Ouverture

Persistent objects form a general and very useful method for storing internal program data between executions of a program.

This presentation is focused on Ada 2012 style containers backed by a memory mapped file.

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, and future improvements are discussed.

Persistent Containers with Ada 2012

Jacob Sparre Andersen

JSA Research & Innovation

The 20th International Conference on Reliable Software Technologies – Ada-Europe 2015

Who wouldn't want easy orthogonal persistence?

```
O : T with Persistent, Storage => ...;
```

Unfortunately this is not quite Ada.

Who wouldn't want easy orthogonal persistence?

```
   O : T with Persistent, Storage => ...;
```

Unfortunately this is not quite Ada.

But this is possible:

```
with Ada.Containers.Persistent.Sets;
...
   A_Persistent_Set.Insert (O);
```

This is legal Ada. – And with a small, fixed overhead it is enough to give us persistent objects.

A bit of project history...

The work presented here is the result of reviewing "An Efficient Implementation of Persistent Objects" (2010) with two things in mind:

- Benefiting from new features of Ada 2012.
- Avoiding the conflict with address space layout randomisation inherent in my earlier work on the subject.

This presentation is a summary of my paper in Ada User Journal 36.2.

The ideas. . .

- Ada would benefit from an easy-to-use persistence facility.
- Memory-mapping is an extremely efficient I/O method.
- Ada 2012 style containers is a much more programmer-friendly way of storing objects than explicitly allocating them on a storage pool.

. . . and trying to solve the problem within the language, without having to modify the compiler.

Combining the ideas. . .

Combining these ideas gives us 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.

An example

A persistent container package is instantiated just like an equivalent package in the standard library:

```
with Persistent_Containers.Linked_List;

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

Declaring a container and associating it with a file:

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

An example (continued)

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;
   end if;
```

An example (continued)

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;
```

An example (continued)

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:>
```

Faster I/O

Quoting the POSIX specification of the function "mmap":

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

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.

Limitations

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.

Slower access

Address space layout randomisation[1] has effectively made this version of `Map_Memory` unusable:

```
function Map_Memory
  (First      : System.Address;
   Length     : System.Storage_Elements.
       Storage_Offset;
   Protection : Protection_Options;
   Mapping    : Mapping_Options;
   Location   : Location_Options;
   File       : POSIX.IO.File_Descriptor;
   Offset     : POSIX.IO_Count)
   return System.Address;
```

[1] An operating system feature introduced to limit the damage of buffer overflows in C.

Slower access (continued)

Address space layout randomisation means that you have to let the operating system decide where in virtual memory the file will be mapped.

This again means that you have to work with relative addresses in a memory mapped file.

Which again introduces an overhead on practically all operations on a container stored in a memory mapped file.

:-(

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 generic package, parameterised 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.

Persistent containers

The demonstration implementation of a persistent linked list container primarily differs 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;

- **new** → `Operations.Allocate`,
- **.all** → `Operations.Element`,
- **access** → `Operations.Reference_Type`.
- `'Access` → `Operations.Reference_Type`.

Example: **procedure** `Prepend`

```
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)
                                ;
```

Example: **procedure** `Prepend` (continued)

```
            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;
```

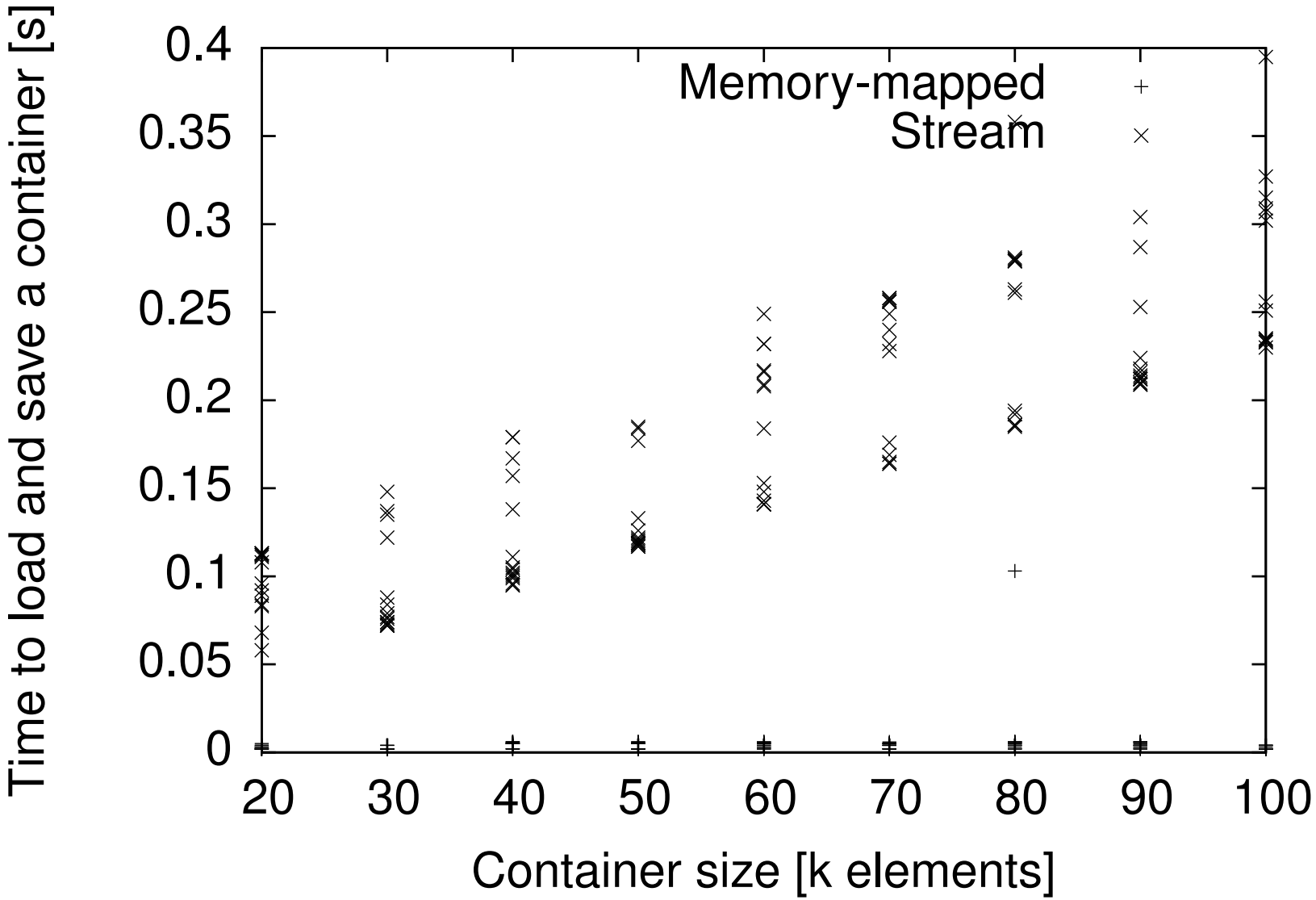
Timing experiments

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.

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


Memory-mapped
Stream
Time to load and save a container [s]
Container size [k elements]
0
0.05
0.1
0.15
0.2
0.25
0.3
0.35
0.4
20
30
40
50
60
70
80
90
100

Performance balance

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.

Persistence management and data consistency

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

One could claim that this reduces the risk of loosing data, as the persistent state isn't lost if the application dies unexpectedly.

But this leaves the application with the risk of being started with the persistent data in an inconsistent state.

A safe implementation should either maintain the persistent data constantly in a consistent state, or keep a journal, which can be used to recover from an inconsistent state.

Storage format stability

When a program is recompiled, the layout of data types, type tags, etc. may change, leaving persistent objects from one version of a program unusable for another version of the program.

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, compiler-independent storage format.

It this is relevant for the application, one would have to replace memory mapping with (for example) serialisation (streams) as the storage implementation.

Memory-mapped persistent containers

I have demonstrated a technique for implementing persistent objects using Ada 2012 style containers and memory-mapped files.

We have seen how small a change to an application source text it takes to make objects stored in a specific container in the application 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.

Serialisation of containers

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.

Persistent storage pools

Comparing the technique presented here with using persistent storage pools:

- 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.

Conclusion and future work

It wasn't **quite** the right way to do it.

- Ada 2012 style containers (using `Implicit_Dereference`) are an elegant interface to an orthogonal persistence implementation, **but**
- memory mapping is not really a viable backing implementation for generic persistence implementations.

Backing an Ada 2012 style container with serialisation to a file on finalisation is probably a more sensible, generically usable persistence implementation.

More future work

Another option could of course be to make the fictive aspects from my introduction a part of Ada 2020:

```
O : T with Persistent, Storage => ...;
```

Contact information and links

Jacob Sparre Andersen
JSA Research & Innovation
jacob@jacob-sparre.dk
http://www.jacob-sparre.dk/

Examples from this presentation:
http://repositories.jacob-sparre.dk/persistent-objects-with-ada-2012

You can find my Open Source software repositories at:
http://repositories.jacob-sparre.dk/