PSTL—A C++ Persistent Standard Template Library*

Thomas Gschwind†
tom@infosys.tuwien.ac.at
http://www.infosys.tuwien.ac.at/Staff/tom/

Institut für Informationssysteme
Technische Universität Wien
Argentinerstraße 8/E1841
A-1040 Wien, Austria

Abstract

The C++ Standard Template Library provides efficient storage of data in containers, and efficient operations on such containers. While STL can be parameterized with custom allocators, these cannot be used to add persistency to the container classes provided by STL. Thus, we have designed the Persistent Standard Template Library (PSTL) that overcomes this by providing its own containers that are compatible with STL, but store their elements on disk. This compatibility provides a programming model that is known and more natural to C++ programmers and enables the reuse of many of the algorithms provided by STL in combination with PSTL. In this paper we discuss PSTL's design, show the challenges we faced, and how STL's design would have to be extended to provide native support for persistency.

1 Introduction

Persistent container libraries such as [GND99, Sle00] have two key advantages. Compared to volatile containers, the size of their database can grow beyond the size of the available memory and compared to transactional databases, they exhibit less overhead since they do not provide any functionality to rollback transactions. Persistent container libraries are the key element of many programs such as sendmail [CA97], `NNTPCache` [AB], or NewsCache [GH99].

Today, a large number of persistent storage libraries exists, ranging from simple text-files to complex databases. The most prominent among these are the various types of `dbm` databases used for instance by sendmail. The creators of the Java programming language have even added a type to Java to indicate whether a given object may be stored on external storage [AG97]. Any object implementing the `Serializable` interface can be serialized and deserialized transparently.

We have not found any persistent container for C++ that is compatible with STL and fulfills the requirements to be used within a server application. The importance of persistent libraries has already been identified by Bjarne Stroustrup. According to [Ste95], his assumption was that persistency could be provided using custom allocators. This assumption, however, was based on an early version of STL that had allocators that were real objects [Str94, Str00]. In later versions of STL the allocator mechanism was simplified due to logical and performance problems with that kind of generality. However, it is still unclear whether these problems would have been unsurmountable [Str00]. Problems with the current allocator mechanism have also been identified by [Ste98b] and will be explained in detail in Section 3.

We think that compatibility with STL is one of the key requirements since it allows for a more natural object-oriented programming style and the reuse of many of the algorithms provided by STL. These algorithms range from iterating over the container's elements to sorting the container and exchanging elements between different containers. Compatibil-

*This work was supported in part by a grant from the USENIX Advanced Computing Association.

†Part of this work was implemented during an internship at Hewlett Packard Labs, Palo Alto, CA 94304.

ity is also one of the key challenges since some key properties of persistent containers such as serialization are not necessary for their volatile counterparts.

The paper is structured as follows. Section 2 gives a brief description of the requirements for a persistent container library. Section 3 explains why the allocator mechanism provided by STL is not sufficient to fulfill these requirements. The design and implementation of PSTL is presented in Section 4 along with the challenges we faced in preserving compatibility with STL. Section 5 presents an evaluation and related work is considered in Section 6. We outline our ideas for improving PSTL in Section 7 and draw our conclusions in Section 8.

2 Requirements

The requirements for persistent containers are simple. Data has to be stored on persistent storage, each container should use its own file, and the container should be able to store more elements then fit into the computer's memory (main memory plus swap space). Additionally, the containers should be simple to use and compatible with existing standards.

Since data is stored on disk, the objects stored in the container need to be serialized. Depending on the object, a `memcpy()` operation might be sufficient. In the following, we refer to these objects as *simple objects* and to those that require special serialization functionality (i.e., objects using pointers) as *fragile objects*.

Since persistent containers are frequently used by server daemons to store their databases (e.g., sendmail, NewsCache, ...), the database must be accessible by several different processes. This is because daemons handle several clients simultaneously and typically spawn a new process for each client connection. Depending on the daemon, the database is created once and accessed read-only or is manipulated during the program's operation. If the container is manipulated, support for locking is needed and changes need to be visible to other processes immediately.

In case of the C programming language, these requirements are fulfilled by [GND99], for instance. For C++, however, we could not find a persistent container class library fulfilling these requirements and seamlessly integrating with STL.

PSTL was primarily developed to replace NewsCache's [GH99] newsgroup and article database. Thus, the containers have to be efficient enough for handling the typical Usenet traffic. The typical spool size of a Usenet News server is about 60GB and the traffic is at least 3–5GB of article data per day [CBB97] excluding the traffic generated by news reading clients. To ease the maintenance of the spool area, NewsCache stores each newsgroup in a separate file/container.

3 STL and Persistence

The containers provided by STL are stored in main memory which is volatile, but can be parameterized with different memory allocators. Thus, our initial approach was to provide a custom allocator that allocates its elements on persistent storage.

While the constructors of the container classes do not directly provide an argument to pass the name of the file the container should be stored in, they provide constructors taking a custom allocator class as argument that could encapsulate the container's filename.

The custom allocator class, however, only provides methods that allow to allocate and free memory and to construct and destroy a given object [Str97, ISO98]. There is no mechanism to query the allocator for previously stored elements. Thus, the constructors of STL's container classes do not check for pre-existing elements. The only way to fix this problem is to subclass the original STL class and replace the constructors and destructor as identified in [Ste98b, Ste98a].

It is insufficient, however, to simply replace the constructors and destructor of the original STL container except in the simple case where the constructor reads all elements from the file and the destructor writes all elements back to the file [Ste98b]. This is due to the fact that typical STL implementations make certain but legitimate optimizations based on the internal representation of the container. For instance, GNU C++'s STL implementation uses a pointer as the vector's iterator. While this is fine for a non-persistent container, it cannot be used for a persistent container as we will show in the following sections.

Another drawback is that a standard allocator is assumed to hold no per-object data [Str97, Chapter 19.4.3] allowing the library to implement some container-manipulation functions by relinking elements. For instance, `splice()` may be implemented by moving an element from one list to another without copying the element. This is not possible if different lists can be stored in different files. In this case the elements have to be copied from one container/file to the other. Thus, the allocator is bound to the type the container is parameterized with rather than to the container. Hence, this would restrict a program to use the same file for all containers parameterized with the same value type.

Due to these reasons it is not possible to convert an STL container into a persistent STL container just by supplying a different allocator class to its constructor. Adding persistence requires a tighter coupling between the container and its corresponding allocator. Thus, we were forced to reimplement the containers provided by STL. The containers provided by PSTL are compatible with their corresponding STL containers, but use an extended interface provided by our persistent allocators.

4 Design and Implementation

Since PSTL tries to adhere to the STL specification, most of PSTL's design is equivalent to STL's design. The differences between PSTL and the typical STL implementation are outlined in the following sections. For instance, we had to add a serialization mechanism that provides transparent serialization of the container's elements. Additionally, we added an argument to the container's constructors to indicate the file where the elements should be stored in. This allows the user to instantiate a container without having to explicitly instantiate the corresponding persistent allocator.

4.1 Serialization

As mentioned in Section 2 special care needs to be taken when *fragile objects* need to be stored in the persistent container. C/C++ pointers cannot be stored because the data stored at the address the pointer is pointing to will likely be located at a different address at the program's next invocation. This problem can be solved by using a pointer swizzling technique as presented in [SKW92] or by serializing the object before storing it on disk. Since the pointer swizzling technique has a major drawback with regards to concurrent accesses, as we will point out in Section 6.5, we have chosen to store the objects in a serialized form.

STL itself does not provide any functionality for the serialization of its elements because the container's elements are stored in memory and do not have to be made persistent. Thus, we had to extend the interfaces defined by STL with a means for serializing and deserializing fragile objects. Typically, this functionality is implemented using one of the following approaches.

- Instrumentation of the data structures [Kni99].
- Requiring the user to provide the code for serializing and deserializing the data structures stored within the container [GND99, Sle00].

The first approach allows the container to identify the type of the data stored at each position. This makes it possible to use a generic algorithm for garbage collection and defragmentation of the database. The price for this, however, is an increased size of the container and that only instrumented data structures may be stored. Another disadvantage is that the layout of the data structure is pre-determined and cannot be changed at run-time thus making the exploitation of polymorphism difficult. Since we want our implementation to be as compatible with the STL as possible, and no garbage collection mechanisms are provided by STL anyway, we have chosen the second approach.

A straightforward implementation would be to require the classes of the data structures to be stored to provide a member function that implements the serialization and deserialization functionality. This implementation, however, would have two shortcomings. It cannot be easily added to any existing class and it does not work in combination with builtin types.

Thus, we have decided to use traits [Mye95, Str97] for PSTL. Traits allow us to extend a template argument without requiring it to provide the extension. The extension is moved to a trait class which is supplied as an additional template argument to the container class. An advantage of this implementation is the possibility to use different serialization algorithms for different container instances parameterized with the same value type.

```
template <class T, class Tr=serialize_trait<T>>
class pvector {
  // typedefs, ...
  reference
  front() {
    // See Section 4.5 for a description
    // of getdata
    offset_type head=alloc.getdata();
    return Tr::deserialize(head);
  }
}
```

Figure 1: Vector Using Serialization Trait

A simplified version of the `pvector` container using the serialization trait class is shown in Figure 1. Whenever the container needs to serialize or deserialize an object it calls the corresponding functions of the trait class. The container and the serialization classes have both a reference (`alloc`) to the persistent allocator class and can use its functions for allocating memory on the persistent file and converting offsets to pointers and vice versa.

Our persistent container classes provide trait classes for storing builtin types, objects that do not have any pointers, and strings. The traits for the serialization of other classes must be provided by the user of the persistent template library. The implementation of a custom serialization trait is fairly simple. The interface that has to be provided is shown in Figure 2.

4.2 Low-Level Disk Access

So far, we have not discussed how and when the data is stored onto disk. Typical solutions to this problem are:

- Read the data from disk in the constructor of the container and write it back in its destructor [Ste98b]. Unfortunately, this approach violates two of our requirements. It does not allow the container to grow beyond the memory size and modifications of the container won't be visible to other processes until the container is freed or flushed explicitly.
- Whenever an element needs to be accessed, seek to the element's position and read it from or write it to disk [GND99, Nel98]. This requires the elements to be read from disk and to be copied from/to the I/O buffers every time an element is accessed. Since all modern operating systems provide elaborate caching strategies, the disk access is negligible. The real problem is that every element needs to be copied at each access no matter whether it actually would have to be serialized or not.

```
template <class T> struct serialize_trait {
  // typedefs, ...
  pstl_serialize_traits(allocator_type &a)
    : alloc(a) {}
  void
  serialize(const T &t, offset_type o) {
    // serialize t to alloc.off2ptr(o)
  }
  reference
  deserialize(offset_type o) {
    // deserialize element and return a
    // reference to it
  }
  const_reference
  deserialize(offset_type o) const {
    // deserialize element and return a
    // constant reference to it
  }
  size_type
  size() {
    // return T's size; in case of a complex
    // object use an offset here and allocate
    // extra memory via the persistent
    // allocator
  }
}
```

Figure 2: Sample Serialization Trait

PSTL, however, uses memory mapped files, an approach different from both of the approaches presented above. Memory mapped files have the advantage that the operating system maps the file associated with the container into the program's address space and the program can read and write the file like memory allocated using `malloc()`.

As shown in Figure 3, PSTL maps the container's file storing the serialized representation of the container into its address space. Whenever an object is accessed, it is deserialized and thus copied. In the Figure below, this is the case with the container's first element, the author's address. Now, the program can interact with the object representing the element like with any other object. Finally, when the object is destroyed, it is serialized back to the memory mapped file.

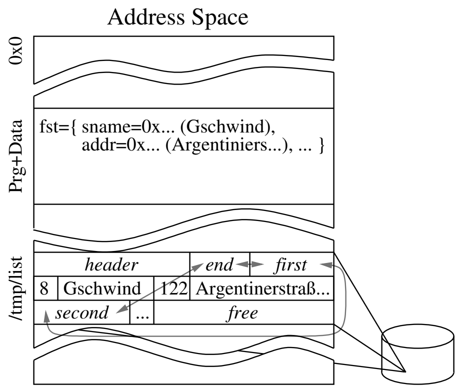

Figure 3: Serialization and Deserialization of Elements

Using memory mapped files has several advantages:

- The program can operate directly on the memory mapped file. No copying from and to I/O buffers is necessary.
- When virtual memory gets tight, the contents of the container's associated file are swapped back to the file instead of to the swap space.
- User programs may use pointers pointing into the memory mapped file and may act on them like on all other pointers.
- *Simple objects* do not have to be deserialized and copied from the container. Since their deserialized representation is the same as their serialized one, it is sufficient to return a reference/pointer to the object's address in the memory-mapped area.

While this approach sounds simple, much care needs to be taken with its implementation. The size of the area to be mapped into memory needs to be known a priori. If more space is required than originally requested, the file needs to be resized and the memory area needs to be remapped. In this case all pointer references to the memory mapped region might have to be calculated anew since it cannot be guaranteed that the resized container can be remapped to the same address as before [Bac86, LMKQ90].

Theoretically, it would be possible to map different parts of the file to different memory regions and thus getting around this problem. This, however, would require the container to maintain a mapping for each region and address calculation would get complicated and most probably inefficient.

Unfortunately, no memory management is available for the memory provided by the memory mapped file. Thus, we had to implement our own persistent allocator class that manages the free blocks of the memory mapped file. The interface provided by the persistent allocators is explained in Section 4.5. Similar to STL, this allocator class is used by all persistent containers.

4.3 References

The most challenging part of PSTL with regard to STL-compatibility was the references returned by some of the container's member-functions (e.g., `front()`). For this problem, however, it is important to distinguish between fragile objects (objects using pointers) and simple objects (objects without pointers).

If the container only stores simple objects, it is sufficient to return a reference to the memory address the object is stored at. This is because the object's serialized and deserialized representation is the same and the object's size is well known and does not change. Hence, there is no danger of accidentally overwriting the container's internal data.

If the container stores fragile objects, however, the object needs to be deserialized and thus copied into a temporary buffer. Otherwise, the user would not be able to use it. However, if the deserialized copy changes, we want the serialized version of the element to be changed as well.

The straightforward approach would be the provision of a wrapper class overriding all of the original methods and serializing the element back to disk when necessary. This is ugly, however, and is not guaranteed to work since some of the class's non-virtual functions might have been expanded inline. Fortunately, this problem can be solved cleanly using templates.

PSTL returns a reference object that encapsulates the position of the data structure in the memory mapped area, provides a deserialized copy of the data structure and in case of a non-constant reference writes the element back to disk when the reference object is destructed. This is achieved by a generic wrapper similar to the one shown in Figure 4. The wrapper only needs to be parameterized with the value-type's serialization trait.

```
template <class Tr>
class __pstl_ref: public Tr::value_type {
  // typedefs, ...
  serializer_type *s;
  offset_type o;
public:
  __pstl_ref(const value_type &x,
             Tr *ser, offset_type off)
    : value_type(x), s(ser), o(off) {}
  ~__pstl_ref() {
    s->destruct(o);
    s->serialize(*(value_type*)this,o);
  }
};
```

Figure 4: PSTL's Reference Class

While PSTL's reference objects ensure that changes will be written back to disk after the element's modification, the element will be written back to disk only after the reference's destruction. To prevent multiple wrappers of the same element from interfering with each other a cache object should keep track of the instantiated wrapper classes and ensure that no more than one wrapper is instantiated for a given element within one process. The locking mechanism ensures that different processes are only allowed to have constant references at the same time.

4.4 Locking

Since we want several processes to be able to access PSTL's container's simultaneously, we added a locking mechanism that ensures mutual exclusion. The current implementation of PSTL simply adds two new functions to each container: `lock()` and `unlock()`. `lock()` locks the container for shared or exclusive access and `unlock()` unlocks the container. PSTL also maintains an internal lock stack to ensure that different functions locking the same container do not interfere with each other. For instance, a function might request to lock a container that has already been locked by its process (i.e., by its caller). In this case `lock()` does not have to lock the container since it is already locked. When the function unlocks the container, however, the container must not be unlocked since the container should be still locked by the caller.

Based on the design of STL, however, it should be possible to add a transparent locking facility. Whenever an iterator is requested, the container could be locked and with the iterator's destruction it could be unlocked. Depending on a constant or non-constant iterator, the container would be locked shared or exclusively. The same would apply to references returned by the container. Unfortunately, depending on the container's value type PSTL uses a wrapper class or a builtin C++ reference. While the wrapper class would support a transparent locking mechanism, the latter does not since C++'s builtin references do not provide a destructor that could be used for unlocking the container. A simple solution of course would be to always return a wrapper object. This issue will be attacked in future versions of PSTL.

4.5 Implementing More Persistent Containers

In case the container types provided by PSTL are not sufficient, it is easy to implement new ones that better fit the requirements of special purpose applications. The implementation of a persistent container is similar to the implementation of a volatile container, except that offsets have to be used instead of pointers and instead of allocating memory using `new`, it has to be requested from the persistent allocator class. The persistent allocator class provides the following functions for the management of the container's persistent memory:

`off2ptr()/ptr2off()` convert offsets to pointers and vice versa. While offsets must be stored in the container, they need to be frequently converted to pointers to operate on the memory-mapped storage area.

`nvalloc()/nvfree()` allocates/frees a memory area within the file. Whenever one of these functions is called, the container's storage area might have to be resized and might have be remapped to a different memory area. Thus, pointers into the persistent storage area need to be recalculated or verified after calling this function.

`getdata()/setdata()` returns/sets the offset to the *root* memory area of the container's data structure. Using `getdata()`, the persistent containers gain access to pre-existing elements at the container's construction time.

5 Evaluation

We evaluated PSTL in terms of compatibility by adapting some sample applications we had written previously to be used by the containers provided by PSTL instead of those provided by STL. Though PSTL has not yet been tuned for performance, we will also give a short performance evaluation to give the reader a rough estimate of PSTL's current performance.

5.1 Compatibility

The main difference from STL is the extension to allow the user to supply a custom serialization functionality. While we have included some serialization classes for common value types, it is likely that the user will have to supply a custom serialization trait for his own value types. Additionally, the filename of the container needs to be specified when instantiating a PSTL container.

```
vector<int> v();                 // STL vector
pvector<int> pv("/tmp/vector"); // PSTL vector
pvector<int,pallocator,myserializer>
    pv2("/tmp/vector2");   // use my serializer
```

When converting our test applications from using the containers provided by STL to the ones provided by PSTL, we identified that users typically make assumptions about the implementation of the containers. A common assumption is that the iterator of a vector is implemented as a pointer or that a reference is implemented as a C++ reference. While this works in combination with most STL implementations, it is not compliant with the standard [ISO98].

```
T* i=pv.begin();                         // error
pvector<T>::iterator j=pv.begin();       // ok
T& r=pv.front();                         // error
pvector<T>::reference s=pv.front();      // ok
```

As we have explained in Section 4.3, PSTL uses its own wrapper class as a reference. Fortunately, this is identified by the compiler and thus can be corrected by the user easily. The same applies to references returned by PSTL.

PSTL's non-constant references copy their elements back to disk, regardless whether they have been modified or not. This is not the case for constant references as they must not be changed by definition and thus do not have to be written back to disk. Hence, if performance is of importance, one should distinguish between member functions returning constant (e.g., `front() const`) and non-constant references (e.g., `front()()`).

C++ resolves the member function to be called based on its function name and the arguments including the implicit `this` argument. Only in case of a constant object pointer/reference, the constant method will be chosen. The following code clarifies this.

```
pvector<int> pv1("/tmp/filename");
const pvector<int> &pv2=pv1;


/**** ok, but inefficient ****/
pvector<int>::reference ref=
  pv1.front();  // call ref begin()
pvector<int>::const_reference cref1=
  pv1.front();  // call ref begin() and
                // convert ref to const_ref


/**** ok, and efficient ****/
pvector<int>::const_reference cref2=
  pv2.front();  // call const_ref begin() const
```

A set of polymorphic elements is typically stored by parameterizing the container with the pointer to the base class (<`base_type*`>) and allocating the elements on the heap. Achieving the same behavior with PSTL requires the user to specify a custom serialization trait for `base_type*`. Serialization is straightforward by calling the appropriate serialization function. To allow for extensibility the deserialization function should be implemented using exemplar types [Cop92] (sometimes also referred to as virtual constructors).

If the performance of manipulating persistent elements is of major concern the user may specify his own reference type within the serialization trait. This allows the reference type to gain access to PSTL's data allocator and manipulate the data structure in the persistent storage area directly.

Even though, in case of concurrent access to the containers, PSTL requires to lock and unlock the containers explicitly, this is not a compatibility problem. STL implementations do not implement persistent containers and thus do not allow simultaneous access. In case a program should be retrofitted to allow multiple processes to access the same data structure, it is necessary to review the code for possible deadlock situations anyway. In future versions of PSTL, however, we will try to implement a transparent locking facility as mentioned in Section 4.4.

5.2 Performance

So far, our efforts on PSTL focused on compatibility with STL and not on its performance. However, we were still curious to see how it would perform in comparison to Berkeley DB [Sle00] (with logging for recovery or transactions disabled) and gdbm [GND99], the leading persistent container libraries available for Unix.

The computer used for this benchmark was a Pentium II (350MHz), 256MB of RAM, and a Seagate 4GB hard drive (ST34323A). The computer was running RedHat 6.1 (Linux kernel 2.2.12, glibc 2.1) and all container libraries and test applications were compiled using gcc-2.95.2 using `-O2` for optimization. Each application was executed 5 times and the median was chosen for our performance comparison.

Due to the lack of available benchmarks for the evaluation of persistent container libraries, we have implemented our own applications: an address book mapping family names to addresses and phone numbers and a resource reservation system. Both applications are based on associative containers with the difference that the resource reservation system uses a simple object as key and thus favoring PSTL. In the following, however, we will limit our discussion to the address book application (Table 1). The results of the resource reservation system are available from the PSTL web site together with the source code of the benchmarks.

	59840 entries			148397 entries	
Database	PSTL	BDB	gdbm	PSTL	BDB
Insertion	9.23	14.73	15.59	31.65	45.28
Iteration	4.30	7.32	8.50	10.30	20.28
Lookup	2.21	5.10	3.00	5.86	16.00
Deletion	69.23	16.27	12.53	482.19	27.58

Table 1: Address Book Benchmark (seconds)

Table 1 shows the results for two different database sizes. One with 59840 entries without duplicates since gdbm does not support duplicate keys and one with 148397 entries with duplicates. *Insertion* refers to inserting all the elements, *iteration* to iterating over all of the elements 10 times, *lookup* looking up each element, and *deletion* to removing all elements one after the other.

Astonishingly, in many cases PSTL performs better than its competitors. We assume that one of the reasons is PSTL's use of memory mapped files.

Typically, Berkeley DB does not use memory mapped files since this would restrict the size of the database to the size of the address space. With the advent of 64bit computers, however, we do not believe this to be a problem for PSTL. Berkeley DB only uses memory mapped files for databases opened read only and smaller than a given threshold. If the smaller Berkeley DB database is opened read only, iterating over the elements takes 6.43 seconds and 2.03 seconds for looking them up. This does not show much potential for performance improvement of PSTL here. It is interesting to note that Berkeley DB scales better for inserting new elements and PSTL scales better for iterating over elements and looking them up. This might be due to the fact that PSTL uses a red-black tree and Berkeley DB a B-tree.

GNU gdbm uses a hash table for its internal representation. This also gets clear by looking at its fast lookup speed, even though it uses normal disk I/O. Iteration over the elements is poor since gdbm only returns the key when iterating over the container requiring another lookup for the associated value. If the key is sufficient, the time for iterating over the elements is just 4.05 seconds.

PSTL's performance, however, for deleting the elements shows plenty of potential for improvement. This is due to the fact that PSTL uses a linear list for the management of free blocks sorted by their location within the file. Deleting all of the elements one after the other in random order populates the list with a huge number of elements even though adjacent free blocks are merged immediately. This problem will be attacked in future versions.

6 Related Work

Currently, several persistent container class libraries are available. The most prominent are the various `*dbm` databases.

6.1 `dbm` and Variants

`dbm` is one of the oldest libraries that provide access to persistent containers. Under Unix, `dbm` [Unia] and its newer variants (ndbm [Unib], gdbm [GND99], Berkeley DB [Sle00]) are a de-facto standard for persistent information storage.

The most advanced of the `dbm` databases is Berkeley DB. Unlike the older variants, it does not only provide a hash table for its internal representation, but also a B-tree and two different kinds of queue formats as well as optional transaction management. Additionally, it provides bindings for C, C++, Java, and Tcl.

Berkeley DB requires the elements to be stored in the database to be serialized before the database allocates the element's memory forcing the element to be copied twice. Additionally, the serialization function does not have access to Berkeley's memory allocator and is forced to store the whole element in one chunk. On the other hand, this approach ensures that the database cannot be easily corrupted by the serialization algorithm.

Even though Berkeley DB provides C++ bindings, it provides a very low level API like the older `dbm` variants. Most of its functions work on DBTs (Data Base Thangs) representing raw regions of memory. Whenever a key/value pair needs to be stored in a database, both key and value need to be converted into a DBT. Additionally, when a value corresponding to a key needs to be requested, the key needs to be converted into a DBT and the database returns a DBT representing the value.

PSTL, however, has an API compatible with STL and the serialization function is part of the container's type. Besides that, the serialization function is called transparently and the user does not have to deal with it. Based on our work on PSTL it might be fairly simple to provide a C++ API for Berkeley DB which is compatible with STL too.

6.2 Disk Based Container

The disk containers presented in [Nel98] use the same approach as used by the `*dbm` databases to access elements stored in its containers. Similar to PSTL, the elements in the database are referenced by offsets instead of pointers.

Unfortunately, this work seems to be more an experiment on how persistency could be achieved in C++ without going into much detail. For instance, the management of the free blocks in the container is overly simple and only provides a fixed-size block allocation scheme. This makes it difficult and inefficient to use these containers in combination with variable sized elements. The library is also incompatible with the containers provided by STL.

6.3 Persistent Template Library

The Persistent Template Library is presented in [Ste98b] and [Ste98a]. PTL is a library that provides containers compatible with the ones provided by STL. Unfortunately, the containers have some severe drawbacks:

- Whenever a PTL container is allocated it is copied from disk into main memory. Afterwards it behaves like the STL containers. This is trivial since it uses the same code. When the container is destroyed, its data is written back to disk.
- Due to the above, the size of the container is limited to the size of the main memory. A persistent container, however, should be able to accommodate more elements than fit into the memory. This demand is essential for programs that have to manage huge amounts of data.
- Only one process may instantiate a container at one time since the container is copied into main memory and other processes will see the changes only when it is saved back to disk at destruction.

This work is interesting because the author comes to the conclusion that it is not possible to use the allocator mechanism provided by STL to add persistency. The author has solved the problem by subclassing the corresponding container provided by the STL and by supplying his own constructors for reading the elements from disk and destructor for writing them back. This, unfortunately, led to the above disadvantages.

6.4 POST++

While POST++ does not provide persistent containers directly, it provides a simple and effective storage for application objects [Kni99] and supports the use of different storages for different objects.

Except for a slight instrumentation of the data structures to be stored persistently, POST++ transparently manages the persistence of the objects. For the instrumentation of the data structures, POST++ uses C preprocessor macros that register the attributes to be made persistent. A special macro is provided for the identification of pointers as their management is more complex. Due to the explicit identification of pointers, POST++ even provides garbage collection to reclaim unused storage.

POST++ uses a different but nevertheless interesting approach. The choice whether to use PSTL or POST++ depends largely on the application domain. For instances, POST++ does not provide persistent containers per se, it only provides the infrastructure to make your objects (including container objects) persistent. Thus, if special purpose data structures need to be made persistent and their instrumentation is possible, POST++ might be a good choice. Otherwise, we recommend the use of PSTL.

6.5 Texas

Texas [SKW92] is a persistent storage system similar to POST++ but instead of requiring the user to instrument the data structures, it uses a pointer swizzling technique in combination with runtime type descriptors and slightly modified heap allocation routines. Runtime type descriptors are generated using an optional feature of gcc.

Objects are either allocated on the conventional (transient) heap, or the persistent heap. In the implementation presented in [SKW92], all the objects allocated on the persistent heap are stored within a single file, but the authors claim that it would not be difficult to remove this restriction.

While Texas provides a simple and powerful way to manage data structures located on persistent storage, it has some shortcomings with regards to sharing the persistent database. If several different processes need to share the same database, they need to share the same persistent page mappings. While this might be simple in case of a single file used for all persistent objects, it cannot be easily achieved if different files are used for different persistent objects and the files to be shared between the processes (or the processes sharing the files) cannot be known a priori.

7 Future Work

For the maintenance of free blocks, PSTL uses a linear list, sorted by the memory location of the free blocks. This representation is highly efficient for debugging purposes since it allows an efficient way to check the allocation status of the whole file and whether the free list has been corrupted (e.g., by a method writing past its allocated memory area). As we have shown in Section 5.2, however, it is inefficient from a performance point of view. To alleviate this problem, we plan to provide a better allocator class using a binary tree for the management of free blocks.

PSTL does not yet implement all the containers efficiently. It might be interesting to see whether PSTL could profit from a data structure optimized for block-sized access patterns like a B or B+tree [Com79] or whether the operating system's cache management is sufficient.

We also plan to provide an associative container optimized for lookups using a hash table. It might also be interesting to see whether gdbm's algorithms can be reused for this container to achieve not only interoperability with STL, but also with the most widely used type of persistent container.

In future versions of PSTL we will also try to implement a transparent locking mechanism as explained in Section 4.4.

8 Conclusions

Persistent STL (PSTL) containers provide a variety of benefits not available with existing persistent container implementations available for C++. They provide a variety of different persistent containers, allowing the container's size to grow beyond the available memory, and supporting STL's object-oriented programming model known to many C++ programmers.

In this paper we have shown the problems of adding persistency to STL and demonstrated why it is impossible to use STL's allocator mechanism to add persistency to existing STL containers. This impossibility forced us to implement our own container classes that are interface compatible with STL. Based on this implementation, we have presented how STL's design had to be extended to support persistency.

Based on PSTL, we explained how to implement a transparent serialization facility without breaking interface compatibility with the corresponding STL containers and without requiring any support from the objects to be stored in the persistent container. We solved this using a traits approach that encapsulates the serialization functionality and allows different containers to use different serialization algorithms.

Except for the declaration of the container, elements are serialized and deserialized transparently. This has been solved using a reference wrapper that works in combination with both objects using virtual and non-virtual member functions. Additionally, the persistent allocator class used by PSTL's containers provides a simple but efficient means to create new persistent container classes for special purpose applications.

In an evaluation of the library we identified that STL containers can easily be replaced with PSTL containers. Only little modifications are necessary in case of implicit assumptions about the container's implementation. This, however, is detected by the C++ compiler. We have also included a performance evaluation which will serve as a basis for future improvements of PSTL.

Acknowledgements

This paper could never have been accomplished without the help of several people. I would like to thank Pankaj Garg, my supervisor during my internship with HP labs, for his support of the project, Mehdi Jazayeri, my supervisor at Technische Universität Wien, for his continuous support, and my colleague Georg Trausmuth for his suggestion to add the serialization functionality using traits. Finally, I would like to thank Bjarne Stroustrup for shepherding this paper.

Availability

PSTL is distributed under the GNU Public License and is available at http://www.infosys.tuwien.ac.at/-NewsCache/pstl.html.

References

[AB] Julian Assange and Luke Bowker. NNTPCache. http://www.nntpcache.org/.

[AG97] Ken Arnold and James Gosling. *The Java Programming Language (Java Series)*. Addison-Wesley, 2nd edition, December 1997.

[Bac86] Maurice J. Bach. *The Design of the Unix Operating System*. Prentice-Hall, 1986.

[CA97] Bryan Costales and Eric Allman. *sendmail*. O'Reilly, January 1997.

[CBB97] Nick Christenson, David Beckemeyer, and Trent Baker. A Scalable News Architecture on a Single Spool. *;login:*, 22(3):41–45, June 1997.

[Com79] Douglas Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[Cop92] James O. Coplien. *C++ Programming Styles and Idioms*. Addison-Wesley, August 1992.

[GH99] Thomas Gschwind and Manfred Hauswirth. NewsCache—A High Performance Cache Implementation for Usenet News. In *Proceedings of the 1999 USENIX Annual Technical Conference*, pages 213–224. USENIX, June 1999.

[GND99] Pierre Gaumond, Philip A. Nelson, and Jason Downs. *GNU dbm: A Database Manager*, 1.5 edition, 1999. For GNU dbm, Version 1.8.

[ISO98] ISO/IEC. *ISO/IEC14882: Programming Languages—C++*, 1st edition, July 1998.

[Kni99] Konstantin Knizhnik. Persistent Object Storage for C++. http://-www.ispras.ru/~knizhnik/post.html, March 1999.

[Lip98] Stanley B. Lippman, editor. *C++ Gems*. Cambridge University Press, 1998.

[LMKQ90] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1990.

[Mye95] Nathan C. Myers. Traits: A New and Useful Template Technique. *C++ Report*, June 1995.

[Nel98] Tom Nelson. Disk-Based Container Objects. *C/C++ Users Journal*, pages 45–53, April 1998.

[SKW92] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: An Efficient, Portable Persistent Store. In *Proceedings of the Fifth International Workshop on Persistent Object Systems*, September 1992.

[Sle00] Sleepycat Software. The Berkeley Database 3.1.17, July 2000. http://www.sleepycat.com/.

[Ste95] Al Stevens. Al Stevens Interviews Alex Stepanov. *Dr. Dobb's*, March 1995. http://www.sgi.com/Technology/STL/drdobbs-interview.html.

[Ste98a] Al Stevens. AntiPatterns and PTL 2. *Dr. Dobb's*, pages 114–116, April 1998.

[Ste98b] Al Stevens. The Persistent Template Library. *Dr. Dobb's*, pages 117–120, March 1998.

[Str94] Bjarne Stroustrup. Making a Vector Fit for a Standard. *The C++ Report*, 6(8), October 1994. Also in [Lip98].

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.

[Str00] Bjarne Stroustrup. Personal Email Communication, October 2000.

[Unia] University of California, Berkeley. *dbm(3) Unix Programmer's Manual*.

[Unib] University of California, Berkeley. *ndbm(3) 4.3BSD Unix Programmer's Manual*.