Dynamic Dispatch in Object-Oriented Languages

Scott Milton* and Heinz W. Schmidt†

TR-CS-94-02

(scott@cs.anu.edu.au, hws@csis.dit.csiro.au)

January 1994

Abstract

Dynamic binding in object-oriented languages is perhaps the most important semantic aspect of these languages. At the same time it can contribute to inefficiency and lack of robustness because it incurs lookup overheads on function calls and hinders the compiler determining the exact type of objects held in variables or returned by functions. This may, for instance, preclude inlining of small functions or attribute offset computation at compile time. Yet attribute accesses are the most frequently executed operations. As a result, to regain lost performance, OO programmers are tempted to break the encapsulation of classes or want explicit control over dynamic dispatch, trading off extensibility.

In the implementation of parallel object-oriented languages the additional complication arises that object accesses may require more expensive remote memory accesses. Lookup at the call may be inappropriate if the code has to be executed on a different processor and there perhaps has a different address.

This paper summarizes dispatching as addressed in several modern object-oriented languages. We then describe and benchmark fast and flexible dispatch schemes that we are currently implementing on SPARC based workstations and multi-processors. These involve elements of C++ virtual function tables and Eiffel's and Sather's ability to redefine abstract functions as attributes. Initial benchmarks seem to promise improved efficiency on a range of modern RISC based architectures.

Categories and Subject Descriptors: D.1.1 [**Programming Techniques**]: object-oriented programming; D.3.2 [**Programming languages**]: Language Classification – *object-oriented languages*, C++, Eiffel, Sather; D.3.3 [**Programming languages**]: Language Constructs – *dynamic binding*; D.3.4 [**Programming languages**]: Processors – *compilers, optimization*; E.1 [**Data**]: Data Structures – *tables* ;

General Terms: Algorithms, Design, Languages.

Additional Keywords and Phrases: benchmarks, dynamic dispatch, efficiency, object-oriented runtime system, multiple inheritance.

*Department of Computer Science, The Australian National University, Canberra
†CSIRO – Division of Information Technology, Canberra

1 Introduction

Dynamic binding of function names to function definitions (or code entry points) is one of the central features of object-oriented languages. It contributes to flexibility and extensibility because it allows new code to be called from unaltered old code by passing instances of new types to old functions or assigning them to old variables. Conceptually, the objects themselves respond to calls. Function binding is at their discretion (in object-based languages) or at the discretion of their classes (in class-based languages). We call functions and attributes collectively *features*. Features are *abstract* if they are polymorphic, i.e., redefinable for different types of objects. In some languages all features are abstract, in others one has to distinguish them upfront, i.e., before any redefinition is permitted or dynamic dispatch is supported. For instance, in C++ one uses the *virtual* keyword [27] to this end, or in CLOS, one declares functions as *generic* [1, 25, 10].

The mechanism implementing dynamic binding is called *dynamic dispatch*. Typically it uses runtime type information to lookup, or bind to, the proper function. Systems like the Lisp-machine operating system with about 2000 classes or Smalltalk systems with many hundreds of classes rely on this mechanism to make prototyping and incremental compilation fast, and systems in operation robust in the presence of adhoc extensions.

There is a tension between dynamic dispatch and static typing. The flexibility resulting from dynamic dispatch goes directly against the safety and increased compile time knowledge of data types needed for early error detection and for the best optimisation. There are inherent tradeoffs in how much checking is possible at compile time versus how much must be done at runtime to guarantee that, for instance, optimiser assumptions about typing are preserved during execution. The dynamic lookup is a major obstacle to optimisation from two different viewpoints: first, the lookup time has to be paid in addition to the function call; secondly, since different functions may be invoked by the same call in the course of execution, small functions can no longer be inlined, and attribute offsets cannot be computed statically, in general. Yet attribute accesses are the most frequently executed operations.

The inlining problem can also cause intra-procedural optimisations to be deferred that would only be possible after inlining in the context of an expanded function body. This is particularly relevant with loops in OO programs when a generic iteration method is encapsulated with a class and a caller drives these iterations in a loop outside the corresponding class.

For efficiency reasons, OO programmers are then tempted to break the encapsulation and/or demand more explicit control over dispatch while giving away extensibility. Efficiency is critical for the wide use of any language. We are particularly interested in efficient dispatch mechanisms for our experimental high-performance Sather [20, 21].

The implementation of efficient dynamic dispatch has been the subject of several papers. For instance Rose [19] compares various low-level interrupt-based dispatching on stock hardware that makes single dispatch as fast as function calls at the cost of portability. Some implementations such as CLOS [11] and Sather [17, 18, 22] use hashing while others such as C++ and Eiffel [15] use lookup tables. CLOS calls a *generic* function which encapsulates the lookup, while most other languages generate special call instructions. Sather and C++ [27] currently employ very different dynamic dispatching mechanisms. C++ objects have embedded pointers to function lookup tables to allow shared code (inherited functions) to find possibly redefined functions in subclasses. Sather uses object type tags and function keys to hash for function pointers.

On top of the dispatching mechanisms various optimisations are used. A foremost goal is to avoid dynamic dispatch wherever possible. Languages like C++ and Sather restrict prototyping flexibility to help the programmer and compiler control the associated cost of dynamic dispatch. In C++ one distinguishes abstract functions (with several different implementations) by the *virtual* keyword when defining them the first time. Sather distinguishes dispatched calls according to the type of objects: the declaration *x:* A, allows *x* to hold only objects of exactly type A, while *x:* $A can hold objects of any subtype. In the former case calls to the object can be resolved at compile time.

In addition, some languages such as CLOS, Self [4] and Sather try to offset the cost of dispatching by other optimisation, too, including so-called *customisation* and *function pointer caching*.

Function pointer caches allow cheap lookups if the function was recently looked-up for the same type of object.

Customisation duplicates and customises inherited code for each subclass. With customisation, calls to *self* can be resolved statically. *Full* customisation, which customises all functions in a given class, has another advantage: exclusively customised code is going to access instances of this class. The class can therefore choose the layout of instances to improve compaction and alignment (which can make an order of magnitude speed difference on RISC architectures such as the DIGITAL Alpha). Furthermore, all attribute accesses to *self* can be hardwired in the code. The downside of full customisation is of course increased code size and longer recompilation times, as changes in inherited functions may affect several subclasses.

In this paper we briefly review different OO dispatching schemes, particularly focusing on C++ and Sather efficiency needs, and complexity resulting from multiple inheritance. We report initial benchmarking results, and propose a new efficient hybrid scheme extending C++ function tables. We discuss this in the context of the Sather language but elements of the scheme can be used with C++, Eiffel or other MI languages.

2 Dynamic Dispatch: Concepts and Implementations

In general terms, the *dispatch problem* can be characterized as follows: *Given a class name c and a feature name f, determine the actual feature location (function pointer, attribute offset)*

$$p = Loc(c, f).$$

A naive implementation encodes Loc by a table of the size $C \times F$ leading to a fast constant time lookup, where C is the number of class names and F the number of feature names. Each column in this table can be viewed as representing an *abstract feature f* that can lookup a function pointer or attribute offset given a class index. Dually, each row represents a function table for the corresponding class c. Unfortunately, for large OO codes these tables are extremely sparse. The space requirements are unacceptable and the tables are not extensible. When new classes or features are added, a large table has to be reorganised, which is unacceptable with incremental compilation. Various dispatch schemes therefore try to take advantage of the most common incremental compilation, viz., adding and compiling individual classes without requiring recomputation of tables for already compiled classes. Moreover they try to take advantage of the inheritance structure to share common portions between related function tables.

Figure 1: Simple memory and table structure for SI

With multiple inheritance (MI) in general one can at most share indexes with one superclass, say the first. For subsequent superclasses one may, like C++, use an attribute offset and table indirection scheme in which portions of a class' instance layout or function table exactly follow the layout in the superclass but are offset in the instance or accessed from the instance via embedded pointers. Code shared with a superclass then works after changing *self* to a pointer to the embedded portion. In C++ a superclass block is preceded by a pointer to a lookup table for that superclass, so shared code of an inherited function finds the proper perhaps redefined functions to call for that subclass. When such an inherited function is called, a pointer to the block can be passed by adding a fixed offset to *self* (*this* in C++). This offset is contained in the function table, paired with the function pointer to the actual function to be called. Figure 2 shows the layout of C++ objects according to the program in Fig. 3.

The same technique can be used if one of these superclass functions needs to invoke a redefined function. The negative offset used in this case has the effect of readjusting the *self* pointer to the original object. A fixed index into that table can be used for all functions of the same name. The space for all the tables is much smaller than $C \times F$. This is because the attribute block and function table for the first superclass can be treated like in the SI implementation. In this case, the pointer does not require to be changed and the superclass table can be embedded directly in the subclass table.

With MI, *inheritance conflicts* between functions of the same name are possible. In what-

Figure 2: Object and virtual function table layout in C++

ever way the inheritance conflict is resolved, the proper function pointer can be stored in the proper field in *all* of the respective superclass tables associated to the class. Just as classes can redefine functions defined in superclasses so too can they redefine attributes. With attribute inheritance conflicts however, it is not a reasonable solution to keep several fields in sync. For function pointers the filling-in of several related fields can be done in compile time. If we wanted to use the same technique for attributes, superclass code that assigned to an attribute would have to assign to the corresponding fields of other superclasses as well, adding to the access cost proper. It is simpler and less costly to map the conflicting attributes, that are meant to serve the same purpose, to just one field, and to lookup an offset both on read and write. C++'s solution is the exclusion of such abstract attributes, which share one field in all subclasses. Instead, attributes from different superclasses are kept disjoint. They access different fields in a subclass' instance. In the case of inheritance conflict (same name in different superclasses), the programmer must explicitly resolve the conflict in the program text. Thus inherited code from superclass `A` cannot be redirected to access attributes from superclass `B`, *abstract attributes* cannot be expressed. At the same time *abstract functions* can.

The dynamic call *c– >Method4()* can be compiled down to:

```
((void(*)())((*((Table**)c+BTblOffInC)+Method4InB)->FunctionPtr))
  ((char *)c + BTblOffInC +
              ((*((Table**)c+BTblOffInC)+Method4InB)->PtrAdjust)))
```

The Common Lisp Object System (CLOS) supports many orthogonal and flexible OO

```
class A                              |     class C:
{                                    |     public A, public B
   int privateAMember;               |     {
 public:                             |        int privateCMember;
   virtual void Method1();           |      public:  v
   virtual void Method2();           |         virtual void Method1();
};                                   |         virtual void Method4();
                                     |         virtual void Method5();
class B                              |         virtual void Method6();
{                                    |     };
   int privateBMember;               |
 public:                             |
   virtual void Method3();           |
   virtual void Method4();           |
};                                   |
...

main(){
   B *b = new B;
   B *c = new C;
   b->Method3();
   c->Method4();
}
```

Figure 3: Virtual functions in C++

elements such as multi-inheritance, multi-dispatch and method combination. Its dispatch mechanism supports various caching and optimisation schemes [11] to make up for the inherent resulting complexity. In CLOS abstract functions are objects, so-called *generic* functions, which encapsulate (hash) tables mapping from class tags (pointers to so-called class wrappers) to methods. The hash functions are chosen to balance development efficiency and runtime efficiency, based on data collected from several large CLOS applications. The data also justifies separating and optimising the dispatch used with the following categories of generic functions:

reader-only automatically generated attribute readers;

writer-only automatically generated attribute writers;

1-effective exactly one method is ever called (dynamically);

n-effective more than one method is called;

multi-argument it dispatches on more than one argument;

Of these categories, reader-only and n-effective are called most frequently. CLOS' dispatch mechanism supports caching. Moreover it uses various special caches that bypass standard

caching for special cases according to the above classification, particularly 'one argument' dispatch (attribute access) and 'two arguments' dispatch (statistically common enough to deserve special treatment).

Eiffel allows the redefinition of *abstract functions* as attributes. Like C++ the language allows single-argument dispatch only. One argument is syntactically distinguished in the *dot* notation *o.foo(arg1,...)* where the function *foo* is called on object *o* with arguments *arg1* etc. "Dispatched" attribute accesses are implemented by indirect addressing using a looked up offset, while dispatched calls look up a function pointer. Older Eiffel implementations use a $(C \times F)$ function table in which function pointers are accessed by class index plus function offset. To avoid the required size for a large number of classes (> 1000), in more recent implementations, this table is squeezed considerably by actually adding a function offset that is a function of the class index. This is implemented by another table lookup. No caching is used to our knowledge.

Sather generates C code and gives programmers control over types not requiring dynamic dispatch, such as basic types or constructor types, i.e., types of objects that do not have subtypes. While Sather 1 allows abstract function to be redefined as attributes, the current public domain implementation of Sather (v0.5) does not, to be able to use faster dispatching for the more frequently used attributes. Each attribute and function name is assigned a unique integer value. A hash table hashes on that integer and a type tag embedded in each object. The contents of the table are interpreted as a function pointer for function calls, as a pointer to a static storage location for access to shared class variables, as an immediate value for constants, and as an object offset for public attribute access. Only those features that are accessed in a dispatched fashion are put into the table. The table itself uses an inexpensive hashing function and collision resolution mechanism and is kept at a load factor of .5 for efficiency.

Measured $N * 1,000,000$ iterations	w/	w/o
Pure loop cost (placebo)	0.15	–
Normal attribute access	0.21	0.06
Normal (C) function call	0.28	0.13
Abstract attribute access via customised call	0.49	0.34
Abstract attribute access via dynamic dispatch call	0.59	0.44
C++ dynamic dispatch	0.56	0.41
Sather 0.5 dynamic dispatch (cache hit)	0.69	0.54
Sather 0.5 dynamic dispatch (cache miss)	7.07	6.92

Table 1: Raw runtime costs in existing systems (μs Sparc 2)

Sather uses a caching scheme in connection with dynamic dispatch. A static cache is associated to every call for caching the type tag and function pointer on lookup. On the next call around if the type of the object is the same, the cached function pointer is used and the lookup avoided. As long as the type of the objects stays the same on repeated calls, the lookup can be avoided. This is a considerable speed up particularly in the most expensive parts of the code, viz. loops on homogeneous lists and arrays, which dominate many benchmark programs [5]. In this case the dispatching overhead is only one extra comparison. However on heterogeneous collections the performance degrades considerably due to an expensive lookup

scheme [30]. The Karlsruhe Sather implementation uses compressed function tables to achieve a much better lookup time [13].

The raw times needed for various elementary runtime operations as found in some existing systems are shown in Table 1 with and without loop control overheads. Our method of measurement involves several millions of iterations including loop control and array access cost of about 0.15μs per iteration. By inspecting the generated assembler, we verified that this "placebo" cost is measured by a skeleton loop whose optimisation is identical to that of the "true" loops' skeletons. Different arrangements of the arrays allow us to guarantee cache hits or misses for Sather. The times were measured on a SUN SPARCstation 2 using GNU g++ and Sather 0.5 with GNU gcc. Other members of our group at DIT are also working on comparing these and a few public domain OO systems on Mips and Alpha architectures using native C++ compilers and Sather with a native C compiler[1].

3 Customisation Tradeoffs

Customisation is an optimisation technique first used in Self [4, 29]. Customised code runs exclusively on instances of one particular subclass. In customised code the type of *self* is known at compile time. All attribute offsets and *self* calls can therefore be resolved at compile time, *self* calls to small functions can be inlined. As mentioned above, fully customised classes can reorder attribute fields and function tables at their discretion because no shared superclass function is ever operating on the instance. In particular, fields can be permuted for optimal compaction and optimal data alignment. For example, on the Alpha architecture this can make a factors 3-10 performance difference [6, 24, 23]. Moreover, customised classes can avoid embedded pointers to function lookup tables for superclass code, which saves additional space. While customisation drastically reduces the number of dispatched calls, its price is code duplication. As customised code cannot run on subclass instances, subclasses need their own version of inherited code. For the current Sather implementation, for instance, which customises *all* classes fully, the resulting code duplication can lead to considerable recompilation time when only a single function definition is changed, because many classes inheriting that definition may have to be recompiled.

The lookup table approach of C++ has the advantage of reduced code size and faster raw dispatching. Clearly, customisation versus sharing is a time vs. space tradeoff and a compile time vs. runtime tradeoff, inherently affecting the dispatch mechanism and its amortised cost. C++ uses pointer adjustment to solve the attribute access problem, so that the attributes are always a known offset from the *self* pointer. The *self* dispatching must be dynamic, but the pointer adjustment allows a fast dispatching mechanism, so the overhead is tolerable.

4 Fast Hybrid Dynamic Lookup

We combine the above mechanisms to an efficient hybrid dispatching that allows multiply inherited attributes to be shared as well as functions. Moreover it can cope with abstract functions being redefined as attributes or vice versa. This is important for supporting incremental compilers and interpreters, which need to minimise the prototyping turnaround time, or allow users to add debug or profile wrappers to attribute accesses.

[1] In the final paper we may be able to include some of these results.

Abstract attributes can be implemented by regarding each attribute as an index into a table, which contains the proper offset of the object's field holding the attribute value. In fact we can use a lookup mechanism similar to abstract functions if we store a further flag that indicates whether the feature is an attribute or a function, allowing attributes to be reimplemented as functions in subclasses and vice versa. An example illustrates this.

class A **is attr** hop, skip, jump: INT **end**;
class B **is attr** walk, skip, run: INT **end**;
class C **is** A; B;
        attr dance: INT **end**;
class D **is** C;
        skip: INT **is end end**;
a: $A := A::new;
b: $B := B::new;
c: $C := C::new;
d: $D := D::new;

We want *skip* to be shared between the A and B parts of a C object. A possible memory allocation is given in Fig. 4. The table entries here are assumed to have three subentries - *(PtrAdjust, IsAttribute, Field).* - *PtrAdjust* is a pointer modifier, *IsAttribute* discriminates attributes and functions, and *Field* is either an attribute offset (*FieldOff*) into the object or a function pointer (*FieldFPtr*) implementing the abstract function for this class. An expression *c.run* can be evaluated by the C code:

```
((*((Table**)c+BTblOffInC)+RunInB)->IsAttribute)
?*((int *)((char*)c+((*((Table**)c+BTblOffInC)+RunInB)->FieldOff)))
:((int(*)())((*((Table**)c+BTblOffInC)+RunInB)->FieldFPtr))
        ((char *)c + BTblOffInC*sizeof(char *) +
                    ((*((Table**)c+BTblOffInC)+RunInB)->PtrAdjust)))
```

We first check to see if the feature is an attribute or a function. If it is an attribute, then the offset can be fetched from the lookup table. If it is a function, then the dispatched call can be made. This strategy fundamentally changes the meaning of the *self* pointer, which can now be thought of as a pointer to the list of lookup table pointers and as an address which to add the fields offsets in the lookup tables.

Shared superclasses, i.e. the case where a class indirectly inherits from another twice, are even easier to handle than in the normal C++ case (virtual base classes). Consider the following code and its layout in Fig. 5.

class A **is attr** hop: INT **end**;
class B **is** A; **attr** skip: INT **end**;
class C **is** A; **attr** jump: INT **end**;
class D **is** B; C; **attr** dance: INT **end**;

Allocation, and placement of attributes in the objects is flexible, if the layout information for the object is stored in the lookup tables. Note that we are free to change the meaning of the *self* pointer, as we can adjust for this by means of field offsets in the table. We make use of this in our experimental implementation when we combine customisation and lookup tables by adding table pointers at the end of compacted object layouts.

Figure 4: Memory and table structure for shared feature lookup

4.1 Access Optimisation and Adjust-And-Call

Above we assumed total code reuse, and used no knowledge about whether a piece of code will have to run on subclass instances or whether a subclass will redefine attributes and functions. This knowledge however can produce significantly optimised code. There is a tension here though between efficiency and flexibility. The object-oriented ideal is that programmers

Figure 5: Shared superclass table layout

write classes in the most reusable way. They cannot, in general, know in advance how the classes are used. It is impossible to design or to encode them in the most efficient way for these future uses. C++ forces the programmer to anticipate which methods are to be redeclared, and hence which should be declared *virtual*. No provision is made for a subclass to redefine the attributes of an object. Arguably the task is better left to the compiler. In a *closed compilation* approach, a compiler can assume it knows all classes and distinguish where abstract attributes are actually used and produce optimal code accordingly. If future classes change these assumptions, at most recompilation of some classes is required but code does not require much more expensive changes. We can treat the following categories in a special way by optimising the generated access code presented above:

- If an attribute is known to never be redefined as a function, then code to check whether it is an attribute or function can be eliminated, saving time and space. Likewise if a function is known to never be redefined as an attribute.
- Non-abstract attributes can be allocated in the object statically, that is, with a known offset from the base pointer. This will save consultation of the lookup table each time

the attribute is accessed. Note that this will involve no longer having all of the table pointers at the start of the object - and the complexity, but not the inefficiency, of the C code above will be slightly increased.

- Even in the presence of attribute inheritance conflict, if the attribute is known never to be redefined as a function, it can be left in place for one of the multiple inheritance cases, and the table lookup can be avoided - for example skip in the A object in the first example can be given a constant offset, and the B objects can have their relative position of skip redefined when they are parts of the C or D objects to suit the A class.
- Constant optimisation at the C compiler stage should eliminate many of the constant additions which are likely to be generated by a naive compiler using the above coding scheme.

In general, the C++ scheme requires an adjustment to the *this* pointer as the dynamic dispatch is made. This is because the function being called expects the attributes to be at fixed, and known offsets from the *this* pointer. While clearly unnecessary in the single inheritance case, it seems to be a necessary evil of multiple inheritance. There are however, ways in which this can be avoided. Two pointers of different type which point to the same object may have different values, as they have to point to different lookup table pointers. Customisation consists of re-producing code for each inherited function, even if it has not been re-defined. The code can be customised to expect the pointer to be the pointer to the lookup table pointer it was called with. When there is an inheritance conflict it will require multiple customisations even for the same class. Consider:

```
class AA is xx: INT end;
class BB is attr yy: INT;
        rr is xx := 4 end end;
class CC is AA; BB end;
cc: $CC := CC::new;
bb: $BB := cc;
cc.rr;
bb.rr;
```

If the C++ scheme is used, then the cc and bb pointers have different values, even though they point to the same object. The customised code for rr can be constructed as to expect the *self* pointer to point to the lookup table pointer for the BB part. Clearly this can be done with no loss of efficiency within the function, and with the elimination of one lookup and addition on the invocation.

When there is an inheritance conflict, this mechanism fails. One workaround involves generating the function several times, for each possible offset, though if the generated code is very large, then replicating it will be very expensive. The situation can be salvaged however, if we generate the function once, and let all of the other copies simply employ an *adjust-and-call mechanism*, which adjust the pointer, and calls the first function. Our experiments to date suggest that the adjust-and-call written in assembly code is faster than the C code implementing the table lookup. We have summarised preliminary timings in Table 2.

4.2 Assembly Call Tables

As any assembly code to do an adjust-and-call will be of a fixed and known size, it is possible to do away with the lookup tables altogether. The lookup table pointers can instead point to arrays of fixed size adjust-and-call assembly routines. Dispatching becomes a branch into the table. The instructions in the table then call the proper function.

A variation of this assembly adjust-and-call table is also likely to support attribute accesses efficiently on modern RISC architectures, with low function call overhead. For this purpose the entries in the lookup table contain attribute accessor code. In our experiments with on SPARC hardware, we have been able to do this in just two instructions, so the size of the lookup tables is unchanged from the traditional C++; attributes and functions are treated on an equal footing; and the overhead in checking whether the feature is an attribute or a function is saved. The following SPARC instruction sequence shows a portion of an assembly "lookup table". The first entry is a call, and the second is an attribute read.

```
_CodeVtbl_B_In_C:
        b       _B_ACall        ! 1st entry: branch to ACall, but first ..
        sub     %o0,8,%o0       !            adjust 'self'
        retl                    ! 2nd entry: return, but first ..
        ld      [%o0+8],%o0     !            get the field value
```

While the generation of assembly code reduces the portability of the system, the code can be trivially constructed, and will be very similar for most architectures. Other fallbacks are also easy to construct. C preprocessing macros can be redefined to revert to a non-assembly code solution.

4.3 Benchmarks

We have produced code for the cases described above, and the timing results are summarised[2] in Table 2.

As expected, the raw dynamic dispatch mechanism employed by C++ is far more time efficient than that of Sather. In fact, it is so efficient there is little point in the Sather style caching mechanism when this dispatching mechanism is used. The overheads swamp the benefits even when there is a cache hit. We also find that the need to check if a feature is a function or an attribute is expensive, and is best avoided.

The code table dispatch mechanism is faster than the other dispatch mechanisms tried. While the attribute access via this mechanism is slower than the static attribute access, the performance is not so slow as to reject the mechanism as a default attribute access particularly as it offers abstract attribute functionality without programmer anticipation or expensive and complex closed compiler optimisation.

5 Implementation

We have modified the Sather 0.5 implementation to use the base mechanism described. Using a hidden feature allowing overriding functions in existing classes, few changes were required and the changes were surprisingly limited and local. Unlike C++ however, our scheme employs

[2]For simplicity of comparison we repeat the basic cost of existing runtime implementations of Table 1

Attribute accesses	
Normal attribute access	0.06
Abstract attribute access known to be attribute	0.21
Abstract attribute access known to be either call or attr	0.47
Dynamic dispatch with assembly code lookup tables (attr)	0.29
Function calls	
Normal (C) function call	0.13
C++ dynamic dispatch with no pointer adjustment	0.31
C++ dynamic dispatch	0.41
C++ dynamic dispatch with adjust-and-call	0.46
Abstract call known to be either call or attr	0.74
Function calls with Sather 0.5 caching	
Cache hit	0.54
Cache miss then C++ dynamic dispatch	1.20
Cache miss then Sather 0.5 dynamic dispatch	6.91
Assembler adjust-and-call	
C++ dynamic dispatch with assembly adjust-and-call	0.36
Dynamic dispatch with assembly code lookup tables (call)	0.31

Table 2: Raw runtime costs hybrid schemes (μs Sparc 2)

Benchmark	Unoptimimised			Optimimised		
	Sa 0.5	C++	Sa 0.6	Sa 0.5	C++	Sa 0.6
Colorado	16.8	8.34	11.0	14.2	3.00	3.50
Stanford	55.5	40.0	55.4	25.5	21.3	25.7

Table 3: Standard Benchmarks - Total CPU Use (sec) Sparc 2

the full customisation of the standard Sather. In theory, such a system could run faster than C++, because no dispatching needs to be done when an object is dispatching on itself, because the type of the object is known. The system will however suffer the overhead of extra code size, which may result in more paging.

Optimisations are not yet complete, and the implementation still suffers some unnecessary overheads of the older version, but the results are still an interesting indication of potential. The tables 3 and 4 are comparative execution times for benchmarks performed on a SPARC-station 2. Table 3 summarises the results obtained for our modified Sather (v0.6) for both the Colorado dispatch benchmark [30], and for the Stanford integer benchmark.

It should be noted that the Stanford integer benchmark does not require dynamic dispatch. Its C++ code uses standard C function calls explicitly. The slight variation between Sather 0.5 and 0.6 both in the unoptimised and optimised runs appears insignificant. The Stanford integer benchmarks are "micro benchmarks". Portions of these may run entirely out of the RISC processors' caches, depending on the given cache size. It is still interesting to compare these, as some language implementations fall back on these benchmarks considerably. We use

them mainly as a coarse indicator of "competitive" target C control and integer coding.

We have also benchmarked Sather 0.5, 0.6 and K (the University Karlsruhe Sather implementation) and C++ measuring the times required for the different critical accesses and dispatches discussed and integrated in the new Sather 0.6 coder so far[3]. These times are outlined in Table 4. Note that the Karlsruhe Sather implementation is based on the compiler construction toolkit *Cocktail* [7, 8] with a backend generating Sparc assembler code. The currently available backend does not yet include optimisations.

Benchmark	Unoptimimised				Optimimised		
	Sa 0.5	C++	Sa 0.6	Sa K	Sa 0.5	C++	Sa 0.6
Dynamic dispatch	10.97	1.35	1.74	1.57	9.05	0.36	0.55
Double dispatch	12.46	1.96	2.04	1.74	9.23	0.74	0.71
Normal call	0.98	0.66	0.95	0.79	0.22	0.08	0.22
Dynamic attr read	9.33	1.64	1.44	1.65	8.03	0.41	0.34
Dynamic attr write	9.09	1.63	1.25	1.67	8.03	0.42	0.45
Static attr read	0.87	0.72	0.88	0.76	0.13	0.05	0.13
Static attr write	0.88	0.69	0.88	0.87	0.32	0.10	0.28

Table 4: Current Implementations' Cost Comparison (μs Sparc 2)

The times displayed in Table 4 represent the following measurements.

- *Dynamic Dispatch* A normal dispatched call to a function. The benchmark guarantees that the Sather 0.5 caching mechanism will always miss.
- *Double Dispatch* A dispatched call to a function that follows by a *self*-call. This is placed here to demonstrate the usefulness of customisation. Sather 0.6 makes only one dynamic and one static call in this case.
- *Normal Call* A call to a function where the called function is known at compile time and no runtime lookup is needed. Sather does this through monomorphic typing. C++ must use non-virtual routines to achieve a similar effect, which requires programmer anticipation of eventual class use.
- *Dynamic Attribute Read* Reading an attribute where the type is not known at compile time and where conflicts are possible. C++ must handle this case through a dynamic dispatch call to an accessor function.
- *Dynamic Attr Write* Writing the attribute above. Once again, C++ must do this through a dynamic dispatch to an accessor function.
- *Static Attr Read* Reading an attribute where the type is known at compile time. These are equivalent to an access of a record field.
- *Static Attr Write* Writing the attribute above.

[3]Table legend: Sa = Sather

As expected, our modified Sather 0.6 performs better than the Sather 0.5 in most cases, and close to the performance of C++ in all cases. It was expected that Sather 0.6 would perform as well as C++ in the normal dispatch benchmarks, but the loop control code as produced by the current Sather version is not as efficient as that generated by the C++ compiler. The modified Sather outperformed C++ in the case where it could avoid a dynamic dispatch call, and where C++ had to make a call to access an abstract attribute – where Sather handles this with the standard dispatch mechanisms.

These are only first benchmark results and mostly raw times. Although overall performance on larger and more application-specific benchmarks can look differently, clearly with raw times of Sather so close to those of C++ the overall performance can get close enough. There is some hope that the cleaner semantics of Sather and Eiffel would allow simple and yet effective optimisations, and that Sather still gives programmers more explicit control about costs than Eiffel. But this is a subject for a subsequent paper. We are currently refining our benchmarks to validate our conjectures on the interplay of dispatch with customisation, inlining, and hardware caching and to understand high-performance trade-offs of our scheme with our experimental multi-processor version of Sather [20, 21] in the presence of local and remote object accesses.

In general we expect that for an expensive raw dispatch mechanism, customisation is much more important than for a very cheap one, because it avoids dispatches. However, as the dispatch overhead gets smaller, we expect that customisation creates more and more penalties, because it increases the size of code blocks and thus the probability of hardware cache misses and page faults. Its advantage regarding more inlining potential may be achievable by much more selective customisation or other refinements of the dispatch mechanisms. Sather 0.5 is better off with customisation, but it is not clear that this is still the case with the modified Sather 0.6.

While dispatches are offline branches, in general, branches into an assembly code table would add to the number of such offline branches. We expect this to cause penalties on new RISC architectures. For instance on the Alpha such branches can mean a penalty of several tens of cycles as the instruction pipeline is drained and potential instruction cache misses are incurred. It will be interesting to see whether this could be offset by some sort of soft caching mechanism with the proper adjust-and-call instruction sequences cached or whether the lookup table mechanism is a winner in this context too.

6 Conclusion

Dynamic dispatch is a major overhead in many OO languages, and it is clear that efficient mechanisms are critical to the performances of these languages. The issues are complex and the solutions offer many trade-offs between complexity, speed, code size, and flexibility.

We have proposed and benchmarked several hybrid solutions which solve many, but not all of the problems. We have shown that the fast dynamic dispatch mechanism employed by C++ can be extended to allow an implementation for abstract attributes, and the redefining of attributes as functions and vice versa, such as required by Eiffel 3 and Sather 1. A two instruction assembly code table realises an 'adjust-and-call' dispatch mechanism as a call table. It is slightly less portable but considerably faster according to our measurements. None of the mechanisms proposed precludes further optimisations by good optimising OO compilers (or target code C compilers).

It is possible to use the Sather 0.6 dispatch mechanism with shared code. Such a system would have to do a dynamic lookup for each abstract attribute in the current object as well as for function dispatch. Likewise the C++ dispatching mechanism could be used with completely customised classes. Shared code has the advantage that all code for library classes can be precompiled and precompiled inherited code runs as is on subclass instances. This is likely to reduce the recompilation time, and therefore the development time spent in the compile-link-run cycle. We find however, that an adoption of the feature table lookup scheme removes some of the benefits of customisation, and the employed dispatching mechanism may change the optimum.

Our efforts to date have improved the speed of Sather almost to the speed of C++, still without sacrificing, at all, the benefits offered by the Sather language, including safe down-casting and better compaction. These mechanisms work well with shared inherited target code or customised target code for each inheriting subclass. This will be beneficial for incremental compilation and mixed compiled and interpreted code, with a range of options between fast compile time vs. fast runtime.

We have chosen Sather for its clean and simple semantics and the public availability of a runtime system and compiler written mainly in Sather. The results, however, are not limited to Sather but are applicable to future versions of C++, Eiffel or other MI languages. We are currently extending these schemes to work with migrating objects in a parallel environment.

While there are no panaceas, the results are encouraging.

Acknowledgement

Thanks particularly to Stuart Hungerford, and to Oscar Bosman and Richard Walker, for interesting discussions on a prelimary dispatch scheme, and the interpretation of benchmark results.

References

[1] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. Keene, G. Kiczales, and D. A. Moon: *Common Lisp Object System Specification, X3J13 Document 88-002R.* (Also published in *SIGPLAN Notices,* **23,** special issue, Sept. 1988, and in Guy Steele: *Common Lisp, The Language, 2nd ed.,* Digital Press, 1990.)

[2] H. Boehm, and M. Weiser: "Garbage Collection in an Uncooperative Environment". *Software Practice & Experience,* September 1988, pp. 807–820.

[3] L. Cardelli: "Basic polymorphic type checking", *Science of Computer Programming* **8,** pp. 147–172 (1987)

[4] C. Chambers, D. Ungar and E. Lee: "An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes." *OOPSLA 89 Conf. Proc., Special Issue, SIGPLAN NOTICES,* **24**(10), 1989

[5] C.-C. Lim and A. Stolcke: *Sather Language Design and Performance Evaluation.* TR-91-034, ICSI, 1991

[6] DIGITAL: *Alpha Architecture Handbook.* DIGITAL Equipment Corp., 1992

[7] J. Grosch and H. Emmelmann: "A Tool Box for Compiler Construction", *LNCS 477,* Springer Verlag, pp. 106–116, 1990

[8] J. Grosch: "Transformation of Attributed Trees Using Pattern Matching", *LNCS 641,* Springer Verlag, pp. 1–15, 1992

[9] U. Hoelzle, C. Chambers, D. Ungar: "Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches", in O. Nierstrasz (ed.): *Proc. ECOOP 91,* 1991

[10] S. Keene, "Multiple Inheritance in CLOS". *JOOP* **2**(5), pp. 75–77 (1990)

[11] G. Kiczales and L.H. Rodriguez Jr.: "Efficient Method Dispatch in PCL", in A. Paepcke (ed.): *Object-Oriented Programming: The CLOS Perspective.* MIT Press, 1993.

[12] B. Liskov and J. Guttag: *Abstraction and Specification in Program Development.* New York (NY): McGraw-Hill, 1986.

[13] R. Mauch and M. Trapp: *Ein Übersetzer und Laufzeitsystem für die objectorientierte Sprache Sather.* Diplomarbeit, Univ. Karlsruhe, 1992

[14] B. Meyer, *Object-oriented Software Construction.* Prentice Hall, 1988

[15] B. Meyer: *Eiffel: The Language.* Prentice-Hall, 1992

[16] D. A. Moon: "The Common Lisp Object-Oriented Programming Language Standard." in *W. Kim and F. Lochovsky (eds.): Object-Oriented Concepts. Applications, and Databases,* Addison-Wesley, 1988

[17] S. M. Omohundro, *The Sather Language.* ICSI, Berkeley, 1990.

[18] S. M. Omohundro, *The Sather Programming Language.* Dr. Dobbs, 10/93, pp. 42–48, 1993 ICSI, Berkeley, 1993.

[19] J. R. Rose: "Fast Dispatch Mechanisms for Stock Hardware", *SIGPLAN Notices* **22**(2) pp. 85–94, 1987 also: *OOPSLA 88 Proc.* pp. 27–38, 1988

[20] H. W. Schmidt: "Towards Data-Parallel Object-Oriented Programming", *Proc. of the Third Workshop on Compilers for Parallel Computers.* Vienna (AU): Univ. Vienna, 1992

[21] H. W. Schmidt: "Data-Parallel Object-Oriented Programming", *Proc. of the 5-th Australian Supercomputer Conf.* Melbourne (AUS): Univ. Melbourne, 1992

[22] H.W. Schmidt and S.M. Omohundro, "CLOS, Eiffel and Sather: A Comparison". in A. Paepcke (ed.): *Object-Oriented Programming: The CLOS Perspective* MIT Press, 1993.

[23] R.L. Sites, A. Chernoff, M.B. Kirk, M.P. Marks and S.G. Robinson: *Binary Translation* CACM **36**(2), pp. 69–81, 1993

[24] R.L. Sites *Alpha AXP Architecture* CACM **36**(2), pp. 33–44, 1993

[25] G. L. Steele Jr., *Common Lisp – The Language.* Digital Press, 1990

[26] Guy L. Steele Jr.: *A Philosophical Overview of the CM-5.* Thinking Machines Corporation, 1992.

[27] B. Stroustrup: *The C++ Programming Language.* Addison-Wesley, 1989.

[28] B. Stroustrup: *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[29] D. Ungar, R.B. Smith, C. Chambers, and U. Hölzle: *Object, Message, and Performance: How They Coexist in Self.* IEEE Computer, **10,** 1992.

[30] Henderson, Zorn: *Comparison of modern object-oriented languages.* TR CU-CS-641-93, Colarado Uni., 1993