An Efficient Implementation of Persistent Objects

Jacob Sparre Andersen

Jacob Sparre Andersen Research & Innovation
Vesterbrogade 148K, 1. th.
1620 København V
Danmark

Abstract. Persistent objects form a general and very useful method for storing internal program data between executions of a program. And as [1] points out, Ada is an excellent language for implementing persistent objects. This paper introduces a low-impact, efficient implementation of persistent objects based on storage pools and the "POSIX.Memory_Mapping" API [2]. The performance and reliability of the implementation is compared with serialisation.

1 Introduction

There are two basic ideas behind this paper. The one is that memory-mapping is an extremely fast I/O method. The other is that Ada storage pools allow us to control where in virtual memory dynamically allocated objects are stored. These two ideas combined allow us to make dynamically allocated objects be stored in a part of virtual memory which is mapped to a file, and thus automatically stored. Binding an access type to a specific storage pool takes only a single attribute definition clause, making this technique very easy to use.

In section 2 an interface for managing persistent objects is presented. In section 3 the actual implementation is described. In section 4 a comparison to other techniques for implementing persistence, including an experimental comparison with serialisation, is presented. Finally, in section 5, we conclude and point to possible future enhancements of this technique. The full source code of the system, as well as demonstration programs is available from `http://www.jacob-sparre.dk/persistence/`.

2 An Interface for Persistent Objects

The concept of persistent objects is about maintaining a collection of objects created by an application from one execution of the application to the next. Two of the techniques devised for this purpose are serialisation, where the objects are written to a file represented as a stream, and storage in a database, where the objects in the process memory are simply buffers for data stored in a relational database.

J. Real and T. Vardanega (Eds.): Ada-Europe 2010, LNCS 6106, pp. 265–275, 2010.

The reader is pointed to [3] for a general and thorough presentation of persistent objects in relation to Ada. The persistent objects technique presented in the following gives the programmer something close to *orthogonal* persistence, with the major limitation that it only works on explicitly allocated objects.

2.1 Package Specification

The package "Persistent_Storage_Pool" declares a descendant of "System.Storage_Pools.Root_Storage_Pool". All objects allocated on a storage pool of this type will be persistent.

```
package Persistent_Storage_Pool is
   type Instance is new System.Storage_Pools.
      Root_Storage_Pool with private;
```

The package also declares the abstract tagged type "Root_Object", and a function to access the root object of a storage pool:

```
   type Root_Object is abstract tagged null record;
   subtype Root_Class is Root_Object'Class;
   type Root_Name is access all Root_Class;
   [...]
   function Root (Pool : Instance) return Root_Name;
```

The root object of a persistent storage pool is the starting point for the collection of persistent objects. I.e. if one wants to make a tree structure of objects persistent, one will make the root node of the tree the root object of the persistent storage pool. Notice that the root object of a storage pool is the object referred to by "System.Storage_Pools.Root". Other objects in the "System.Storage_Pools.Root_Class" are only persistent if they are allocated in the storage pool.

A persistent storage pool object is activated by a call to one of the procedures "Create" and "Load". "Create" takes four arguments: the storage pool to activate, a file name, an initial value for the root object, and the number of storage elements to allocate for the pool:

```
   procedure Create
     (Pool          : in out Instance;
      As            : in     String;
      Initial_Value : in     Root_Class;
      Size          : in     System.Storage_Elements.
         Storage_Count);
```

"Load" manages with two arguments: the storage pool to activate, and the name of the file to load it from:

```
   procedure Load (Pool : in out Instance;
                   From : in     String);
```

Before "Create" or "Load" has been called on a persistent storage pool object, all operations on the pool will raise "Ada.IO_Exceptions.Status_Error".

The specifications of "Allocate", "Deallocate" and "Storage_Size" are in the private part of the package.

Notice that there is no "Close" operation, since that easily would result in dangling pointers. A persistent storage pool is only closed and deallocated as the program terminates. This is done by the operating system.

2.2 Use

Using this system for making objects persistent is actually quite simple:

- An object of the persistent storage pool type is declared.
- For each type of object which should be persistent, an access type is declared as using the persistent storage pool object as its storage pool.
- A storage pool root type is derived from "Persistent_Storage_Pool.Root_Object".
- Finally the persistent storage pool is created or loaded from a file.

```
with Persistent_Storage_Pool;
   [...]
   Persistent : Persistent_Storage_Pool.Instance;
   [...]
   type Some_Reference is access all Some_Class;
   for Some_Reference'Storage_Pool use Persistent;
   [...]
   type Another_Reference is access all Another_Class;
   for Another_Reference'Storage_Pool use Persistent;
   [...]
   type Root is new Persistent_Storage_Pool.Root_Object
      with [...];
   [...]
   Create_or_Load (Pool => Persistent, [...]);
```

2.3 Example

The demonstration program makes a single-linked list persistent by allocating its elements in a persistent storage pool object. First the pool object is declared:

```
Storage_Pool : Persistent_Storage_Pool.Instance;
```

and then the access type for the nodes in the list is hooked up with the pool object:

```
type Reference is access all Object;
for Reference'Storage_Pool use Storage_Pool;
```

To access the head of the list, a descendant of "Persistent_Storage_Pool.Root_Object" is declared:

```
type Root is new Persistent_Storage_Pool.Root_Object
   with
   record
      First : Reference;
   end record;
```

Now we either create a new pool:

```
Create (Pool          => Storage_Pool,
        As            => Argument (2),
        Initial_Value => Root'(First => null),
        Size          => 100);
```

or load an existing one:

```
Load (Pool => Storage_Pool,
      From => Argument (2));
```

(We see that the name of the storage pool file is passed to the demonstration program as the second command line argument.)

Then we can find the tail of the linked list as usual:

```
Tail := Root_Reference (Storage_Pool.Root).First;
while Tail.Next /= null loop
   Tail := Tail.Next;
end loop;
```

And when we allocate a new object as the new tail of the linked list:

```
Tail.Next := new Object'(Data => "Tofta Teld",
                         Next => null);
```

it automatically ends up in the persistent pool.

3 Implementation Using Memory-Mapped Files

3.1 Memory-Mapped Files

To understand memory mapped files, we can start with a quote from the POSIX specification of the function "mmap" [4]:

> The mmap() function shall establish a mapping between a process' address space and a file.

The actual copying of data between disk and RAM is handled by the operating system. Essentially the mapped file is assigned as swap space to its part of the process' address space. This gives us the possibility of saving some copying between disk and RAM; if the operating system for example already has "swapped" the file to disk, saving the data has zero cost – they are already in the file.

The POSIX specification of "mmap" gives us an implicit guarantee that the mapped file will contain an exact copy of the process memory once "unmap" has been called (i.e. once the program has stopped).

For the purpose of the technique presented in this paper, a very important feature of "mmap" and "POSIX.Memory_Mapping.Map_Memory" is that it is possible to ask the operating system to map a file to a specified part of the process address space. I.e. when we map a file into memory, we can have it placed at the same address as last time, thus maintaining the validity of pointers pointing to specific locations in the memory mapped file.

Without this feature, we would need to be able to implement relative access types[1] in Ada, to make the technique feasible. Since POSIX does not guarantee that we always will be allowed to map a file to the address we request, it would broaden the possible use of the technique, if Ada allowed us to implement relative access types. The use of address space layout randomisation is likely to create problems for this technique, so a future-proof version of this technique will require relative access types.

Although this implementation is using the Ada POSIX API, it is likely that memory mapping implementations in non-POSIX operating systems will work equally well. According to [5] "Most modern operating systems or runtime environments support some form of memory-mapped file access", so even if your target platform isn't POSIX compatible, it is likely that the technique can be used without too many modifications.

3.2 Implementation Technique

Besides declaring access types to use a persistent storage pool, the core of the technique lies in the procedures "Create" and "Load", which take care of asking the operating system to map a file to memory.

procedure Create. The first step in creating a persistent storage pool is to create and open a file for it. With the POSIX Ada API this is done with "POSIX.IO.Open_Or_Create".

The second step is to allocate the requested space for the storage pool in the file. This is done using "POSIX.IO.Seek" together with an instantiation of "POSIX.IO.Generic_Write"[2].

The third step is to map the file into memory (at an address chosen by the operating system) with "POSIX.Memory_Mapping. Map_Memory":

```
Pool.Address := Map_Memory (Length     => Pool.Size,
                            Protection => Allow_Read
                               + Allow_Write,
                            Mapping    => Map_Shared,
                            File       => Pool.File,
                            Offset     => 0);
```

[1] I.e. access types, where the system address corresponding to an access type can be modified with a fixed offset at run-time.

[2] A more general implementation might extend the size of the storage pool automatically, when needed.

The fourth step is to store the basic parameters of the storage pool in the storage pool itself. Since the storage pool at this point is mapped into memory, this can be done by placing an object in the storage pool memory space with:

```
Header : Persistent_Storage_Pool.Header;
pragma Import (Ada, Header);
for Header'Address use Pool.Address;
```

and then copy the parameters to the "Header" object:

```
Header := (Key       => Persistent_Storage_Pool.Key,
           Address   => Pool.Address,
           Allocated => Conversions.Storage (Header'
              Size),
           Root      => null);
```

Finally we allocate space for the root object in the storage pool, and copy the initial value of the root object there.

procedure Load. The first step in loading a persistent storage pool is to open the file it is stored in. With the POSIX Ada API this is done with "POSIX.IO. Open".

The second step is to load the basic parameters for the storage pool from the file with an instantiation of "POSIX.IO.Generic_Read".

The third step is to check that the memory area, where it was last time, is not occupied. This is done with the function "mincore()" (which unfortunately isn't in POSIX yet). If this check fails, "Load" will raise the exception "Storage_Error"[3].

The last step is to map the file into the memory area where it was last time, so references will continue to point to the same objects. Like in procedure "Create", this is done with "POSIX.Memory_Mapping.Map_Memory".

In between these steps, there are various consistency checks on the loaded data. If these checks fail the exception "Persistent_Storage_Pool.Bad_Pool_ Format" will be raised.

4 Comparison with Other Techniques

To test the actual impact of this technique, two test programs have been made. Both of them create or load a network of objects, and then explore it. The only difference between the two programs is which persistence implementation they use to avoid creating the network, if it already has been created. One of these programs (B) uses Ada.Streams to implement persistence following the pattern described in [6], while the other one (C) uses the technique presented in this paper to implement persistence.

[3] The expected reason for this check to fail, is that the operating system uses address space layout randomisation. In that case, the solution could be to let "POSIX.Unsafe_Process_Primitives.Exec" rerandomise the address space, since it is likely that a new random layout will not occupy the relevant memory area.

4.1 Speed

Figure 1 shows how much time it takes to run programs B and C in the mode where they create a network from scratch and write it to disk. The measurements appear to be dominated by the time it takes to create the network. Since program C uses the operating system to save the data, measuring the time inside the program would not be fair to program B. As we can see from the graph, there appears to be a small, but significant speed difference in favour of program C, although the large error bars on a few of the data-points make the case a bit muddled.

Figure 2 shows how much time it takes to run programs B and C in the mode where they load a network from disk and explore it. Here we can see that for large numbers of objects, program C is significantly faster than program B. The two fitted lines are linear with a small offset. For large network sizes this corresponds to program C being approximately 8 times faster than program B.

No measurements have been made of loading a network, modifying it, and then saving the modified network. Since program B has to rewrite the whole network, it is likely that program C will be significantly faster for this combination of actions.

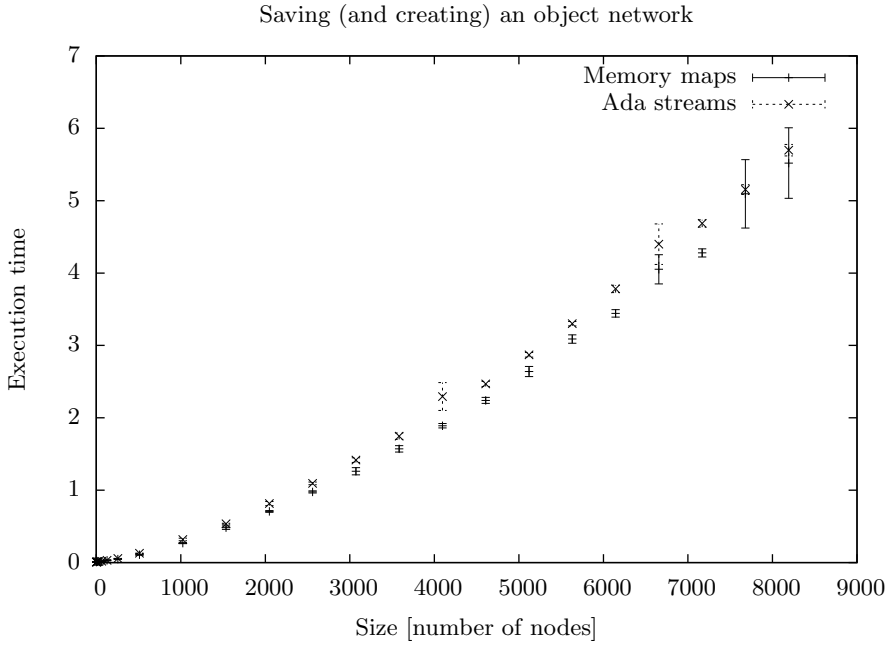

Fig. 1. Comparing the writing speed of program C (Memory maps) and program B (Ada streams). Error bars correspond to 95% confidence intervals.

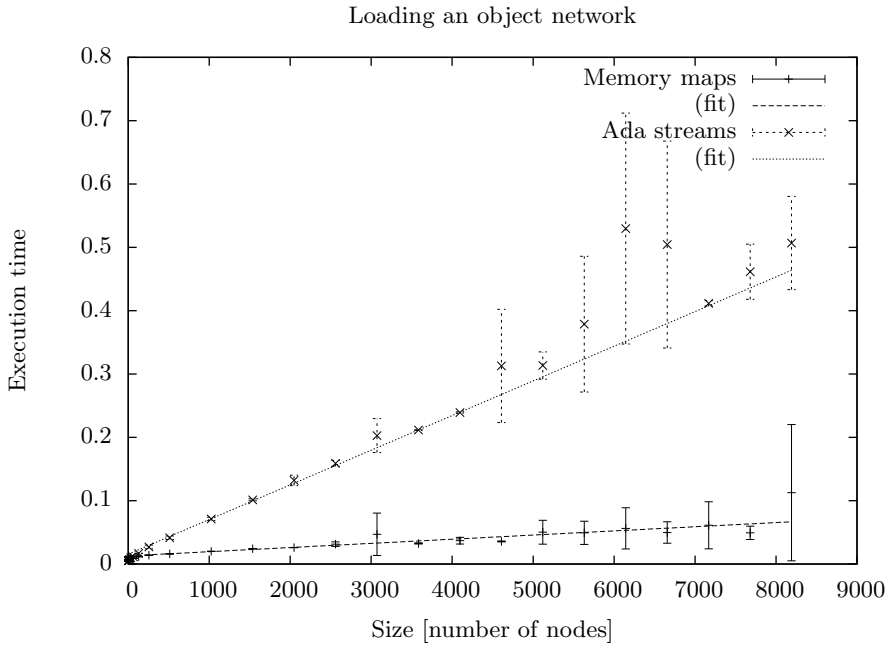

Fig. 2. Comparing the reading speed of program C (Memory maps) and program B (Ada streams). Error bars correspond to 95% confidence intervals. Linear functions are fitted to the measurements.

4.2 Persistence Manager

Using memory maps to implement persistence moves a bit of the responsibility from the application to the operating system.

One could claim that this reduces the risk of loosing data, since the operating system will take care of saving the persistent objects, if the application dies[4]. The down-side of this is that the application may die while the persistent data are in an inconsistent state, leaving the data inconsistent for the next time the application is executed. A safe implementation should therefore either maintain the persistent data constantly in a consistent state, or maintain a flag in the persistent data, which indicates if the data are consistent or not.

4.3 Shared Data

Memory maps can be shared between several processes, such that several instances of the same program can operate on the same persistent data structure.

[4] The guarantee that the mapped file will contain an exact copy of the process memory once "unmap" has been called (i.e. once the program has stopped) is only implicit in the POSIX specification of "mmap" [4].

Although this requires that the programmer implements the appropriate locking using primitives supplied by the operating system[5], it is still an improvement over stream-based and other load-work-store type persistence implementations.

4.4 System Calls

System calls have a large impact on the performance of applications on practically all architectures, since the CPU has to switch from the application/user context to an operating system context.

Using memory maps reduces the number of system calls needed to implement persistence to a fixed number per execution of the application, no matter how much data is being stored[6].

Implementing persistence using serialisation (streams) will result in a number of system calls which will scale linearly with the number of objects being stored. Inserting a buffering stream between the serialisation routine and the operating system, will reduce the number of system calls. With a careful implementation a buffering stream may even use as few system calls as using memory-mapped files.

4.5 Virtual Memory

To understand the performance of I/O implementations on a modern operating system, it is necessary to remember that modern operating systems work with the concept of virtual memory. Virtual memory is **not** the same as RAM. Virtual memory should rather be seen as a unified address space, where the operating system freely moves the actual data around between disk (swap), RAM and CPU caches. At the same time, the operating system maintains RAM caches with parts of files. Each process has its own virtual memory, and when data are copied from an operating system controlled resource, such as a disk, to a process, there is a performance cost since the operation requires both a context switch and moving data around.

Virtual memory, disk based swap space, and RAM cached files make it hard to make an exact estimate of how large a volume of data is copied between disk and RAM. What we can do is estimate the minimal volume of data copied around. For a traditional persistent object system it is $O(N)$ whereas the implementation presented here is $O(1)$, since the only data the operating system is required to copy is the fixed size head of the persistent storage pool file. In practise we will of course expect the process to access some of the objects in the persistent storage pool, and then they will have to be copied as well. But since we use a memory map, the whole process of managing which parts of the persistent storage pool are in RAM, and which are on disk is handled by the operating system. – This is not likely to be the dominant performance cost, since moving data around is a relatively cheap process.

[5] Ada protected objects cannot be shared in this way, if one wants their semantics to be preserved.

[6] Extending the persistent storage will require some system calls.

4.6 Dangling References

If a persistent object contains a reference (access) to a non-persistent object, the referenced object will have disappeared the next time the persistent storage pool is loaded, resulting in a dangling pointer.

The technique presented here requires that all references from persistent objects are references to persistent objects. Currently this is checked manually, and thus a potential source of errors. A source code analyser, such as AdaControl [7], should be extended to make this check automatic.

Serialising objects using Ada streams pose a similar challenge, but in this case the tool presented in [6] appears to be able to solve the problem in an automated fashion.

4.7 Storage Format Stability

When a program is recompiled, the layout of data types, type tags, etc. may change. Since Ada uses name based type equivalence, this makes sense. Unfortunately this (and name based type equivalence) will make a persistent storage pool from one version of a program unusable for another version of the program, such that programs cannot rely on this technique for long-term storage. For long-term storage – i.e. data which should persist beyond the life-time of a specific version of a program – it is still necessary to use a documented, implementation-independent storage format.

Implementing persistence using streams, does not automatically solve the problem of saving in an implementation-independent file format, but it is probably easier than with memory-mapped files.

5 Conclusion and Future Work

We have demonstrated a technique for handling persistent objects in Ada. The technique works on existing Ada compilers without any modifications.

We have demonstrated that the technique is significantly faster than the most prominent competing technique for implementing persistent objects.

We have demonstrated that persistence can be implemented at the cost of adding a single attribute definition clause to each persistent object access type in an application.

Altogether the technique presented here delivers faster I/O and less impact on the code of the program using it, compared to serialisation.

Although the technique does not **require** external tool support, the use of the technique will be safer with tool support. An obvious choice would be to add the required rule to AdaControl.

We have not yet demonstrated that the technique works on major non-POSIX operating systems. Nor have we solved the problems which address space layout randomisation may introduce.

The big drawback of the technique is that it isn't guaranteed to work on all operating systems with an Ada compiler.

There are thus three steps, which together will improve the benefit of using the presented technique in the areas of safety, reliability and portability:

- Extend AdaControl to check that persistent objects only contain access types which refer to objects in the same persistent storage pool.
- Extend how Ada handles access types, so we can create relative access types, and thus avoid the problems address space layout randomisation may introduce.
- Move the handling of memory-mapped files into the Ada standard.

The first of these steps is easy, the second is difficult, and the third does not seem all that likely to happen.

References

1. Card, M.P.: Why Ada is the right choice for object databases. CrossTalk (1997)
2. IEEE: IEEE STD 1003.5: 1990, Information Technology – POSIX Ada Language Interfaces Part 1: Binding for System Application Program Interface, as amended by IEEE STD 1003.5b: 1996, Amendment 1: Realtime Extensions (1996)
3. Crawley, S., Oudshoorn, M.: Orthogonal persistence and Ada. In: Proceedings of the conference on TRI-Ada 1994, pp. 298–308. ACM, New York (1994)
4. The Open Group Base Specifications: mmap (2004), `http://www.opengroup.org/onlinepubs/000095399/functions/mmap.html`
5. Wikipedia, the free encyclopedia: Memory-mapped file (2010), `http://en.wikipedia.org/wiki/Memory-mapped_file`
6. García, R.G., Strohmeier, A., Keller, L.: Automatic Serialization of Dynamic Structures in Ada. Technical report, École Polytechnique Fédérale de Lausanne, Infoscience (2003)
7. Adalog: AdaControl (2009), `http://www.adalog.fr/adacontrol2.htm`