Persistent Containers with Ada 2012

Jacob Sparre Andersen

JSA Research & Innovation
Jægerparken 5, 2. th.
2970 Hørsholm
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 Ada 2012 [2] style containers with a memory mapped file as the backing. The persistent containers allow an application programmer to make objects stored in a container persistent with only small modifications to the source text of the application. The performance and reliability of the implementation is compared with serialisation and with persistent storage pools [3].

1 Introduction

This paper is the result of a review of my paper on persistent storage pools [3] with two things in mind: Benefitting from new features of Ada 2012, and avoiding the conflict with address space layout randomisation inherent in [3]. There are two basic ideas behind this paper. The one is that memory-mapping is an extremely efficient I/O method. The other is that Ada 2012 style containers is a much more programmer-friendly way of storing objects than explicitly allocating them on a storage pool.

These two ideas combined allow us to create a container, which allocates space for its contents in a part of virtual memory which is mapped to a file, and thus automatically stored. Using a persistent container or one of the containers declared under `Ada.Containers` only differs in the call to bind the container to its backing file, making this technique very easy to use.

In section 2 the interface to and use of persistent containers is presented. Section 3 describes the actual implementation. In section 4 a comparison to other techniques for implementing persistence, including an experimental comparison with serialisation, is presented. Finally, in sections 5 and 6, we summarise, conclude and indicate how persistent containers can be enhanced compared with this technique covered in this paper. The full source code of the system, as well as demonstration programs are available from `http://www.jacob-sparre.dk/persistent-containers`.

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.

The reader is pointed to [4] for a general and thorough presentation of persistent objects in relation to Ada. The ideal for persistence implementations is generally that it is "transparent" (or "orthogonal"). I.e. the programming environment/compiler does all the work of saving and loading the full program state between executions.

The interface presented here is not *quite* that easy to use, but it has the benefit of allowing the programmer to control which parts of the program state is persistent; objects stored in **persistent containers** are persistent, while all other objects are not. The persistent containers have to be bound to a file, but once that is done, the container library and the operating system takes care of maintaining a persistent copy of the objects stored in the persistent containers. The Ada 2012 aspect "`Implicit_Dereference`" allows containers to give users safe, direct access to objects stored in them (without having to copy the objects out of the containers first), thus letting the program work directly on the persistent version of an object.

There are no special requirements on the types of objects made persistent, except that system addresses, access types and objects related to tasking and synchronisation should not be stored in persistent containers.

At the time of writing, the library of persistent containers is limited to a linked list container proof-of-concept.

2.1 Package specification

The generic package "Persistent_Containers.Linked_List" declares a persistent linked list container. Comparing it with "Ada.Containers.Doubly_Linked_List", the main difference[1] is the procedure used to bind a list to the file used to persist the container contents:

```
procedure Open_Or_Create (Container    : in out Instance;
                          Name         : in     String;
                          Minimum_Size : in     Positive);
```

Although the container contents are copied automatically to disk, and the associated file is closed on finalisation, the package still provides a `Close` procedure and a `Is_Open` function to inspect the state of the container. If the container is closed (i.e. not associated with a file), all operations but `Open_Or_Create` will raise an exception.

[1] Besides the author being too lazy to make the list doubly linked from the outset.

2.2 Use

A package from the persistent container library is implemented just like one of the containers in the standard library:

```
with Persistent_Containers.Linked_List;

package Character_List is
   new Persistent_Containers.Linked_List (Element_Type =>
       Character);
```

Using the list type declared through the instantiation above, we first declare a container object (`List`) and associate it with a file:

```
   List : Character_List.Instance;
begin
   List.Open_Or_Create (Name         => Name,
                        Minimum_Size => Minimum_Size);
```

Then we can check if the list is empty:

```
   if List.Is_Empty then
```

And append objects if it is:

```
      Insert_Test_Data :
      for C of Test_Data loop
         List.Append (New_Item => C);
      end loop Insert_Test_Data;
```

We can manipulate the objects contained in the list:

```
   ASCII_Caesar_Code :
   for C of List loop
      C := Character'Succ (Character'Succ (Character'Succ (C)))
          ;
   end loop ASCII_Caesar_Code;
```

If we print the contents of the list after updating them;

```
   Iterate :
   for C of List loop
      Ada.Text_IO.Put (C);
   end loop Iterate;
```

then the output will be different (shifted three character values) every time we run the program:

```
% ./bin/example
Ghfhpehu#43wk#4;48
% ./bin/example
Jkikshkx&76zn&7>7;
% ./bin/example
Mnlnvkn{):9}q):A:>
```

The program does not have to do anything to persist the container before it stops.

The full source text of the example described in this section can be found in appendix A.

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" [5]:

> 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 big value of using memory mapping is this saving in physical copying of data between disk and RAM.

The cost of using memory mapping is that we can't handle objects containing absolute memory addresses (such as `System.Address` and access types). Other persistent implementations have the option of "flattening" structures of objects using access types for inter-object reference.

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 (or the program has stopped).

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 [6] "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 Relatively addressed, persistent heap

Between the memory mapped files and the persistent containers, there is a persistent heap addressed with relative addresses such that it does not matter where in the virtual memory the backing file is mapped to.

The `Persistent_Heap` package interfaces with the POSIX API to map and unmap the backing file. It contains a genering package, parametrised with an `Element_Type` for allocating objects on the heap, accessing the "root object"

on the heap, and turning relative heap addresses into Ada 2012 style reference objects with an `Implicit_Dereference` aspect.

The details of the interfacing with the POSIX API are equivalent to the description in [3], with the major difference being that we don't ask to have the file mapped to a specific address.

3.3 Persistent containers

The demonstration implementation of a persistent linked list container primarily differes from any other linked list implementation written in Ada in how **new**, **.all**, **access** and `'Access` have been substituted with the equivalent operations and types from the `Persistent_Heap` package; `Operations.Allocate`, `Operations.Element`, `Operations.Reference_Type`.

As an example, we can look at the `Prepend` procedure:

```
procedure Prepend (Container : in out Instance;
                   New_Item  : in     Element_Type) is
begin
   if Container.Heap.Is_Open then
      declare
         New_Node : constant Node_Operations.Reference_Type
           := Node_Operations.Allocate (Container.Heap);
      begin
         New_Node := Node_Type'(Element => New_Item,
                                Next    => Header (Container
                                    ).First);
         Header (Container).First := New_Node.Address;
         Header (Container).Length := Header (Container).
             Length + 1;
      end;
   else
      raise Constraint_Error with "Prepend: Container has no
          file backing.";
   end if;
end Prepend;
```

`New_Node` is a `Node_Operations.Reference_Type`, which is equivalent to an **access** `Node_Type` type.

`Node_Operations.Allocate` allocates a `Node_Type` object on the indicated persistent heap.

`New_Node.Address` returns the heap-relative address of `New_Node`, as we can't just store the absolute address stored in the reference type.

The observant reader will have noticed that the implementation not yet includes the locking required to ensure safe use of the containers.

4 Comparison with other techniques

4.1 Speed

To test the actual impact of this technique, two test programs have been made. Both of them create or load a linked list of characters, and then manipulate it. The only difference between the two programs is which persistence implementation they use.

As a baseline I use the GNAT implementation of `Ada.Containers.Doubly_Linked_Lists`, with a manual persistence implementation using `Ada.Streams.Stream_IO` and the `'Read` and `'Write` attributes of the list implementation.

The baseline is compared with the described `Persistent_Containers.Linked_List` implementation.

Three timing experiments have been performed:

I+M+W Insert test data into a linked list, modify it and write it to persistent storage.

L+W Load an existing linked list from persistent storage (disk) and write it again.

L+M+W Load an existing linked list from persistent storage (disk), modify it and write back to persistent storage.

The results from the experiments are summarised in table 1.

Experiment	Baseline	Persistent containers
$I + M + W$	0.685	1.000
$L + W$	0.376	0.004
$L + M + W$	0.705	0.390
$M = (L + M + W) - (L + W)$	0.329	0.386

Table 1. Wall clock execution times. Each experiment is performed ten times, and the lowest and highest value is removed before an averaged is calculated. Finally the values are normalised so the highest value is 1.

The first observation we can make is that neither of the implementations is consistently faster.

In pure, measured input+output performance the persistent containers win by an incredible margin. This is because of how the operating system implements memory mapping. The actual copying of data from disk to RAM only happens once the application attempts to access the mapped memory area. Similarly data are only copied from RAM to disk if the application has written to the mapped memory area, and even then only in case of swapping or once the program stops. One way of demonstrating this (see figure 1 for actual measurements) is to run the "L+W" example with larger persistent data sets. The baseline timing will scale linearly with the size of the data set, while the persistent containers timing will be constant.

The calculated times for modifying the linked list (the Caesar code operation shown in section 2.2) are roughly equivalent, with the persistent containers

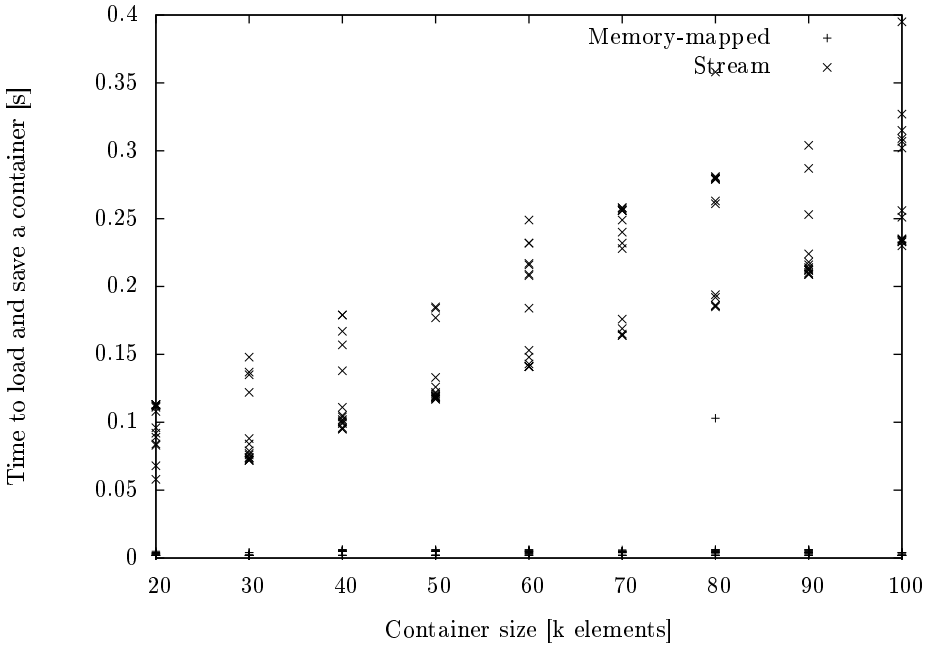

Figure 1. Measured time to load and save the contents of a container depending on the size of the data set and the persistence implementation.

being slightly slower. I expect that the difference would be even larger, if the persistent containers included all the same checks as the baseline container. Considering that the persistent containers are using relative addresses, it is not at all unreasonable that they are a bit slower executing in-memory operations.

It looks like the implementation for inserting objects into the persistent container is unreasonably slow. It may be because of the number of operations on relative addresses, or because the implementation has been written without special considerations for performance.

The performance balance between stream-based and memory map-based persistence lies in the cost of reading and writing the whole container versus the relative addressing cost of each operation on the container. If an application is to run for a long enough time, the per-operation cost will out-weigh the input-output cost, making a persistence implementation based on streams preferable.

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[2]. 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 persistent flag, 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. Although this requires that the programmer implements the appropriate locking using primitives supplied by the operating system[3], 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[4].

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

[2] 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" [5].

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

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

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 persistence implementation 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 – which has algorithms tuned through long experience to do this as efficiently as possible.

4.6 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.

The `Persistent_Heap` package, and thus also the persistent containers built on top of it, has checks to ensure that the source text version of the library using a persistent heap/container matches the version which created it.

Implementing persistence using serialisation and streams, does not automatically solve the problem of saving in an implementation-independent file format, but it is probably easier to implement that way than when mapping objects directly into memory-mapped files.

5 Summary

We have demonstrated a technique for implementing persistent objects using Ada 2012 style containers and memory-mapped files. The technique has been tested on Linux, but is expected to work on any Unix system without modifications. Use of the implementation on a Microsoft Windows system requires an extension of the (currently incomplete) Ada POSIX API, wPOSIX, to include memory mapping, but except for that, the implementation is expected to work unchanged.

We have shown how little a change to an application source text it is to make objects stored in a specific container persistent.

It is not safe to make access types and `System.Address` objects persistent using this technique.

The existing library requires an implementation of the POSIX Ada API to work, but this can be substituted with an explicit binding to the appropriate operating system calls.

Comparing with serialisation

Comparing the technique presented here with using serialisation to persist objects:

- The present technique handles data loading and storage significantly faster.
- The increased load/store speed comes at a cost when accessing the persistent objects.
- Serialisation can in some cases persist objects using access types, while that is never the case with the present technique.
- Serialisation requires only the standard library to work.

Comparing with persistent storage pools

Comparing the technique presented here with using persistent storage pools [3]:

- The present technique avoids the explicit use of pool allocation to make objects persistent.
- The present technique avoids the conflict with address space layout randomisation which is inherent in the use of absolute addresses in persistent storage pools.
- The present technique is slower, as it has to dereference relative addresses.

6 Conclusion and future work

Although the presented technique may be interesting in some cases, it looks like it – in its current form – has too many drawbacks to be generally usable as it is.

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

- Substitute the `POSIX.Memory_Mapping` backing with a `Ada.Streams.Stream_IO` backing in the persistent container library, using `Ada.Finalization` to ensure that the container contents are stored when a persistent container goes out of scope.
- Extend AdaControl with rules to check if access types, `System.Address` objects, or objects related to tasking and synchronisation are written to a stream.

References

1. Card, M.P.: Why Ada is the right choice for object databases. CrossTalk (June 1997) 9–13
2. ISO/IEC JTC 1/SC 22/WG 9 Ada Rapporteur Group: Ada Reference Manual – ISO/IEC 8652:2012(E). http://www.adaic.org/ada-resources/standards/ada12/ (December 2012)
3. Andersen, J.S.: An Efficient Implementation of Persistent Objects. In: Reliable Software Technologiey – Ada-Europe 2010. Volume 6106/2010 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2010) 265–275
4. Crawley, S., Oudshoorn, M. In: Orthogonal persistence and Ada
5. The Open Group: MMAP(2). Issue 6 edn. (2004)
6. Wikipedia: Memory-mapped file — wikipedia, the free encyclopedia (2014) [Online; accessed 16-January-2015].

A Example source text

The full source text of the example used in section 2.2:

```
with Ada.Text_IO;

with Character_List;

procedure Example is
   Name         : constant String := "iterate.list";
   Test_Data    : constant String := "December 10th 1815";
   Minimum_Size : constant := Test_Data'Length;

   List : Character_List.Instance;
begin
   List.Open_Or_Create (Name         => Name,
                        Minimum_Size => Minimum_Size);

   if List.Is_Empty then
      Insert_Test_Data :
      for C of Test_Data loop
         List.Append (New_Item => C);
      end loop Insert_Test_Data;
   end if;

   ASCII_Caesar_Code :
   for C of List loop
      C := Character'Succ (Character'Succ (Character'Succ (C)))
        ;
   end loop ASCII_Caesar_Code;

   Iterate :
   for C of List loop
      Ada.Text_IO.Put (C);
```

```
   end loop Iterate;
end Example;
```