Dynamically Extensible Objects in a Class-Based Language

René W. Schmidt
Department of Computer Science
University of Aarhus
DK-8000 Aarhus C, Denmark
rws@daimi.aau.dk

Abstract

Object-oriented programming techniques support construction of reusable and extensible code. However, class-based languages have poor support for implementing type-orthogonal behavior and extending non-leaf classes, which results 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 are statically typed and have been integrated into the BETA programming language. Measurements of a persistent store show that they significantly improve performance.

Introduction

The object-oriented programming paradigm is generally renowned for its modeling capabilities. Real-world concepts and phenomena can be modeled directly by classes and objects, and refinement of concepts is supported by subclassing. A concept is typically organized into a framework, which is a set of classes that implements a given concept, e.g., a window framework. A framework can typically be used as-is, but can also be used as a starting point for more specialized frameworks.

Figure 1: Reference asymmetry

Inherently in a framework is a predefined object structure. The designer of the framework has decided how objects refer to each other. Subclassing a framework allows adding new functional components that follows the predefined object structure, but only allows limited changes to the object structure. For example, it is not possible to extend a window framework, so all objects have a direct reference to the top-most window, without modifying the basic class describing a window. However, it is possible to create an object that references all window objects. This is due to the asymmetry between in-going and

out-going references. It is always possible to create more in-going references to an object (by creating objects that have references to it) but not vice-versa. This is illustrated in Figure 1.

This asymmetry gives rise to a number of problems must notable the back-pointer problem. It is not possible to construct a class that implements a list, where an object in a list has a back-pointer into the list it is contained in. Another example is when implementing type-orthogonal abstractions. A type-orthogonal abstraction can be viewed as imposing a new role (or view) onto an object. For example, an application looks at a window object as a "physical" window on the screen, where as a *PersistentStore* object looks at it as some state that needs to be serialized. The "role information" cannot easily be attached to an object, because it is not possible to have an extra reference out of an existing object that describes the role.

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 language [Jul et al. 88] is designed and implemented with compiler and runtime support for distribution.

A mechanism for fixing the above asymmetry is proposed in this paper, called *dynamic slots*. Dynamic slots make it possible to extend objects at runtime with additional attributes. Dynamic slots are statically declared in the source code, hence statically typed, but storage is not allocated until their first use.

We have incorporated dynamic slots into the static 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 programs, 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 is significantly improved.

The rest of the paper is organized as follows. The following section describes dynamic slots in detail. Section 3 gives several real-word examples of where they are applicable. Section 4 presents how they can be implemented efficiently in a statically typed language. Section 5 discusses related strategies. Section 6 evaluates our implementation of dynamic slots for BETA. Finally, Section 7 presents our conclusions.

Dynamic Slots

A dynamic slot is an attribute that 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*. Adding behavior 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 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.

Syntax

Instead of creating a new syntax for declaring dynamic slots, they have been added to the BETA language as a class. Each dynamic slot is represented as an instance of the DynSlot class, shown in Figure 2.

```
class DynSlot:
(# type:< Object;    (* virtual class *)
   method set:
   (#
     enter (o: ^Object, value: ^type)
     do ...
   #);

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

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

Figure 2: Declaration of the DynSlot class

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. 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 dynamic slot definition to class Object and initializes the slot to NONE for all objects.

The drawback of modeling dynamic slots as a class is that access to dynamic slots and ordinary BETA attributes are different. In a language that supports operator overloading, e.g., C++ [Stroustrup 93], this could potentially be avoided. Another approach is to extend the BETA language definition with a special syntactical construct for dynamic slot declaration. However, for the initial experimentation with dynamic slots that was considered less of an issue.

Declaration

The declaration point of a dynamic slot is independent on which objects it exists in. Recall that a dynamic slot is available on all objects. However, the declaration point of a dynamic slot determines which part of the code that has access to the slot. A dynamic slot is represented as an ordinary BETA object, hence, the BETA scopeing rules determine which code has access to the dynamic slot. Of course, a dynamic slot object can be passed by reference to all parts of the code, just as any other object.

The following BETA fragment shows a declaration of a dynamic slot psSlot that is of type: reference to a psInfo class.

class psInfo: (# OID: @integer #);

psSlot: @DynSlot(# type::class psInfo #); ***(* Dyn. Slot instance *)***

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 maintain static type-checking. In BETA, this is done using a virtual class. In C++, the template mechanism could be used.

Motivation

To further motivate dynamic slots, we will look at three examples where they are applicable. Based on these examples, we will argue that the kinds of runtime extensibility provided by dynamic slots are indeed applicable and useful in a compiled language. The same kinds 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 not possible by using subclassing alone.

Persistent Storage

The persistence model for BETA is based on reachability. A set of objects is 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. Implied in this model is that the PStore is completely type-orthogonal. It can serialize any object with no regard to its type.

```
class PStore:
(# class psInfo: (# OID: @integer #);      (* role information *)

   psSlot:  @DynSlot(# type::class psInfo #);  (* dyn. slot   *)
   nextOID: @integer;                          (* OID counter *)

   method getOID:
   (# info: ^psInfo;
     enter (o: ^Object)
     do o[]->psSlot.get->info[];      (* access dyn. slot of o *)
        (if info[]=NONE then
           new &psInfo[]->info[];         (* create new psInfo *)
           nextOID->info.OID;
           nextOID+1->nextOID;            (* incr. OID counter *)
           (o[],info[])->psSlot.set;    (* assign to dyn. slot *)
        if);
     exit info.OID                                (* return OID *)
   #);

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

Figure 3: Using dynamic slots to store object identifiers

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 a unique OID by the PStore.

The translation between object references and OIDs during the checkpoint operation is a major source of overhead in the current implementation. Converting an object reference into an OID requires a linear scan of a list containing pairs of object references and OIDs an operation which complexity is proportional to the number of objects that have been serialized. The linear scan is necessary because, in general, is it impossible to hash on an object reference in a garbage-collected language, since most garbage collectors move objects around in memory. The OIDs cannot be stored directly in an object, because the object can be of an arbitrary type. 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 improved.

Figure 3 outlines how an OID look-up method (getOID) can be implemented with dynamic slots. The Pstore creates a dynamic slot (psSlot) to store the extra OID for each object. The getOID method looks up the value of that dynamic slot on a specific object and returns the OID. If an OID has not been assigned yet, the method automatically assign an OID to the object.

A 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 persistent objects. Hence, the memory requirements for storing the OID values will be minimal. This same property is difficult, if not impossible, to achieve through compile or link time program modifications. Such modifications must rely on a conservative program analysis to determine which objects that may need to be extended.

Distribution

The BETA 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 by the proxy 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 addition state information such as which communication channels 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 type-orthogonality away for efficiency.

Dynamic slots can be used in the distribution library, in a similar way as they were used in the PStore example. It will thereby be possible to remove the type-limitation, while retaining efficiency.

Extending Frameworks

The last two examples discuss how dynamic slots are practical when implementing type-orthogonal abstractions. This example is more general than the preceding two. It outlines how dynamic slots can be used to solve an extensibility problem in object-oriented software engineering.

Consider the classical example of a class hierarchy describing graphical shapes. A general shape is described with the class Shape, which implement methods for manipulating shapes, such as moving, resizing, etc. More specific shapes (e.g., triangles, squares, etc.) are added to the system by subclassing the Shape class.

Now we want to describe shapes that have colors. Then we have to modify the Shape class with an extra attribute describing the color. Unfortunately, this requires us to change and recompile the entire framework, including all subclasses of Shape. This might not only be impossible if the framework is supplied by a third-party vendor, it might also not be a desirable solution. It can be argued that a color attribute is not an intrinsic part of a shape, and clearly not all conceivable application utilizing the shape framework will need the color attribute. Hence, the color attribute does not belong in the shape class. However, having two parallel hierarchies, Shape and ColoredShape, is not an ideal solution either.

An alternative solution is to implement the color attribute using delegation. Shapes with colors have a reference to an object describing its color. In this manner, extra substance can be added to all shapes orthogonal to the shape hierarchy. A dynamic slots reference can be used to dynamically add the color substance to a shape, by referring to an object describing the color. A method, getShapeColor, can be written (similar to the getOID method), that given a shape as an argument, returns a color object.

This solution requires no recompilation or access to source code, and the color substance is available for all current and further implementation of shapes.

Implementation

The main motivation behind dynamic slots is to enhance the performance of certain types of abstractions, so 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.

- 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.

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 static typed and garbage-collected object-oriented language. A similar implementation can be used for languages such as Modula-3 [Nelson 91], Eiffel [Meyer 92], and Java [Arnold & Gosling 96].

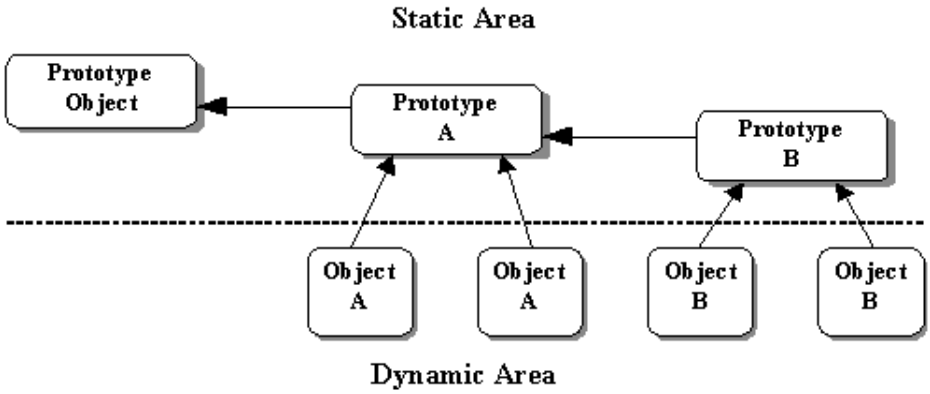

Figure 4: Object layout in BETA

The memory layout is split into two areas. The static part that contains immutable information about each class in a program execution, and a dynamic part that contains the dynamically created objects during a 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 structure for two instances of class A and two instances of class B. 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.

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 is 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-collector as for IOA, a mark & sweep collector is used.

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. Thereby 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.

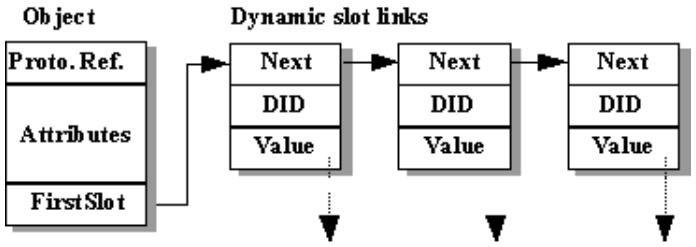

Figure 5: Representation of dynamic slots in an object

Figure 5 shows an object with three dynamic slots. A slot is located by following the *FirstSlot* reference that is stored in the last word of the object. The functionality of the DynSlot primitives are as follows: 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.

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 dynamic slots are used, it might be feasible to store the dynamic slot values in a hash table instead. The dynamic slot implementation could automatically 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 approach aim at fast access at the cost of a high setup time, since it requires a scan of the heap to update pointers and possible recompilation of code. In contrast, representing dynamic slots as a linked list aims at medium setup cost and at medium access time. In addition, this scheme can easily be integrated into most compiled language implementation without requiring modifications to the compiler.

Extending Objects

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

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

The memory behaviors of several BETA programs have previously been studied [Grarup & Seligmann 93] and are 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, thereby significantly degrade performance.

The ideal solution would be to only extend objects that 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. However, the smallest size of a prototype in BETA is 32 bytes. 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 & 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. In addition, 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.

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 the addSlot method in a way similarly to dynamic slots. Instead, a primitive that returns an immutable hash value on an object exists in Self. 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. We find that the "extensible object" metaphor to be a more direct and general concept.

LISP [Steele 84] 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 lists 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 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 static typed languages, including Open C++ [Chiba 95], Napier88 [Kirby et al. 94], 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 changing the language implementation. Many such abstractions need to associate additional state to the objects they work on, e.g., object distribution,

profiling, and debugging tools. 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 source code. The metalevel interface for BETA is entirely a runtime entity. It does not rely on compile-time modifications or access to source code. Hence, a mechanism for runtime extensibility of objects, such as dynamic slots, is required.

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 measured the performance of a version of the persistent store that uses dynamic slots, in comparison with the original implementation and a non-type-orthogonal 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.

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 primitives. 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.

Runtime Overhead Benchmark

Extending the BETA runtime system with dynamic slot support introduces two kinds of runtime overhead: i) objects in AOA are eagerly extended with the FirstSlot reference, so the garbage-collection behavior is changed, and ii) during IOA collections, 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 a version where objects are only eagerly extend in AOA (Dynamic). By comparing, the Trivial and the Dynamic implementation we see the effect of avoiding extending objects in IOA.

	Standard	**Trivial**		**Dynamic**	
		Total	**Overhead**	**Total**	**Overhead**
Execution time	352s	386s	9.7%	360s	2.3%
Memory 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 do 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 new dynamic slot handling code in the garbage collector.

Our current implementation of dynamic slots could be further improved. In the current implementation, an object is extended with two words, because BETA objects are required to be double word aligned. On average, the last word in 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 that have an unused word. If the compiler was modified to set a flag in the prototype indicating this situation, this optimization 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.

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 asymptotically 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 ($m=n^2$).

The results are shown in Figure 6 and Figure 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 fluctuations in the graphs are due to different garbage-collection behavior.

As expected, we get a dramatic speed-up by using dynamic slots, since the linear scan is avoided. For the single linked list, it is apparent that the dynamic slot implementation exhibits a near linear time complexity in the number of objects, whereas the original implementation exhibits a higher-order time complexity. Furthermore, the dynamic slot implementation performs very close to the optimal speed exhibited by the "static" implementation. They are only separated by a small constant factor. This shows that the dynamic slot abstraction is very well suited to solve the type-orthogonality/speed trade-off.

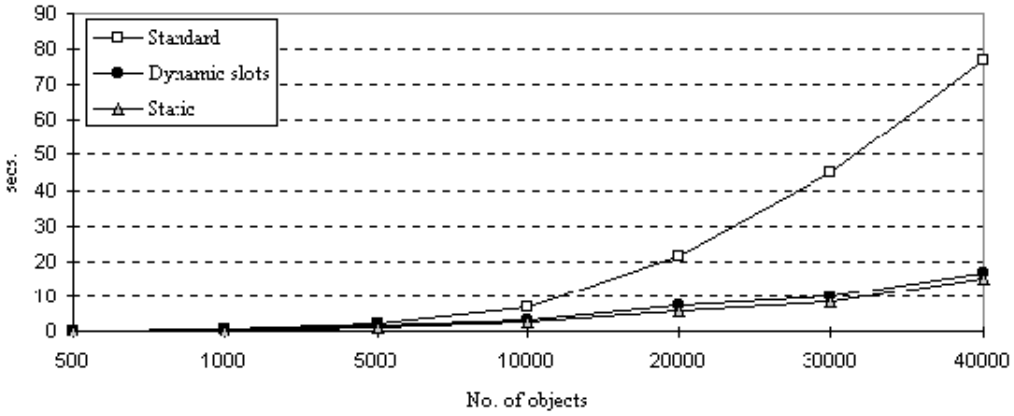

Figure 6: Checkpointing

a single-linked list

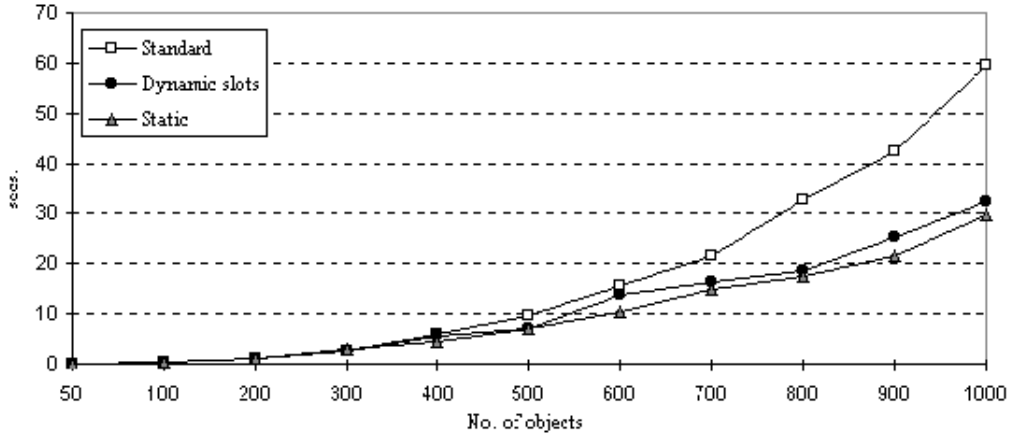

Figure 7: Checkpointing

a fully connected graph

Conclusion

At first, it might seem that the types of problems we are trying to address with dynamic slots are coursed by bad design of frameworks. Thus, they could be solved by statically including some new attributes in the design. However, often this is neither possible nor feasible. Such solutions are likely to either go against the principle of extensibility or simplicity or both.

The persistent store and distribution examples illustrated that the original implementations suffered from limited type-orthogonality or performance. The graphical shape example illustrated how dynamic slots can be used to avoid bloated or parallel class hierarchies. Furthermore, dynamic slots make it possible to extend non-leaf classes without having access to the source code. Dynamic slots extend the expressive power by providing a new dimension of extensibility in addition to the class hierarchy.

It has been shown that dynamic slots can be seamlessly integrated into a statically typed language, such as BETA. They do not compromising type-safety, introduce potential runtime errors, nor require runtime

type checking. 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. Furthermore, as described in the last section, we expect that it is possible to enhance our implementation further by simple compiler support.

Acknowledgment

Most parts of this work were carried out at the Department of Computer Science at the University of Aarhus. The author wish to thanks Søren Brandt, Alexander Sousa, and Ole Lehrmann Madsen for the many valuable comments and discussions about dynamic slots.

References

[Arnold & Gosling 96] K. Arnold and J. Gosling. *The Java™ Programming Language*. Addison-Wesley, 1996.

[Brandt & Madsen 94] S. Brandt and O. L. Madsen. Object-Oriented Distributed Programming in BETA. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Object-Based Distributed Programming*, 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, *Advances in Object-Oriented Metalevel Architectures and Reflection.* CRC Press Inc., Boca Raton, Florida, 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)*, 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)*, November 1992.

[Kiczales et al. 91] G. Kiczales, J. Rivieres, and D. Bodrow. *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*, 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 6th International Conference on Distributed Computing Systems*, 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,* April 1984.