) are obtained from the dynamic measurements information.

| Benchmarks | Lines of Code | Description | Inheritance | |
|---|---|---|---|---|
| | | | multiple | virtual |
| fresco | 11,157 | IDL specification to C++ code translator | No | No |
| sched | 5,712 | RS/6000 Instruction Timing Simulator | Yes | No |
| metaware | 288 | C++ benchmark kernel from Metaware, Inc. | Yes | Yes |

**Table 1**: C++ benchmarks categorized by class inheritance. Size is given in non-blank lines of code.

4. The 2-COLUMN alternative time overhead for a late cast, **TimeForLateCasts**, is the total number of virtual function calls that occur in a program's execution, **TotalNumberOfVirtualCalls**, times the cost to perform a late cast, **CostOfLateCast**.

$$\mathbf{TimeForLateCasts} \quad = \quad \mathbf{TotalNumberOfVirtualCalls} * \mathbf{CostOfLateCast}$$

   The **TotalNumberOfVirtualCalls** quantity is obtained from the dynamic measurement information.

The quantities **CostOfNVCast**, **CostOfVCast**, **CostOfMandatoryDispatch**, **CostOfThunk**, and **CostOfLateCast** are fixed costs whose values are determined by the target architecture.

# 5 Experimental Results

This section presents the set of benchmarks (Section 5.1) that are used to measure the space (Section 5.2) and time (Section 5.3) overhead for different virtual dispatch mechanisms.

Before discussing our results in detail, the following two explanatory remarks are necessary. First, all of our statistics are for *user code only*. This is because our static and dynamic analysis requires source code. This makes the evaluation of the library code impossible. Second, our results assume a simple target architecture such that the time to execute an instruction is one unit of time (or cycle), irrespective of the type of instruction, and that all instructions take one unit of space. The architecture's psuedo code for the virtual dispatch mechanisms is presented in Section 3. This simple target architecture allows the results to be presented independently of the specific hardware features of a particular architecture. Nevertheless, by substituting in different code sequences and their space and time constraints for a specific target architecture, our results can be extended to that architecture.

## 5.1 Benchmarks

The space and time overhead for the four virtual dispatch mechanisms (that are presented in Section 3.3) is computed for three C++ benchmarks that are described in Table 1. The table gives a broad overview of the type of inheritance that each benchmark uses for user-defined classes: FRESCO uses only single inheritance, SCHED uses single and multiple inheritance, and METAWARE uses multiple and virtual inheritance. The fact that the combination of multiple and virtual inheritance differs for all three benchmarks helps to illustrate the affect of inheritance on the space and time overhead. Both FRESCO and SCHED are real C++ applications. Although METAWARE is a synthetic kernel benchmark and not a realistic program, its use of virtual inheritance is instructive.

Table 2 presents more detailed information on class inheritance. The '**# of classes**' column is the sum of all the columns to its right. The labels '**NV**' and '**V**' refer to nonvirtual and virtual

| Benchmarks | max depth | # of classes | # of singletons | # of roots | | # of classes which directly inherit | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | single | | multiple | |
| | | | | NV | V | NV | V | NV | NV & V |
| fresco | 4 | 89 | 43 (48%) | 5 | 0 | 41 (46%) | 0 | 0 | 0 |
| sched | 3 | 26 | 14 (54%) | 5 | 0 | 3 (12%) | 0 | 4 (15%) | 0 |
| metaware | 5 | 19 | 0 | 0 | 4 | 4 (21%) | 5 (26%) | 0 | 6 (32%) |

**Table 2**: Inheritance statistics for user-defined classes in C++ benchmarks. The labels '**NV**' and '**V**' refer to nonvirtual and virtual.

| Benchmarks | early cast | | thunk code | | late cast | VFT entry | |
|---|---|---|---|---|---|---|---|
| | NVCast | VCast | intro. | VISIBLE | 2-COLUMN | intro. | VISIBLE |
| fresco | 0 | 0 | 0 | 0 | 465 | 1082 | 1082 |
| sched | 1 | 0 | 19 | 41 | 23 | 102 | 141 |
| metaware | 0 | 5 | 0 | 9 | 5 | 10 | 24 |

**Table 3**: Space overhead: number of times a space overhead event occurs. The '**intro.**' column represent the INTRODUCED alternative. The '**late cast**' column is also the number of virtual function call sites that occur in the user's code.

respectively. For example, the column labeled '**NV**' under the columns labeled '**single**' and '**# of classes which directly inherit**' is the number of classes that directly inherit only one class, and the inheritance is nonvirtual. Approximately 50% of the classes of FRESCO and SCHED are singletons: a singleton is a class that is neither a base nor a derived class of any other class. The FRESCO benchmark has no multiple or virtual inheritance. Its maximum depth of class inheritance is four. The SCHED benchmark has no virtual inheritance. Although only 15% (or 4) of its classes directly inherit multiple base classes, an addition 12% (or 3) of its classes indirectly inherit multiple base classes. Its maximum depth of class inheritance is three. The benchmark METAWARE has 79% (or 15) of its user-defined classes that are either a virtual base or inherit a virtual base, 42% (or 8) of its classes are virtual bases, and only 16% (or 3) of its classes are leaves. Its maximum depth of class inheritance is five.

### 5.2 Space Overhead

This section presents, for our set of three benchmarks, the space overhead that is required for each of the four virtual dispatch mechanisms that this paper studies. The space overhead is computed as the number of times an event occurs. The *space overhead events* are: an early cast at a virtual function call site that is required for the INTRODUCED alternative; mandatory dispatch code at a virtual function call site; a late cast at a virtual function call site that is required for the 2-COLUMN alternative; thunk code that is required for the THUNK alternative; and an entry for a virtual function in a VFT [6]. Table 3 presents the number of times each event occurs for a benchmark. An early cast is distinguished by whether or not a VBPTR is accessed: a **VCast** requires a VBPTR access, while a **NVCast** does not. Since the 2-COLUMN alternative performs a late cast at every virtual function call, the '**late cast**' column is also the number of virtual function

[6]In this paper we do not model construction VFTs; that is, VFTs that are used only at object construction time.

| Benchmarks | THUNK | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | mandatory dispatch code | INTRODUCED | | | | VISIBLE | | | |
| | | early cast | thunk code | VFT entry | total | thunk code | VFT entry | total |
| fresco | 1395 | 0 | 0 | 1082 | 2477 | 0 | 1082 | 2477 |
| sched | 69 | 0 | 38 | 102 | 209 | 82 | 141 | 292 |
| metaware | 15 | 10 | 0 | 10 | 35 | 18 | 24 | 57 |

**Table 4**: Space overhead: total number of words of storage that are required for the THUNK alternative when a mandatory dispatch code takes three words of storage, an early cast (**VCast**) and thunk code both take two words of storage, and a VFT entry takes one word of storage. An **NVCast** is not considered.

| Benchmarks | 2-COLUMN | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | mandatory dispatch code | INTRODUCED | | | | VISIBLE | | |
| | | early cast | late cast | VFT entry | total | late cast | VFT entry | total |
| fresco | 1395 | 0 | 930 | 2164 | 4489 | 930 | 2164 | 4489 |
| sched | 69 | 0 | 46 | 204 | 319 | 46 | 282 | 397 |
| metaware | 15 | 10 | 10 | 20 | 55 | 10 | 48 | 73 |

**Table 5**: Space overhead: total number of words of storage that are required for the 2-COLUMN alternative when a mandatory dispatch code takes three words of storage, an early cast (**VCast**) and a late cast both take two words of storage, and a VFT entry takes one word of storage. An **NVCast** is not considered.

call sites that occur in the user's code.

Tables 4 and 5 present the number of words of storage, broken down by event and totaled, that is required for each virtual dispatch mechanism. The alternatives that are incorporated into a virtual dispatch mechanism determines the events that are used to compute that dispatch mechanism's space overhead. The space overhead for a particular event and benchmark is computed from the number of times that event occurs in that benchmark (this value is obtained from Table 3) times the number of words of storage that that event requires. For our simple target architecture, a VFT entry takes one word of storage for the THUNK alternative, and two words of storage for the 2-COLUMN alternative; **VCast**, late cast and thunk code all take two words of storage [7]; and mandatory dispatch code takes three words of storage. Thus, virtual dispatch code takes at least three words of storage, if no **VCast** or late cast are performed, and at most seven words of storage, if both a **VCast** and a late cast are performed.

Within the THUNK and 2-COLUMN alternatives, the space overhead for FRESCO is the same for the INTRODUCED and VISIBLE alternatives. This result is expected since, for single inheritance, each object has only one VPTR; thus, no early or late casts are required. For SCHED and METAWARE,

[7] A **NVCast** is expected to be incorporated into an indexed load instruction that loads the VPTR [12]. We present the number of times **NVCast** events occur for completeness and generality so that the results may be used for other architectures.

| Benchmarks | early cast | | thunk call | | late cast |
|---|---|---|---|---|---|
| | NVCast | VCast | INTRODUCED | VISIBLE | 2-COLUMN |
| fresco | 0 | 0 | 0 | 0 | 45,991 |
| sched | 36,682 | 0 | 466,002 | 429,370 | 967,789 |
| metaware | 0 | 50,000 | 0 | 50,000 | 50,000 |

**Table 6**: Time overhead: number of times a time overhead event occurs. The '**late cast**' column is also the number of virtual function calls that occur in the user's code.

that may have multiple VPTRs per object, the VISIBLE alternative requires more space: 40% (63%) more for the THUNK alternative and 24% (33%) more for the 2-COLUMN alternative for SCHED (METAWARE). This result is expected since the INTRODUCED alternative requires fewer VFT entries than the VISIBLE alternative. Furthermore, the amount of THUNK code in the THUNK alternative also has an effect on overhead: The INTRODUCED alternative requires less thunk code than the VISIBLE alternative. Although this may not be the case for all benchmarks, it is the expected case.

Examining the space overhead within the INTRODUCED and VISIBLE alternative (across tables 4 and 5), the space overhead increases with the 2-COLUMN alternative. FRESCO's space overhead increases 81% with the 2-COLUMN alternative, SCHED increases 53% for the INTRODUCED alternative and 36% for the VISIBLE alternative, and METAWARE increases 57% for the INTRODUCED alternative and 28% for the VISIBLE alternative. This increase is expected because the 2-COLUMN alternative doubles the size of a VFT entry and always requires a late cast at a virtual function call site.

In summary, the THUNK INTRODUCED virtual dispatch mechanism uses the least space overhead: the next best virtual dispatch mechanism requires from 40 to 81% more space. If only single inheritance occurs, the THUNK alternative can be combined with either the INTRODUCED or the VISIBLE alternatives to require the least space overhead. If multiple or virtual inheritance is used, the THUNK INTRODUCED virtual dispatch mechanism requires the least space overhead since the THUNK alternative reduces late casts and the INTRODUCED alternative, although introducing early cast code, reduces VFT size which is the dominant factor in space overhead.

### 5.3 Time Overhead

This section presents, for our set of benchmarks, the time overhead that is required for each of the four virtual dispatch mechanisms that this paper studies. To compute the time overhead, we first compute the number of times an event that contributes to time overhead occurs. The *time overhead events* are: an early cast at a virtual function call site that is required for the INTRODUCED alternative; an execution of mandatory dispatch code that is required for all alternatives; a late cast at a virtual function call site that is required for the 2-COLUMN alternative; and a call to thunk code that is required for the THUNK alternative. Table 6 presents the number of times events occur for each benchmark. As with space overhead, an early cast is distinguished by whether or not a VBPTR is accessed. Since the 2-COLUMN alternative performs a late cast at every virtual function call, the '**late cast**' column is also the number of virtual function calls that are invoked in the user's code.

Tables 7 and 8 presents the number of instructions, broken down by event and totaled, that is required for each virtual dispatch mechanism. The value for a particular event and benchmark is computed from the number of times that event occurs in that benchmark (this value is obtained from Table 6) times the number of instructions that that event requires. For our simple target

| Benchmarks | THUNK | | | | | |
|---|---|---|---|---|---|---|
| | mandatory dispatch code | INTRODUCED | | | VISIBLE | |
| | | early cast | thunk call | total | thunk call | total |
| fresco | 137,973 | 0 | 0 | 137,973 | 0 | 137,973 |
| sched | 2,903,367 | 0 | 932,004 | 3,835,371 | 858,740 | 3,762,107 |
| metaware | 150,000 | 100,000 | 0 | 250,000 | 100,000 | 250,000 |

**Table 7**: Time overhead: total number of instructions that are required for the THUNK alternative when mandatory dispatch code takes three instructions to execute, an early cast (**VCast**) and thunk code take two instructions to execute. An **NVCast** is not considered.

| Benchmarks | 2-COLUMN | | | | | |
|---|---|---|---|---|---|---|
| | mandatory dispatch code | INTRODUCED | | | VISIBLE | |
| | | early cast | late cast | total | late cast | total |
| fresco | 137,973 | 0 | 91,982 | 229,955 | 91,982 | 229,955 |
| sched | 2,903,367 | 0 | 1,935,578 | 4,838,945 | 1,935,578 | 4,838,945 |
| metaware | 150,000 | 100,000 | 100,000 | 350,000 | 100,000 | 250,000 |

**Table 8**: Time overhead: total number of instructions that are required for the 2-COLUMN alternative when mandatory dispatch code takes three instructions to execute, and an early cast (**VCast**) and a late cast take two instructions to execute. An **NVCast** is not considered.

architecture, a **VCast** [8], late cast and thunk code all take two instructions to execute, and mandatory dispatch code takes three instructions to execute. Thus, virtual dispatch takes at least three instructions to execute, if no **VCast** or late cast is performed, and at most seven instructions to execute, if both a **VCast** and a late cast are performed.

Within the THUNK or 2-COLUMN alternative, the time overhead for two of the three benchmarks does not differ between the INTRODUCED and the VISIBLE alternatives. This is expected for FRESCO since it has only single inheritance, and is expected for SCHED in the 2-COLUMN alternative since all virtual function calls perform a late cast and no **VCast** is performed since no virtual inheritance occurs. METAWARE's behavior in the THUNK alternative is not expected, but is easily explained. Of the virtual functions that were called, all were introduced in a virtual base and never overridden by any base of the dynamic type of the object that invoked the virtual function. In this case, the INTRODUCED alternative requires an early cast, but no late cast, for each virtual function call; while the VISIBLE alternative required a late cast, but no early cast, for each virtual function call. Since a **VCast** and a late cast require the same number of instructions, METAWARE's time overhead for the INTRODUCED and VISIBLE alternatives is the same. On the other hand, the time overhead of SCHED in the THUNK alternative and METAWARE in the 2-COLUMN alternative differs between the INTRODUCED and VISIBLE alternatives. SCHED's time overhead differs by approximately 2%: although no **VCasts** occur, late cast time overheads differs by 9%. The INTRODUCED alternative increases METAWARE time overhead by 40% over the VISIBLE alternative because every virtual function call in the INTRODUCED alternative requires a **VCast** in addition to a late cast.

Examining the time overhead within the INTRODUCED and VISIBLE alternatives (across tables 7 and 8), the time overhead either stays the same or increases with the 2-COLUMN alternative. FRESCO's time overhead increases 67%, SCHED increases 26% for the INTRODUCED alternative and 29% for the VISIBLE alternative, and METAWARE increases 40% for the INTRODUCED alternative and has no increase for the VISIBLE alternative. No increase occurs because every virtual function call requires a thunk call in the THUNK VISIBLE alternative.

In general, the THUNK alternative has the least time overhead. When only single inheritance occurs, the INTRODUCED and VISIBLE alternatives have no effect on time overhead. When multiple inheritance and no virtual inheritance is considered, it is not clear whether the INTRODUCED or the VISIBLE alternative should be combined with the THUNK alternative. Although for an execution of the SCHED benchmark, the VISIBLE alternative is slightly better, it is not clear that this is always the case. Whether the INTRODUCED or the VISIBLE alternative is a better combination with the THUNK alternative is a function of the benchmark and its execution; that is, a function of where virtual functions are overridden in the class hierarchy and what is the dynamic type of the objects that are invoking the virtual function. When virtual inheritance is considered, one would believe that the VISIBLE alternative would have less time overhead than the INTRODUCED alternative because of the extra cost of the **VCast**. Nevertheless, an execution of the METAWARE benchmark contradicts this assumption: there is no difference between the THUNK INTRODUCED and THUNK VISIBLE alternatives because $I + E$ everytime.

# 6 Future Work

We are in the process of extending our framework for analytically modeling the space and time overhead of additional VFT layouts, and of object layouts. These extensions will allow us to determine

[8] As with space overhead, an **NVCast** is expected to be incorporated into the instruction to load the VPTR and has no affect on time overhead. We present the number of times **NVCast** events occur for completeness and generality so that the results may used for other architectures.

the effect of alternative object models on space and time overhead.

Our analytical time model computes the number of instructions required for a particular virtual dispatch mechanism. In contrast, a more architecture dependent analysis would be insightful [9]. For example determining the number of cycles that a particular virtual dispatch mechanism requires for various programs' executions, or modeling the performance overhead on a superscalar architecture.

In addition, an interesting extension of this work would be to evaluate virtual dispatch overhead after false polymorphism is detected and removed.

# 7 Conclusions

This paper presents an extensible framework for analytically modeling and measuring the space and time overhead of virtual dispatch mechanisms for four different VFT layouts and a set of benchmarks that have different class inheritance properties. The four VFT layouts are either generally known [10] or else used by a production C++ compiler. This paper is the first study, known to the authors, that quantitatively compares VFT layouts.

Our results, on a set of three C++ benchmarks, demonstrate that the VFT layout has a significant impact on space and time overhead for the virtual dispatch mechanism. Alternative VFT layouts varied by as much as two times for space overhead and by as much as 67% for time overhead on the benchmarks that we used. Our results also demonstrate that virtual dispatch performance may be affected by the source code's use of inheritance (single, multiple, and/or virtual), and where virtual functions are defined in the class hierarchy.

We conclude that of the four VFT layouts that were studied, the THUNK alternative is better than the 2-COLUMN alternative for both space and time overhead; the INTRODUCED alternative is better than the VISIBLE alternative for space overhead; and that the time overhead for the VISIBLE and INTRODUCED alternatives will vary depending on program properties.

# Acknowledgements

The authors thank Michael Karasick, Yee-Min Chee, Derek Lieber, Lee Nackman, and the other team members for their invaluable assistance with their prototype C++ compiler upon which our static analysis module was built.

The authors also thank Ted Law, Lee Nackman, and G. Ramalingam for their help in providing insights into the object model, David Bacon and G. Ramalingam for their assistance with the development of the static analysis module; Paul Carini for his discussions on this matter; Ian Carmichael for his assistance with his object model code that is used in the static analysis module to compute subobject offsets and VPTR sharing; Jim Larus for his support of our work; Ravi Nair for the use of his *xtrace* system, the SCHED benchmark, and for his technical assistance; David Streeter and Brian Thomson for their assistance with understanding the IBM XL C++ compiler on AIX; Karel Driesen and Urs Hölzle for their feedback on an earlier version of this paper; and Yong-Fong Lee and Mauricio Serrano for sharing their benchmark suite and their insights.

Finally, the authors especially thank Michael Burke for his help in providing insights into the object model, and his early comments on the equations and algorithms that are used in the time and space cost module.

## References


[1] Accredited Standards Committee. *Working Paper for Draft Proposed International Standard for Information Systems – Programming Language C++*, Apr. 1995. Doc. No. X3J16/95-0087.

[2] Ole Agesen and Urs Hölzle. Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages. In *Proceedings of the 1995 ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 91–107, Austin, Texas, October 1995. ACM Press, New York, New York.

[3] David F. Bacon and Harini Srinivasan. The class hierarchy graph and the program virtual-call graph: A basis for optimization of C++ programs. In preparation.

[4] David F. Bacon and Peter F. Sweeney. Rapid type inference in a C++ compiler. Submitted to PLDI'96.

[5] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in C++ programs. In *21st Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 397–408, Portland, OR, Jan. 1994.

[6] Paul Carini, Michael Hind, and Harini Srinivasan. Type analysis algorithm for C++. Technical report, IBM Thomas J. Watson Research Center, 1995.

[7] R. Dixon, T. McKee, P. Schweitzer, and M. Vaughan. A fast method dispatcher for compiled languages with multiple inheritance. In *Proceedings of the 1989 ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 211–214, New Orleans, LA, October 1989. ACM Press, New York, New York.

[8] Karel Driesen and Urs Hölzle. Minimizing Row Displacement Dispatch Tables. In *Proceedings of the 1995 ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 141–155, Austin, Texas, October 1995. ACM Press, New York, New York.

[9] Karel Driesen, Urs Hölzle, and Jan Vitek. Message dispatch on pipelined processors. In *European Conference on Object-Oriented Programming*, August 1995.

[10] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[11] Mary F. Fernandez. Simple and effective link-time optimization of modula-3 programs. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 103–115, June 1995.

[12] International Business Machines INC. *The PowerPC Architecture: A Specification for a new Family of RISC Processors.* Morgan Kaufmann Publishers, INC., 1994. (Second Edition).

[13] Stanley B. Lippman. *C++ Primer.* Addison-Wesley, 1993.

[14] A. C. Myers. Bidirectional object layout for separate compilation. In *Proceedings of the 1995 ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 124–139, Austin, Tx, October 1995. ACM Press, New York, New York.

[15] Ravi Nair. Performance evaluation of C++ applications on the IBM RS/6000. Unpublished IBM Confidential Report, November 1994.

[16] Hemant D. Pande and Barbara G. Ryder. Static type determination for C++. In *Proceedings of the Sixth Usenix C++ Technical Conference*, pages 85–97, April 1994.

[17] J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of the 1994 ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 324–340, Portland, OR, October 1994. ACM Press, New York, New York.

[18] W. Pugh and G. Weddell. Two-directional record layout for multiple inheritance. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 85–91, White Plains, NY, June 1990. ACM Press, New York, New York.

[19] J. R. Rose. Fast dispatch mechanisms for stock hardware. In *Proceedings of the 1988 ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 27–35, San Diego, CA, October 1988. ACM Press, New York, New York.

[20] Bjarne Stroustrup. Multiple inheritance for C++. In *Proceedings of the Spring 1987 European Unix Systems Users's Group Conference*, Helsinki, May 1987.