Dynamically Extensible Objects in a Class-Based Language

René W. Schmidt

Department of Computer Science
University of Aarhus
DK-8000 Aarhus C, Denmark
email: rws@daimi.aau.dk

Abstract

Object-oriented programming techniques allow the construction of reusable and extensible code. However, class-based languages have poor support for implementing type-orthogonal behaviour and extending non-leaf classes, which result in implementations with poor performance or limited functionality. This paper presents dynamic slots, a mechanism for extending objects at runtime. We show how this mechanism can be used to build efficient implementations of type-orthogonal abstractions. Dynamic slots can be statically type-checked and have been integrated into the BETA programming language. Measurements of a persistent store show that they significantly improve performance.

1 Introduction

The object-oriented programming paradigm is generally renowned for its modelling capabilities. Real-world concepts and phenomena can be modelled directly by using classes and objects, and refinement of concepts is supported by subclassing. Structuring functionality in classes makes it possible to implement self-contained and thereby reusable software components. For example, most object-oriented languages come with a container library that implements sets, dictionaries, linked list, etc. This simplifies program development by providing high-level abstractions and already debugged code.

However, organizing a concept into a class or a group of classes cannot always be done cleanly. Consider the design and implementation of object persistence. Object persistence is a concept that allows: (i) objects to be checkpointed to stable storage during the execution of a program, and (ii) to reinstantiate the objects from their checkpointed form into a possible new program execution. Object persistence should ideally be modelled, using one or more classes, in such a way that it is possible to apply persistence to all objects. Modelling an object as persistent by it being of a certain subclass violates the type-orthogonality.

Instead, persistence can be modelled as a runtime association from a *PersistentStore* object to the object to be persistent. Hence, an object is persistent if it is in the transitive closure of the objects associated with the PersistentStore. The problem in doing this is that the PersistentStore implementation needs to associate additional state with the objects it works on, e.g., it needs to maintain a unique object-identifier for each object that is persistent. In a strictly class-based language is it not possible to add extra information to an object at runtime. This can have several negative consequences when implementing type-orthogonal abstractions: (i) unnecessary limitations, such as a class is required to be of a certain (sub)type to be used by a particular abstraction; (ii) unsatisfactory performance, because the extra information must be found in auxiliary data-structures; or (iii) the design of special-purpose programming languages, that include tailored support for a specific feature, e.g., the Emerald [Jul et al. 88] language is designed and implemented with compiler and runtime support for distribution.

Recently, metalevel interfaces have been constructed for several class-based languages, including CLOS [Kiczales et al. 91], Open C++ [Chiba 95], and BETA [Brandt & Schmidt 96], and there has been an increased interest in *Open Implementations* [Kiczales 92]. One of the goals of these approaches is to make it possible to implement type-orthogonal abstractions directly in a programming environment, without relying on implementation details or require changes to the language implementation. Many such abstractions may need to associate additional information to the objects they work on, e.g., object distribution, profiling, and debugging tools. A type-orthogonal facility to extend objects is needed to allow for maximum flexibility and efficiency of metalevel programs.

This paper proposes a mechanism for dynamically extending objects at runtime with additional attributes, called *dynamic slots*[1]. Dynamic slots are orthogonal to the class hierarchy. A dynamic slot can be added to any object without regard to its type. Dynamic slots are statically declared in the source code, hence statically type-checking is possible, but storage is not allocated until their first use.

We have incorporated dynamic slots into the statically-typed and compiled language BETA [Madsen et al. 93]. The implementation required no change to the compiler, and only minor changes to the runtime system. Furthermore, it provides virtually no overhead for existing program, while providing efficient access to and creation of dynamic slots. To evaluate dynamic slots, an existing implementation of a persistent store has been modified to take advantage of dynamic slots. By using dynamic slots we simplified the implementation, and measurements show that the performance was significantly improved.

The rest of the paper is organized as follows. The next section describes dynamic slots in detail, and gives several real-word examples of how they are applicable. Section 3 presents how they can be implemented efficiently in a statically-typed language. Section 4 discusses related strategies. Section 5 evaluates our implementation of dynamic slots for BETA. Finally, Section 6 presents our conclusions.

2 Dynamic Slots

An attribute is declared in its class definition for class-based object-oriented languages. Access to an attribute is allowed in all objects that are instances of that particular class or any subclass. For example, Figure 1 shows a BETA fragment defining two classes `A` and `B`, where `B` is a subclass of `A`. Instances of `A` contains the integer attribute `i`, where as instances of `B` contains the two integer

[1]Slots are similar to instance variables in Smalltalk [Goldberg & Robson 89], and dynamic object references in BETA. We use the term *dynamic slot* to disambiguate them from standard BETA references.

attributes `i` and `j`.

```
A: (# i: @integer; #);
B: A(# j: @integer; #);
```

Figure 1: A simple class hierarchy.

A dynamic slot is an attribute that conceptually exists in all objects and for which storage is only allocated if the dynamic slot is used, i.e., if a value is assigned to it. Conceptually, we can view all classes as having a common superclass, typically called *Object*[2]. Adding behaviour or substance to this common superclass will affect all classes and therefore all objects in the system. Hence, creating a dynamic slot is semantically equivalent to adding a new object reference attribute[3] to class Object. A dynamic slot exists in all objects independently of whether an object was created before or after the dynamic slot was created.

Each dynamic slot is represented as an instance of the `DynSlot` class, shown in Figure 2. It contains three methods: `set`, `get`, and `init`. An object reference is assigned to a dynamic slot by the `set` method. It takes as parameters: an object, `o`, which contains the slot we want to access, and the new value for the slot, `value`. For example, a BETA assignment of the form: `newVal[]->o.mySlot[]` is written as `(o[], newVal[])->mySlot.set` for a dynamic slot[4]. The value of a dynamic slot can be read by using the `get` method, which given an object, returns the value of the slot.

Before a new dynamic slot can be used, it must be initialized with the `init` method. In a sense, `init` is a metaobject method (or class-method) that adds the slot-definition to class Object and initializes the slot to `NONE` for all objects. The following code fragment shows a declaration of a dynamic slot `psSlot` that is of type: reference to a `psInfo` class.

```
psInfo: (# OID: @integer #);
psSlot: @DynSlot(# type::psInfo #);
```

In the example, the virtual `type` parameter is specialized to `psInfo`. This tells the compiler to statically check that a value assigned to `set` is at least a `psInfo` object, and that it can safely assume that a value returned by `get` is at least a `psInfo` object. Thus, no runtime type-checking is needed for a dynamic slot. Dynamic Slots are statically typed, but dynamically allocated.

Notice that it is the ability to have generics that makes it possible to implement the dynamic slot interface using a class and still have static type-checking. In BETA, this is done using a virtual class. In C++ [Stroustrup 93], the template mechanism could be used. The drawback of modelling dynamic slots as a class is that access to dynamic slots and ordinary BETA attributes are different. In a language that supports operation overloading, e.g., C++, this could potentially be avoided. Another approach could be to extend the language definition with a special syntactical construct for dynamic slot declaration. Our approach is similar to LISP property lists [Steele 84], which also uses special access methods.

[2] This is the case for both Smalltalk and BETA.
[3] This is the BETA terminology for a C++ pointer.
[4] In BETA assignment (`->`) goes from the left to the right. The box-operator (`[]`) means object-reference.

```
DynSlot:
  (# type:< Object;                    (* virtual class *)

     set: (#                           (* set method *)
      enter (o: ^Object, value: ^type)
      do ...
      #);

     get: (#                           (* get method *)
      enter (o: ^Object)
      do ...
      exit (value: ^type)
      #);

     init: (# do ... #);               (* init method *)
  #);
```

Figure 2: Declaration of the DynSlot class.

2.1 Examples

To further motivate dynamic slots, we will look at three examples where they are applicable. Based on these examples, we will argue that the kind of runtime extensibility provided by dynamic slots are indeed applicable and useful in a compiled language, and that the same kind of functionality cannot easily be provided at compile/link time.

The first two examples are based on the current implementations of object persistence [Brandt 94] and distribution [Brandt & Madsen 94] for the Mjølner BETA System. The last example describes how dynamic slots make it possible to extend an existing framework in a way that is not possible by using subclassing alone.

Persistent Storage

The persistence model for BETA is based on reachability. A set of objects are registered with the persistent object store (PStore) as persistent roots. When the *checkpoint* operation is invoked on the PStore, the transitive closure of the root objects is serialized to stable storage. The PStore is completely type-orthogonal, i.e., it can serialize any object with no regard to its type.

To make it possible to reconstruct objects anywhere in memory when objects are restored from their serialized form, object references are converted to unique object IDs (OIDs) during serialization. Each object is assigned an OID by the PStore. Converting an object reference into an OID requires a linear scan of a list containing pairs of object references and OIDs. An operation which execution time is proportional to the number of objects that have been serialized. This is because it is, in general, impossible to hash on an object reference in a garbage-collected language, since most garbage collectors move objects around in memory. Also, the OIDs cannot be stored directly in an object, because the object can be of an arbitrary type. The linear scan is a major source of overhead in the current implementation.

By using dynamic slots, the OID for an object can be stored directly with the object, so the linear scan is exchanged with a constant time operation. Thus, the time complexity is asymptotically

```
PStore:
  (# psInfo: (# OID: @integer #);        (* extra information *)
     psSlot: @DynSlot(# type::psInfo #); (* dyn. slot for PStore *)
     next_OID: (# ... #);                (* returns a new OID *)

     (* getOID method *)
     getOID:
       (# info: ^psInfo;
       enter (o: ^Object)
       do o[]->psSlot.get->info[];       (* access dyn. slot of o *)
          (if info[]=NONE then
              &psInfo[]->info[];         (* create new psInfo *)
              next_OID->info.OID;
              (o[],info[])->psSlot.set;  (* assign to dyn. slot *)
          if);
       exit info.OID                     (* return OID *)
       #);

      (* Init method *)
      Init: (# do psSlot.init; #);
   #);
```

Figure 3: Using dynamic slots to store object identifiers.

improved. Figure 3 shows how an OID look-up method (`getOID`) can be implemented with dynamic slots.

The main advantage of using dynamic slots in this situation comes from the fact that objects are extended on demand. Dynamic slots will only be allocated for objects that are persistent. This same property is difficult, if not impossible, to achieve through compile or link time program modification. It is not known at compile-time which objects are persistent, since any object reference could potentially be handed to a PStore. The link-time analysis can determine if the persistent store is linked in, and in that case conservatively extend *all* objects with an OID attribute. Even if the link-time analysis would be able to come up with an accurate minimal set of objects that possible could be persistent during a program execution, it might be more efficient to use dynamic slots. For example, in the case of an interactive program where the persistent store is only used under very rare circumstances. The implementation using link-time analysis will then in general require a larger memory overhead than the implementation using dynamic slots.

Distribution

The distribution library provides transparent access to remote objects, i.e., objects located in another process that possibly exists on another physical machine. Transparency is provided by proxies [Shapiro 86]. A remote object is represented by a proxy object in the caller's address-space, which forwards all calls to the remote object. For the caller, the proxy is indistinguishable from the remote object.

A request is forwarded to the remote object by serializing the method invocation along with its parameters and sending it to the process where the remote object resides. This serialization is similar to the PStore implementation. In addition to an OID, the distribution library also needs to maintain state information that tells if an object is acting as a proxy and which communication channel to use.

In contrast to the PStore where OIDs are only needed for the checkpoint operation, the distribution library needs the information on every call on a remote object. Therefore, a linear scan to locate the extra state information is unacceptable. Instead the current implementation requires a distributed object to be a subclass of the `remoteable` class, that contains the extra state information. In effect, the implementation is trading transparency away for efficiency. Dynamic slots can be used in a similar way in the distribution library, as they were used in the PStore example, thereby removing the transparency limitation, while retaining efficiency.

Extending Frameworks

The last two examples discussed how dynamic slots are practical when implementing type-orthogonal abstractions. This example is more general than the preceeding two. It outlines how dynamic slots can be used to solve an extensibility problem in object-oriented software engineering. We will not argue whether the approach presented consists of good software development practice or not, but we do believe that it solves the problem in a simple manner.

Object-oriented software development is known for its support for extensibility and reuse. New subclasses can be derived to specialize and extend the behaviour of a system. For example, a graphical framework might describe a general shape with the class `Shape`. More specific shapes (e.g., triangles, squares, etc.) can easily be added to the system by subclassing `Shape`.

But what if we want to add some behaviour to all shapes in the framework? Then we have to modify the `Shape` class and recompile the entire toolkit. Unfortunately, this is only possible if the source code for the toolkit is available, which is typically not the case if the framework is supplied by a third-party vendor. Recompilation of an entire toolkit might also not be an option if the current version of the library is frozen.

Again, dynamic slots can be used to provide the extra substance needed in class `Shape` and in all its subclasses. No recompilation is necessary, the extra slots are created at runtime. Of course, using dynamic slots will have a higher runtime overhead than making the change directly in the `Shape` class, but it does provide a fairly efficient and direct solution to extending frameworks without having access to the source code. In the case where the source code is available, it makes it possible to quickly prototype a change without recompiling the entire toolkit.

3 Implementation

Since the main motivation behind dynamic slots is to enhance the performance of certain types of abstractions, an efficient implementation is important. However, of equal importance is that the implementation does not impose significant overhead on ordinary programs. Our requirements to the implementation are:

- Dynamic slots should not have any impact on the performance of applications not using them, i.e., you only pay for what you use.

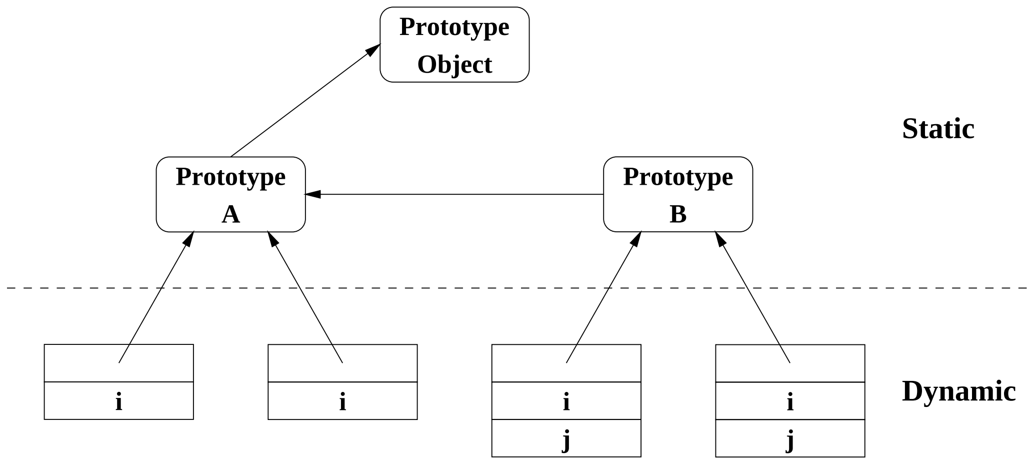

Figure 4: Object layout in BETA.

- Dynamic slots are available on all objects, i.e., they are orthogonal to the type-system. Also, their use does not require special compiler-switches during source code compilation. These requirements ensure that, e.g., persistence is available on all objects, no matter if the code for an object was written with persistence in mind or not. This is especially critical when third-party libraries are used.
- The use of dynamic slots must scale. All dynamic slots are conceptually defined in class Object. Thus, the total number of slots in a system is the number of created slots times the number of objects. The space overhead must be proportional to the number of slots in *use*, not to the conceptual total number of dynamic slots.

Before describing our implementation, we will briefly introduce the runtime organization that our implementation is based on.

3.1 Runtime Organization

We have integrated dynamic slots into the BETA programming language. The object layout used by the Mjølner BETA System is fairly standard for a garbage-collected object-oriented language. A similar implementation can be used for languages such as Eiffel [Meyer 92] and Modula-3 [Nelson 91].

The memory layout is split into two areas. The static part which contains immutable information about each class in a program execution, and a dynamic part which contains the dynamically created objects during program execution. Class information is stored in a structure called a prototype. A prototype contains information that is used for runtime type-checking, locating references in an object, and virtual method look-up. Figure 4 shows the object-layout for two instances of class `A` and two instances of class `B` (from Figure 1). The static area contains prototypes for the classes `A`, `B`, and `Object`. The inheritance hierarchy is encoded in the prototypes by having references to the superclass's prototype.

All objects have a reference to their prototype, which is used both by the garbage collector and by the compiled code. Attributes of an object are allocated in the order declared in the classes, starting with class Object. Notice, that the attribute `i` is located at the same offset in instances of both class `A` and `B`, so the compiler can generate direct memory references to access the attributes of an object.

The BETA garbage-collector is a variant of a generation-based scavenging collector [Ungar 84]. Memory is divided up into two main areas, the *Infant Object Area* (IOA) and the *Adult Object Area* (AOA). New objects are allocated sequentially in IOA. When the IOA area is full, all the live objects are copied from IOA to *To-Space*, and IOA and To-Space swap roles. An object gets promoted to AOA when it has survived a number of IOA collections. In the AOA area most objects are expected to stay alive, so instead of using a copy-collection as for IOA, a mark-and-sweep collection is used.

3.2 Representing Dynamic Slots

The representation of dynamic slots at runtime must be space-efficient, so only slots in use take up space. Secondly, it must be efficient to add a dynamic slot to an object and to look it up. For each object, we store the non-zero slot values in a linked list. The `init` method assigns a unique dynamic-slot ID (DID) to each slot, which is used to identify a slot. The `set` operation adds a new element to the list, or removes an element if the `NONE` value is assigned. The `get` operation scans the list for an element with a particular DID.

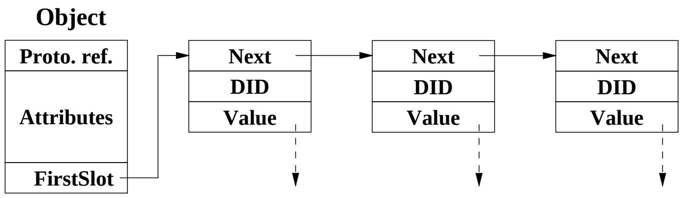

Figure 5: Representation of dynamic slots in an object.

Figure 5 shows an object with three dynamic slots. The object has an attribute, *FirstSlot*, that points to the first dynamic slot. A slot only takes up space if it contains a value that is different from `NONE`, i.e., the memory overhead is proportional to the number of slots in use. Also, a slot can be easily located by following the pointer that is stored in the last word of the object.

Access to a dynamic slot will naturally be slower than for statically allocated attributes, because it requires a traversal of a linked list instead of a direct memory access. We expect only a few slots to be associated with each object, so the overhead of scanning through a linked list will be small. If a large number of slots are in use for individual objects, it might be feasible to store the dynamic slot values in a hash table instead. The dynamic slot implementation could at runtime choose the optimal representation of the slot values to maximize its performance.

The Self system [Hölzle & Ungar 94] allows objects to be dynamically extended during runtime using the `addSlot` method. They allocate a new object with the additional slot, copy the contents of the old object to the new object, and update all pointers to point to the new object. This might require recompilation of code, because attribute offsets can change, but in exchange they provide fast access to dynamically created slots. In contrast, representing dynamic slots as a linked list can be easily integrated into the language implementation outlined above and requires no modifications to the compiler.

Application	Compiler	Sif	Factory
Avg. IOA object size	32b	26b	27b
IOA allocation	1.89Gb	150Mb	373Mb
Avg. AOA object size	46b	65b	35b
AOA allocation	4400Kb	660Kb	2500Kb
AOA to IOA ratio	0.2%	0.4%	0.7%

Table 1: Allocation Statistics.

3.3 Extending Objects

The above representation of dynamic slots requires an extra word for the *FirstSlot* reference in all objects. Unfortunately, a trivial implementation which statically extends all objects with the extra word, will significantly add time and space overhead to a program execution.

The memory behaviour of several BETA programs have previously been studied [Grarup & Seligmann 93] and is summarized in Table 1. The table shows memory statistics for the BETA Compiler compiling itself (Compiler), an interactive editor for BETA source code (Sif), and a discrete-event simulation system (Factory). Based on these numbers, extending each object with an extra word will yield an allocation overhead from 12% to 16%, which for the Compiler means that it will request an additional 240Mb of memory. This increased memory allocation will result in more than 480 additional IOA collections[5], thereby significantly degrade performance.

The ideal solution would be to only extend objects which use dynamic slots with the FirstSlot attribute. This can be achieved by allocating a new prototype for each object that uses dynamic slots and store the FirstSlot reference in the prototype. The prototypes are statically extended with an extra word so it can contain the FirstSlot reference. This avoids changing the size of objects, and since there are considerably fewer prototypes than objects, it is acceptable to increase the size of a prototype[6]. However, the smallest size of a prototype in BETA is 32 bytes, so (in the best case) if more than 12% of the objects use dynamic slots, the memory overhead of this approach will be the same or worse than just adding an extra word to each object. Also, prototypes are not allocated in the garbage-collected heap in the current BETA implementation, so reclamation of prototypes is non-trivial.

Instead we have adopted a solution where the garbage-collector is modified to dynamically extend objects with the extra word at runtime. The worst-case space overhead can then never be worse than statically extending all objects with an extra word. An object in IOA can easily be extended by provoking an IOA garbage-collection, and then extending it when it is copied to To-Space. Objects are lazily extended when the first dynamic slot value is assigned to them. In our implementation, we set a bit in the prototype indicating that an object must be extended, thereby extending all instances of a specific class at a time. This simplifies the implementation, because all objects of a given class have the same size, and it amortizes the cost of the IOA garbage-collection over all objects of a given class.

Extending objects in AOA is non-trivial, because these objects are garbage-collected using a mark

[5]Assuming that the size of the IOA area is 512Kb, which is the default.

[6]To have more than one prototype for the same type of objects complicates runtime type-checking a bit. However, this can easily be fixed with an extra indirection to a canonical prototype.

& sweep algorithm that needs several sweeps of AOA in order to locate live objects and compact storage. Provoking AOA garbage-collections may introduce several disruptive breaks in the program execution, due to its potentially large size. Also, little garbage is likely to be found. To avoid this, we notice that only about 0.2% to 0.7% of all allocated objects are promoted to AOA. Thus, eagerly adding the extra word to all objects that are moved into AOA, will only introduce an insignificant allocation overhead. In fact, based on the numbers from Table 1, an allocation overhead between 0.02% to 0.07%, which in the case of the Compiler translates to a little less than 400Kb. For Sif and Factory, the additional memory overhead is 40Kb and 300Kb, respectively.

We have modified the garbage-collector in the Mjølner BETA System to dynamically extend objects during IOA collections, and to eagerly extend objects when they are promoted to AOA. These changes required only minor modifications to the existing implementation. Extending objects during IOA collections introduces one complication. It is possible that the To-Space runs full during an IOA collection if most objects stay alive and get extended at the same time. We handle this case by promoting the rest of the live objects to AOA.

4 Discussion

The dynamic slot concept presented here for a class-based language has previously materialized itself in a number of other languages and systems, including Self, CLOS, and LISP. Dynamic slots are also related to a number of reflective systems. In the following we will compare and discuss our approach to a number of such systems.

Prototype-based languages use a similar mechanism as dynamic slots to extend objects. In Self objects can be extended with new slots at runtime by using the `addSlot` method. Dynamic slots were strongly inspired by that feature. However, adding dynamic slots to a class-based language was not an attempt to make it more prototype-based, since we are not sacrificing or limiting any expressibility of the original language. Dynamic slots provide an extra dimension in which objects can be extended, that is orthogonal to the class-hierarchy. This is particularly useful when implementing type-orthogonal abstractions.

Interestingly, the Self system itself does not use `addSlot` in a way similarly to dynamic slots. Instead, they have implemented a primitive that can assign an immutable hash value on an object. The extra word we extend BETA objects with, could also be used to store a unique and immutable hash value. However, this will require the programmer to explicitly create a hash table to store all (object,value) relations[7]. We find that the "extensible object" metaphor to be a more direct and general concept.

LISP also contains a mechanism similar to dynamic slots. A LISP symbol has an associated property list, where (key,value) pairs can be stored. The mechanism is not as general as dynamic slots, since it is only available on symbols as compared to on all objects. Also, static type-checking is not an issue in LISP. In LISP implementations, property-list are typically not stored with the symbol, but in an external hash table hashed on the symbol's hash value. Dynamic slots have also been proposed for CLOS [Kiczales et al. 91], where they denote statically declared attributes but for which memory is allocated on demand. These are very similar to the dynamic slots presented

[7]Notice that, e.g., a persistent store could not use the hash value directly as an OID, because OIDs must stay unique across program executions.

in this paper, except that we allow dynamic slots to be declared outside a class definition. That is crucial to support type-orthogonal abstractions.

Metalevel interfaces have been constructed for several statically-typed languages, including Open C++ [Chiba 95], Napier88 [Kirby et al. 94], and BETA [Brandt & Schmidt 96]. They all support the construction of type-orthogonal abstractions directly in the language itself. In principle, all such abstractions can associate additional information to the objects they work on. A type-orthogonal facility to extend objects is therefore needed to allow maximum flexibility and efficiency of metalevel programs. In Open C++ and Napier88 this is supported by compile-time modification of the source code. As argued in Section 2.1, we do not believe that all such modification can be satisfactorily done at compile time. The metalevel interface for BETA is entirely a runtime entity. It does not require access to the source code for the objects it works on. Hence, a mechanism for runtime extensibility of objects is mandatory.

5 Performance

To evaluate dynamic slots we use three benchmarks. First, we measure the performance of the dynamic slot primitives to see how they perform compared to ordinary BETA attributes. This measures the raw speed of the implementation. Second, we examine the runtime overhead of adding dynamic slot support to the BETA runtime system, i.e., what is the impact on ordinary applications. Third, we have implemented a version of the persistent store that uses dynamic slot and measured the speed-up in comparison with the original implementation and a "static" implementation.

All the measurements have been performed on a 75MHz dual-processor SUN SparcStation 20 with 175Mb of memory. The machine was lightly loaded and had approximately 50Mb of free memory at the time of the measurements. The times measured are the total user + system time for a given operation as reported by the Solaris 2.4 operating system. All times are given as an average over a large number of executions.

5.1 Microbenchmark

Table 2 shows the performance of the three dynamic slot primitives. In the benchmark, the object was already extended with the FirstSlot reference and the linked list of dynamic slots only contained one element. Hence, the measured results are the best-case performance of the primtives. The `set` operation will provoke an IOA garbage-collection once for each object of a new type it is invoked on. Performing an IOA garbage-collection takes on average 18 msecs.

Primitive	Time
init	1.1
set	4.1
get	2.3

Table 2: Performance of the dynamic slot primitives (in μsecs).

An access to a statically declared integer attribute takes approximately 0.15 μsecs, so dynamic slots sacrifice more than an order of magnitude in performance. An invocation of a null-method takes 0.9 μsecs, so a large percent of the overhead is due to the method invocation. This could be eliminated by inlining the `set` and `get` primitives.

5.2 Runtime Overhead Benchmark

Extending the BETA runtime system with dynamic slot support introduces two kinds of potential runtime overhead: i) objects in AOA are eagerly extended with the FirstSlot reference, so the memory requirements are increased, and ii) the garbage-collector must check if an object has been extended, and in that case follow the FirstSlot reference when marking live objects.

Since the effect of adding dynamic slots will be most apparent on long running batch-programs, we use the BETA compiler as a benchmark. Three versions of the compiler are used, the standard BETA Compiler (Standard), a version that eagerly extend *all* objects with the extra FirstSlot attributes (Trivial), and the one that only eagerly extend the objects in AOA (Dynamic). By comparing, the Trivial and the Dynamic implementation we see the effect of avoiding extending objects in AOA.

	Standard	Trivial		Dynamic	
		Total	Overhead	Total	Overhead
Exec. time	352s	386s	9.7%	360s	2.3%
Mem. usage	1.84Gb	1.89Gb	2.7%	1.89Gb	2.7%
IOA GCs	5770	7256	+1486	5770	0
AOA GCs	17	20	+3	18	+1

Table 3: Implementation Overhead for the BETA Compiler.

Table 3 shows the runtime statistics of compiling an approximately 28,000 lines program (the compiler itself) with the three versions of the BETA compiler. The Trivial version shows a significantly changed garbage-collection behavior compared to both the Standard version and the Dynamic version. Thus, avoiding changing the size of objects in IOA is an important optimization. The Dynamic implementation shows an execution time overhead of 2.3% which we find acceptable. The extra 2.7% of memory usage does require an extra AOA garbage-collection. The average time for an AOA garbage-collection is 0.26 secs, which means that most of the execution time overhead is spent in the dynamic slot handling code in the garbage-collector.

Our current implementation of the dynamic slot handling is not optimal and could be improved slightly, but that will require two minor compiler changes. The implementation extends an object with two words, because BETA objects are double-word aligned. The last word in, on average, half of the objects is not used, and therefore could be used to store the FirstSlot reference. This will cut the memory requirements in half, as well as avoiding an IOA garbage-collection to extend objects which have an unused word. If the compiler indicated in the prototype that there is an unused word, this could easily be implemented.

Another possible optimization is the handling of the FirstSlot reference. The garbage-collector must follow the FirstSlot reference in order to locate all live objects. This is currently implemented by explicit checks in the garbage-collector code, thereby introducing an overhead even when dynamic slots are not used. Instead, the *dynamic reference table* in a prototype could be extended at runtime. It contains the offsets of all reference attributes in an object. In the BETA implementation, the dynamic reference table is represented as a null-terminated list of offsets. If the compiler was changed to make a double-null terminated list, the list could be extended by overwriting the first terminating null. Using this scheme, following the FirstSlot reference will introduce no performance overhead for applications that do not use dynamic slots, at the expense of slightly larger prototypes.

5.3 Persistent Store Benchmark

Finally, to evaluate the performance increase provided by dynamic slots when implementing type-orthogonal abstractions, we have modified the existing implementation of a persistent store [Brandt 94] to use dynamic slots. We compare that implementation against the original type-orthogonal implementation and against a "static" implementation. In the "static" implementation the OID attribute is pre-allocated for all objects so it can be accessed directly. This is done by requiring all persistent objects to be a subtype of a special PersistentObject class. In effect, the "static" implementation is trading type-orthogonality for speed.

The time complexity of the checkpoint operation is proportional to the number of objects, n, and the total number of object references, m, since it both has to visit all objects as well as following all references. For the original implementation, a linear scan is required to translate an object reference to an OID, so the time complexity is $O(n + mn)$, or simply $O(mn)$. In the dynamic slot implementation (and "static" implementation), an object reference can be translated into an OID in constant time, so the time complexity is $O(m+n)$, or simply $O(m)$, when m dominates n. Thus, we would expect an asympotical speed-up by using dynamic slots.

To evaluate the effect of the reference density, we have measured the time to checkpoint a single linked list, where there is one reference in each object ($m = n-1$), and we have measured the time to checkpoint a fully connected graph, where all objects have references to the others ($m = n^2$). For each structure, both the time to checkpoint it the first time (1), where OIDs have to be assigned, and the time to checkpoint it a second time (2), where all objects have been assigned OIDs, are measured.

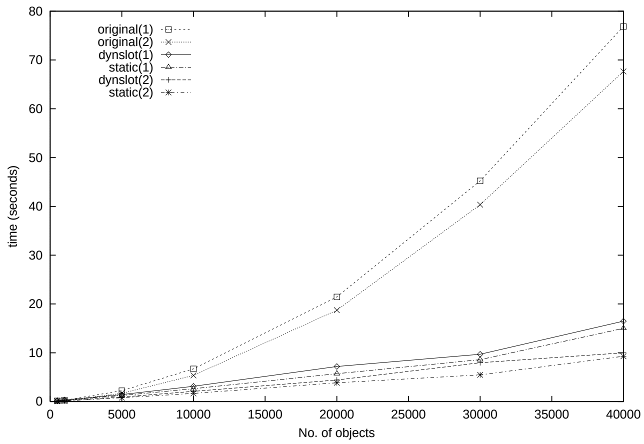

Figure 6: Checkpointing a single linked list.

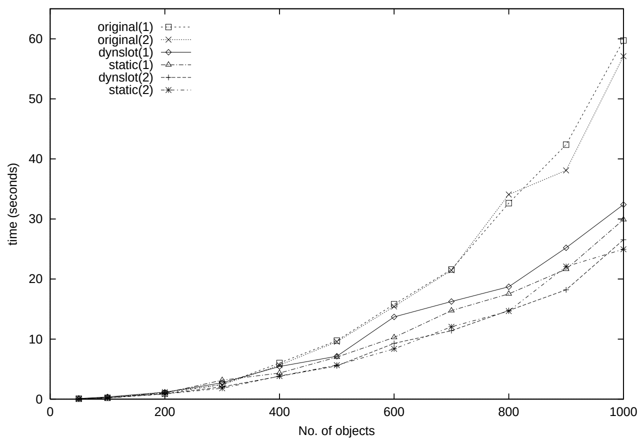

Figure 7: Checkpointing a fully connected graph.

The results are shown in Figure 6 and 7. For the single linked list the size of the data structure is approximately 1 MB for 40,000 objects, and the fully connected graph the size is 3.8 MB for 1,000 objects. The minor fluxturations in the graphs are due to different garbage-collection behavior, e.g., the swap in original(1) and original(2) in Figure 7 at 800 objects.

As expected, we get an asympotic speed up by using dynamic slots, since the linear scan is avoided. For the single linked list, it is apperant that the dynamic slot implementation exhibits a linear time complexity in the number of objects, whereas the original implementation exhibits a higher-order time complexity. Furthermore, the dynamic slot implementation provides a significant performance increase for checkpointing object structures for which OIDs have already been assigned. The dynamic slot implementation performs close to the optimal speed exhibited by the "static" implementation. They are only separated by a small constant factor. This shows that type-orthogonality and speed does not have to rule each other out.

6 Conclusion

We have argued that the concept of dynamically extensible objects can easily be incorporated into a statically-typed and class-based language. Dynamic slots allow new substance to be added to an object at runtime. They extend the expressive power by providing a new dimension of extensibility in addition to the class hierarchy. By using this mechanism, type-orthogonal abstractions can be implemented in a straight forward and efficient manner.

We have shown that strong typing can be maintained, since dynamic slots are statically declared in the source code. Dynamic slots thereby seamlessly integrate into the BETA type system, and do not give rise to potential runtime errors. Furthermore, a simple and efficient implementation strategy was presented. Measurements of our prototype implementation showed both little overhead for ordinary applications, as well as efficient access to the dynamic slot primitives. An existing implementation of a type-orthogonal persistence showed significant speed-ups when re-implemented with dynamic slots. Our implementation strategy is specifically geared towards the current BETA implementation. However, it relies only on simple changes to the runtime system, so we belive that it will be possible to get equal or better performance for a wide variety of language implementations.

Acknowledgements

The author would like to thank Søren Brandt for the many discussions about both the dynamic slot interface, the implementation, and how to modify the persistent store. Ole Lehrmann Madsen and Alexandre Sousa also provided many valuable suggestions.

References

[Brandt & Madsen 94] S. Brandt and O. L. Madsen. Object-Oriented Distributed Programming in BETA. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Lecture Notes in Computer Science 791*, pages 185 – 212. Springer-Verlag, January 1994.

[Brandt & Schmidt 96] S. Brandt and R. W. Schmidt. The Design of a Metalevel Architecture for the BETA Language. In C. Zimmermann, editor, *Metaobject Protocols*. CRC Press Inc, Boca Raton, Florida, May 1996.

[Brandt 94] S. Brandt. Implementing Shared and Persistent Objects in BETA — Progress Report. Technical report, Department of Computer Science, University of Aarhus, May 1994.

[Chiba 95] S. Chiba. A Metaobject Protocol for C++. In *Proceedings of the 10th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 1995.

[Goldberg & Robson 89] A. Goldberg and D. Robson. *Smalltalk-80. The Language.* Addison-Wesley, Reading, MA, 1989.

[Grarup & Seligmann 93] S. Grarup and J. Seligmann. Incremental Mature Garbage Collection. Master's thesis, Department of Computer Science, University of Aarhus, August 1993.

[Hölzle & Ungar 94] U. Hölzle and D. Ungar. A Third Generation Self Implementation: Reconciling Responsiveness with Performance. In *Proceedings of the 9th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 229 – 243, October 1994.

[Jul et al. 88] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[Kiczales 92] G. Kiczales. Towards a new model of Abstraction in Software Engineering. In A. Yonezawa and B. C. Smith, editors, *Proceedings of International Workshop on Reflection and Meta-level Architecture (IMSA)*, pages 1–11, November 1992.

[Kiczales et al. 91] G. Kiczales, J. Rivieres, and D. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[Kirby et al. 94] G. Kirby, R. Connor, and R. Morrison. START: A Linguistic Reflection Tool Using Hyper-Program Technology. In *Proceedings of the 6th International Workshop on Persistent Object Systems*, pages 346 – 365, September 1994.

[Madsen et al. 93] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language.* Addison Wesley, Reading, MA, 1993.

[Meyer 92] B. Meyer. *Eiffel, The Language*. Prentice Hall, 1992.

[Nelson 91] G. Nelson, editor. *System Programming with Modula-3*. Prentice Hall Series in Innovative Technology, 1991.

[Shapiro 86] M. Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *Proceedings of The Sixth International Conference on Distributed Computing Systems.*, pages 198 – 204, May 1986.

[Steele 84] G. L. Steele. *Common LISP: The Language*. Digital Press, 1984.

[Stroustrup 93] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, second edition, 1993.

[Ungar 84] D. Ungar. Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm. In *Proceedings of the First Symposium of Practical Software Development Environments*, pages 157–167, April 1984.