EVector: An efficient vector implementation – Using virtual memory for improving memory

Dries Kimpe[a,b,*], Stefan Vandewalle[b] and Stefaan Poedts[a]
[a]*Centre for Plasma Astrophyics, Celestijnenlaan 200B, 3001 Leuven, Belgium*
[b]*Scientific Computing Research Group, Celestijnenlaan 200A, 3001 Leuven, Belgium*

Abstract Every modern operating system provides some form of virtual memory to its applications. Usually, a hardware memory management unit (MMU) exists to efficiently support this. Although most operating systems allow user programs to indirectly control the MMU, few programs or programming languages actually make use of this facility. This article explores how the MMU can be used to enhance memory handling for resizable arrays. A reference implementation in C++ demonstrates its usability and superiority compared to the standard C++ vector class, and how to combine the scheme with an object-oriented environment. A number of other improvements, based on newly emerged insights in C++ are also presented.

Keywords C++, STL, virtual memory, vector

1. Introduction

C++ has a well proven track record in scientific computing. Its inheritance from C – allowing low level optimisations – combined with dynamic and static polymorphism enables the construction of efficient and clean software. However, existing libraries currently do not make full use of all available system and language features. In this article, a number of improvements to the Standard Template Library [1] (STL) class `vector` will be presented.

1.1. Problem description

The concept of a vector or array, an ordered collection of items, is available in almost all programming languages. It is a base structure on which more complex data structures are built. For example, the STL provides the `deque`, `hash_set` and `hash_map` containers, all of which build on arrays.

In C++, three incarnations of this concept exist (either in the base language or in its standard library): the built-in array inherited from C, the `valarray` class and the `vector` class. Only the latter allows extending an existing instance.

Extending existing vectors is often needed in applications that have to deal with an unpredictable amount of data; even when the number of elements is known in advance extending can still be needed. For example a hash table (implemented on top of a vector), which needs to grow when the number of collisions becomes too high.

*Corresponding author: Dries Kimpe, Celestijnenlaan 200B, 3001 Leuven, Belgium. Tel.: +32 16 327003; Fax.: +32 16 327998; E-mail: Dries.Kimpe@cs.kuleuven.be.

Nevertheless, there are a number of drawbacks to the `vector` class. Although element access happens in constant time, adding or removing an element at the back requires *amortised constant time*, meaning that *only on avarage*, constant time complexity is guaranteed. On some occasions, linear time is needed. This is a consequence of the memory allocation strategy used. Although linear time expansion is sub-optimal, it can be tolerated when extending is not a common operation. The real problem is memory efficiency: extensibility of the vector has a high price in terms of memory. Adding an element to a vector holding n items can momentarily require up to $3n$ elements of memory!

Although this is usually not a problem for small vectors – memory is cheap nowadays – it is clearly non-optimal and can be a disaster for scientific applications, which typically want to make full use of available hardware. Still, on average a vector of size n is wasting $\frac{n}{2}$ bytes! Another well known problem is the inability to *reduce* the size of an existing `vector`. A vector instance can only grow, and will only return its memory upon destruction. Even clearing it completely will not free any memory. The only way to reduce its size is to copy all elements into a new vector and `swap`() it with the old one.

Note that the mentioned time and memory complexity is not specific for C++, as other languages use the same methods and consequently exhibit the same problems. As such, most of the techniques presented here can also be applied to other languages.

1.2. Related work

In [14], Brodnik et al. take an algorithmic approach to improving the standard vector class. They prove that their implementation is optimal within a constant factor. The reason that the evector implementation is able to do better under certain circumstances is that it makes use of operations that do exist in the real memory model but do not exist in the usual theoretical model. Also, evector has the additional advantage that it doesn't change the internal memory layout of the vector. All elements are still contiguously stored in memory. This eases interaction with other languages (fortran, C) and legacy software.

1.3. Paper outline

In the following section, the implementation of the STL `vector` class will be studied. Next, we take a closer look at the specifics of virtual memory, and how they can be used to lift some of the STL `vector` restrictions. A C++ class implementation incorporating these ideas, named `evector`, is described in Section 4. In Section 5 the performance of the `vector` and `evector` class is studied in detail.

2. The STL vector class

The C++ Standard [2] contains the following description of a `vector`:

> A vector is a kind of sequence that supports random access iterators. In addition, it supports (amortized) constant time insert and erase operations at the end; insert and erase in the middle take linear time. Storage management is handled automatically, though hints can be given to improve efficiency.

It has two template parameters, the first being the type of the elements stored and the second the type of allocator to be used. The allocator enables the client to specify custom memory handling routines. Each `vector` instance has a certain capacity (c) and size (s). Its *capacity* is the maximum number of elements which can be stored in the `vector` without reallocating memory. Its *size* is the number of elements actually stored. When adding an element to a `vector` for which $c = s$, thus which is "full", the following happens:

1. Memory for holding $2 \times c$ elements is allocated.
2. All c elements are copied to the newly allocated memory.
3. The old memory block containing c elements is discarded.

After this operation, s remains unchanged and c is now $2 \times s$. Note that during the execution of step two, potentially at least s copy constructors and s destructors had to be called. If the type stored inside the vector aggregates other objects, they too have to be copied and destructed! Step two is also responsible for the *amortized* constant time complexity, while step one is causing the $3 \times n$ memory requirement.

Clearly, this strongly limits the usability of the `vector` class. In an attempt to avoid repeated calls to the expensive grow operation, a `reserve`() method was added. This is one of the "hints" mentioned by the standard. It increases the capacity to a certain value, and is meant to be used in situations where the final vector size is known in advance. As this function garantuees a given *minimal* capacity, it can only be used to *increase* memory usage of the `vector`.

Note that some `std::vector` implementations could use a different grow factor (other than 2). The grow factor controls reallocation frequency (and thus constructing and destructing of the contents) and memory overhead. A large grow factor leads to less frequent reallocation, but has a higher memory overhead. More advanced schemes (for example using a variable grow factor) are also possible but none can provide both memory and time efficiency.

Ideally, extending a `vector` should require constant time and should not claim excessive memory. This first demand clearly excludes all allocate-and-copy strategies, which are to be avoided anyway for performance reasons. Copying and destructing objects can take a significant amount of time, or – as is the case with singletons – is sometimes not even possible. Without memory constraints, an easy solution would be to make c as large as the available memory. The downside of this strategy is of course its poor memory efficiency. On the other hand, reallocating, together with all of its disadvantages, is never needed. It turns out that virtual memory enables exactly that *without* consuming all available memory!

3. Virtual memory

3.1. Introduction

At the time when virtual memory was conceived, memory was very expensive and occupied a lot of physical space. A few kilobytes could easily fill the room. Machines typically had less memory than today's calculators. Large programs had to use complicated tricks to manually load the needed code parts into memory. Virtual memory set out to solve this problem. It gives the application the illusion of vast amounts of memory. However, in reality, only a subset of the total allocated memory is really present in main memory at any given time. The rest is stored on secondary storage (typically disk), ready to be loaded into main memory when needed. Just as cache memory gives the illusion of fast access to the slower underlying (physical) memory, physical memory is used as a cache for the slower disks, which are storing virtual memory.

Nowadays, the situation is reversed. Memory is small and cheap, and machines typically have "enough" memory (alleviating the usage of disks for temporarily storing data).[1] Still, virtual memory is widely present. It turned out to be the perfect solution to a new emerging issue in modern operating systems: memory protection in the presence of concurrency. As virtual memory gives each application its own dedicated virtual memory space, all applications are perfectly isolated, which makes concurrency, fault tolerance and security easier to implement. This is one of the main reasons why virtual memory is still used in almost every modern operating system, and even in some specialized embedded systems.

Virtual memory decouples *physical storage* from its *address*. In a sense, memory is split up in two resources: *address space* and *storage space*. Every single running program has a separate address space, unshared with others. Also, the amount of virtual memory does not depend on the size of the physical memory installed. For example, every process running on a 32 bit linux system will have between two and four *gigabytes* of available address space, even though the system itself may only have a couple of *megabytes* of main memory.

[1] Some CPU's, notably the Intel Xeon with PAE enabled, can even have a larger physical than virtual address space.

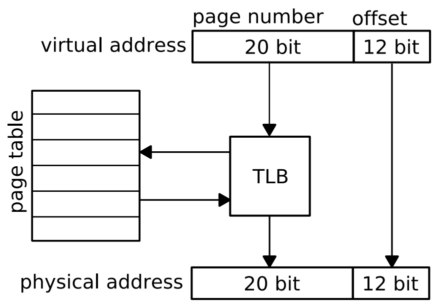

Fig. 1. Translation from virtual to physical address.

3.2. Implementation

Usually, virtual memory is implemented through *page tables*. Both physical and virtual memory is divided in *pages* – typically a couple of kilobytes or megabytes in size – which are atomic. A page is either completely present in physical memory, or not at all. For performance reasons, most CPUs support a number of different page sizes. The page tables define a mapping between virtual and physical pages, and also record whether a page is present or not. For every memory access, the virtual page number is derived and looked up in the page tables. If the page was already present in memory, its corresponding physical page number can be retrieved. When the page is marked as missing, an exception occurs and the operating system gets a chance to resolve the situation. This typically means finding (or creating) a free page, and allocating it to the virtual page. Next, the instruction that caused the memory access is retried. This scheme has the advantage that virtual memory is "automatic". For the running program, it looks as if all pages are always present. By making use of this feature, programs larger than physical memory can be executed: every time a page not present in main memory is referenced, an older virtual page is stored on disk. Its corresponding physical page is subsequently reused to hold the contents of the referenced virtual page.

There is a clear drawback associated to this mechanism: every memory access actually needs *two* accesses! A first one to read the page tables to map the virtual address to a physical one, and a second to actually read the data. To reduce this problem, the *translation lookaside buffer* (TLB) was introduced. It acts as a cache for recent translations. When a mapping can be found in the TLB, the extra memory reference can be avoided. Figure 1 illustrates these concepts on a 32 bit machine with a page size of 4096 bytes. More information on virtual memory can be found in [8].

3.3. Operating system support

Operating Sytems make heavy use of virtual memory; it simplifies memory protection, concurrency and the sharing of libraries between programs. Shared memory is easily implemented: the page tables of different processes just refer to the same physical pages. Because the page tables also store permission flags for every virtual page, a physical page can easily be read-only in one process, and read-write in another.

Usually, the OS exports some interface for controlling virtual memory. Almost all support the concept of memory mapped files. In this case, instead of using a special (zero-initialized) swap area as backing store for the virtual memory, a normal file is used. To the user program, it looks as if a memory region "mirrors" the actual file. Whenever it accesses a page that was not referenced before, a page fault occurs. The OS handles this by finding a free physical page and redirecting the access to it. However, in this case, the page is not cleared with zeros. Instead the part of the file this virtual page refers to is determined, and the file data is read into the page before returning control to the user program. Likewise, when ending the program or freeing the mapping, the page tables are used to detect modified pages, which are written back to the file before being discarded. Usually the hardware supports setting a "dirty page" bit which simplifies detecting modified pages. This allows user programs to easily edit files as if they were simple memory structures, while the OS assures that everything will be persistent once the program terminates.

Both Windows (since Windows 95) and Unix-derivatives (described in the POSIX standard) support memory mapped files. In the windows API, there are different function groups for memory mapping of files and manipulating virtual memory. The latter is done through the use of `VirtualAlloc` and `VirtualFree`, the former by calling `CreateFile Mapping` and `MapViewOfFileEx`. In Unix, the file mapping functions `mmap` and `munmap` handle both uses. A special file, `/dev/zero`, can be mapped to obtain virtual memory that is unconnected to a physical file. The `/dev/zero` file, which is actually a device, discards all writes and provides an infinite source of zeros when read.

All systems also provide functions to optimise memory handling. By communicating usage information about its virtual pages, an application can boost system performance. For example, it can tell the OS that some pages are no longer relevant, allowing the OS to reclaim them without saving the old contents. Or it can inform the OS that some pages will be accessed in the future, thus helping the OS in deciding which pages to keep and which to reclaim. In this last case, a prefetch operation could be started which could pre-fault the pages before they are actually referenced, hiding the large disk access latency.

The first Intel CPU that supported virtual memory, the Intel 386, was created in 1986. Support for virtual memory, through the `mmap` call, has been part of SRV4 UNIX since 1993. Windows 3.1, released in 1992, already had a basic virtual memory subsystem. Nevertheless, at present it is still uncommon for user applications to make use of the advantages virtual memory brings. Even if there is no benefit for the application itself, proper memory handling can relieve memory pressure and increase overall system performance.

4. The Evector class

In this section, a replacement class for the STL `vector` will be presented. Since the design of the STL, new insights showed that template programming enabled much more than just type abstraction. In fact, it has been proven that templates, together with other C++ language features, constitute a Turing-complete [13], compile time sublanguage. While there might be practical limits introduced by the compiler, there are no theoretical limits to what template metaprogramming can achieve. Inside the `evector` class, templates are used to optimise the code *at compile-time*, based on the distinct properties of the datatype stored inside the `evector`. Combined with the use of virtual memory, it enables the creation of a `vector`-compatible class that will outperform its predecessor in almost any situation.

4.1. Memory management

A consequence of the virtual memory model is that to actually *use* memory, two things are required: a free physical memory page, and a free virtual page. The amount of virtual memory is almost always a number of magnitudes large than the installed physical memory. For example, a 32 bit processor provides user processes with – depending on the operating system – two to four gigabytes of virtual memory. On 64 bit processors, which are becoming mainstream now, typically up to 256 *terabytes* of virtual memory are available. [2] By making use of this, it is possible to create a `vector`-like class that will never need to resort to the expensive allocate-and-copy strategy used in other implementations. The exact algorithms depend on the type of the stored data and on the amount of available virtual memory. The latter is determined by the CPU architecture, and to a lesser extent by the memory layout imposed by the Operating System.

4.1.1. Adaquate virtual memory

It is clear that in 64 bit environments there is no shortage of virtual memory, as every process has access to multiple terabytes of it. Hence it is not a problem for each `evector` instance to allocate a large amount of virtual memory. A good choice would be twice the amount of physical memory. The size of the allocated *virtual* memory region corresponds to the *capacity* of the `evector`. This way there will always be sufficient capacity, avoiding all need for reallocation. Note that physical memory only gets allocated the first time a page is actually used. So, unlike other implementations, the memory requirements of an `evector` are a function of its *size*, not its *capacity*. The end result is that appending an element to an `evector` always incurs a fixed cost for every newly used page.

[2] In the current generation, only 48 the 64 address bits are used.

4.1.2. Limited virtual memory

However, with 32 bit CPUs the picture is quite different: the current generation of 32 bit machines is reaching its limits. It is not uncommon to see a system with four gigabytes (or even more!) of main memory. A single application consuming multiple gigabytes of memory is no longer exceptional. For such an application, virtual memory becomes a scarce resource. Reserving too much of it will make future memory allocations fail, even if the system still has plenty of free physical memory.

Consequently it is not possible to overallocate virtual memory. This does not mean that virtual memory cannot be used to optimise memory handling. If there are free virtual pages following the memory block, these can be used to extend the block in place. If this is not the case, the whole block can be relocated to another address range capable of accommodating the increased size. This way, an implicit allocate-copy-free operation is performed *without actually copying data*. Since only the page tables need to be manipulated, data does not move in *physical* memory. However, for the application, it looks as if data really did move, as it is now only accessible at another address. Hence, by making use of all benefits virtual memory offers, a true move operation can be constructed.

4.1.3. Plain old data types

Still, there is a catch: we can only move POD (Plain Old Data) types. In short, the C++ standard states that POD types can be copied by using the `memcpy` function, so that the copy and its original will hold the same value. Non-POD types cannot be copied by merely copying their raw bytes. A reason for this is that the standard allows the compiler to add hidden data members to these types. For example, it is not uncommon for a class to contain internal pointers to its base classes. If its data is moved, these pointers become invalid. Another reason is that creating or moving an object by simply copying its raw data bypasses any user-defined constructors and destructors.

So, for non-POD types, another strategy is used: first an attempt is made to extend the mapping *in place*. This will succeed only if there are enough free *virtual* pages immediately following the mapping. If this is not possible – for example because of fragmentation of the virtual address space – there is no other option but to allocate a new virtual memory area and to copy all data.

However, even in this situation virtual memory management offers a significant benefit. While it is impossible to avoid the construction and destruction of the stored (non-POD) objects, it *is* possible to avoid the $3n$ memory requirement!

Every time an object is moved to its new location, its old memory can be discarded. However, traditional memory allocators only support freeing complete blocks. Virtual memory, on the other hand, has a granularity of a single page. Consequently, it is perfectly possible to discard a single page from a multi-page range. It even offers the possibility to only free the page *contents* without actually *freeing* the memory block. This is done by manipulating the link between the virtual and physical page.

As such, the following strategy is used during the move operation. Every time a certain number of pages is copied from the old area, the OS is informed that these pages are no longer needed. This way, when page faults occur – caused by writing to the newly allocated area – the OS immediately has good quality free pages available: the pages just copied from. It does not even have to zero them first (which is a security measure to avoid inter-process snooping) as they were last used by the very same process they are now needed in. Because of this, the total physical memory requirement of the application only increases slightly during the copy operation.

This technique reveals its full potential when physical memory pressure is high. In such a situation, chances are small that the OS has enough free pages to satisfy the memory demands of the copy operation. So, every time a newly allocated page is written to, the OS will have to find a used (physical) page it can expel to disk in order to reuse the page to back up the new virtual page. In general, when selecting pages to reclaim, an attempt will be made to pick the least recently used one. This ordering will try to keep the source pages of the copy in memory as long as possible. However, these source pages will never be needed again! So, not only does a traditional copy operation require far too much memory, it also causes the OS to expel the wrong pages, causing even more performance degradation.

4.2. *Type optimisation*

The performance of the `evector` class strongly depends on the type that is stored inside, especially in 32 bit environments. When this type can be relocated by a simple bitwise copy, a constant time extension can be guaranteed. If not, all elements need to be copied on every reallocation.

An `evector` uses *type traits* [3,4] to determine *at compile time* if a type is movable. The same traits principle is used to generate optimal code when constructing or destructing elements. If found that a type contains only raw data (for example it is a struct containing a number of integers), this knowledge is used. Instead of constructing elements one by one, a fast `memfill` routine is used to initialize all elements at once (if initialisation is requested by the user). Copying likewise uses the fast `memcpy` function, and destruction becomes a no-operation.

However, because of the lack of adequate reflection capabilities in the C++ language, not all compilers support automatic detection of POD types [5]. When compiler support is not available, a conservative decision can be made to classify all non-scalar types as non-movable. However, the user can always override the automatic detection and specify the type properties by hand. This is done by specialising the traits template for the type in question. The example code in listing 1 declares that the type `ExampleStruct` can be moved but should still be constructed and destructed. When this type is used in an `evector`, code will be generated that allows reallocating data without copying. The detection logic is fully based on template metaprogramming. The second specialisation in listing 1 is an example of this. It declares that every built-in array inherits the type properties of its base type.

Listing 1 Overriding Type Properties

```
namespace Traits {
// Define ExampleStruct to be
movable
template <>
struct TypeTraits<ExampleStruct> {
 enum { pod_level = PODL_CANMOVE; }
};
// Arrays are movable if the
elements they contain are movable
template <typename T, unsigned int
Count>
struct TypeTraits<T[Count]> {
 enum { pod_level = TypeTraits<T>:
:pod_level };
};
}
```

4.3. *Improving safety*

4.3.1. *Taking addresses*

Although the standard `vector` class allows taking the address of its elements, this is usually not a good idea. Every action modifying the size of the vector could cause it to reallocate its contents, rendering all addresses void, and creating bugs that are difficult to track. The `evector` class tries to improve this by using a policy based design [6] which empowers the user to decide if taking addresses is allowed or not.

It is not straightforward to prevent leaking internal addresses, since the class has to provide an `operator []`-method that returns a reference to its elements. This enables element modification and is necessary to be STL-vector compliant. By using a *proxy-pattern* [7], an `evector` can determine if the client code is reading, writing or taking addresses and properly react on it. As this proxy gets optimized out of existence by the compiler, there is almost no overhead during element access. Another implemented policy avoids the problem entirely by simply disallowing reallocations that need to move the data. In this case growing the vector is possible, as long as all element addresses remain valid.

4.3.2. Detecting out-of-bound accesses

By manipulating access policy on a page level, incorrect usage of an `evector` can be detected. When making sure that unused pages have no read or write permission bits set, illegal accesses can be detected. Also, an inaccessible guard page could be set up before the first element. This way illegal read or write attempts can be flagged, even when they bypass the normal `vector` access methods (for example by using pointer arithmetic). However, as the granularity of the access specifiers is limited to one page, not all out-of-bounds accesses can be detected. On the other hand as this check occurs in hardware there is no extra overhead on element access.

4.4. Custom allocators & realloc

When using the GCC compiler suite – depending on the platform – the `malloc` function already uses the `mmap` technique to allocate large memory blocks. Also, for these large blocks, the corresponding `realloc` function can use `mremap` to extend the mapping when possible. However, the `realloc` function is almost useless in C++. It is clear that `realloc` is inherited from C, which does not have class types and consequently does not need constructors or destructors. As such, `realloc` combines resizing the memory block *and* moving its contents – by bitwise copy – to the new block if necessary. There is no possibility to limit reallocation to *in place* extensions. Because of this, it cannot be used on memory blocks containing non-POD types. This strongly limits its usability in C++.

In Section 2 the allocator template parameter of the STL `vector` was mentioned. It seems a logical choice to improve `vector` memory handling by writing a better memory allocator for it. Sadly enough, this is not feasible. The STL allocator interface does not include a `realloc` method. It only declares functions for allocating and freeing a memory block. However, SGI has its own allocator interface[11] which *does* include a realloc function. Nevertheless, it suffers from the same problem as the equally named C function, causing its use to be severely limited in object-oriented environments. More information on writing custom memory allocators can be found in [10].

4.5. Extending clear()

To allow applications to further enhance their memory usage, the `clear`() method of the `evector` class was extended. As its STL counterpart, it destroys all of the elements in the container, but in addition the `madvise` or `VirtualAlloc`(MEM_RESET) function is used to tell the OS that the memory involved will not be needed again. This makes the corresponding physical pages act as if they would be *soft referenced* [12] in Java. As long as there is sufficient memory available, they will not be removed. However, when memory pressure is high, these pages can be decoupled and reused elsewhere.

So, clearing an `evector` frees all of its memory *but only if needed*. This is a remarkable improvement compared to the traditional vector, where the page contents – which will never be needed again – first need to be written to disk before the pages can be reused. And, when the vector is filled again, that old data needs to be reread from the swap space only to be overwritten!

4.6. Caveats

4.6.1. Memory overhead

The algorithms used in the `evector` are not always optimal. Unlike the traditional `vector` class, where memory overhead is relative to its size, an `evector` has a fixed average overhead of half a page. As a page usually is at least 4096 bytes it is clear that for small amounts of data the traditional allocation scheme is better suited. Because of this, a future version of the `evector` class will incorporate a hybrid allocation model. When its *capacity* (in bytes) is smaller than a certain size, traditional allocation methods will be used. If its capacity is large enough, the algorithms discussed in this paper will be applied.

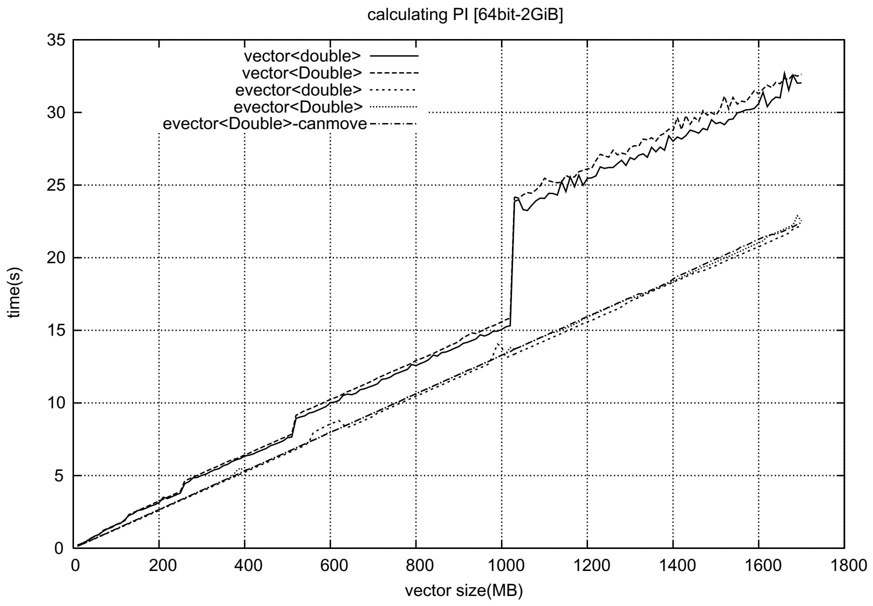

Fig. 2. Application performance.

4.6.2. TLB efficiency

A second problem is related to the TLB. While using memory mappings for large allocations is not a problem – even the `malloc` implementation of the GNU Compiler uses `mmap` to satisfy large requests – it does decrease the TLB hit rate. Creating a huge number of memory mappings may have an adverse effect on memory latency and on OS functions that need to traverse the page tables (an example of such a function is the `fork` system call).

Also, functions manipulating the page tables need to make a system call in order to switch to kernel mode. This transition is not without a cost. In addition, changing the page tables requires a TLB flush on most platforms. Otherwise, stale mappings could still be present in the cache. This again increases memory latency the first time a page is accessed. For small `evector` sizes, these disadvantages could outweigh the benefits.

This is another reason why a hybrid approach will have the best performance.

4.6.3. Multi-dimensional vectors

Multi-dimensional vectors that can only grow in one dimension can be treated as one dimensional vectors in which each element contains the remaining fixed-size dimensions. True multi-dimensional vectors, although not optimal, can be constructed as a vector of vectors.

5. Results

In this section, the new `evector` class is tested in real-life situations and compared to the STL `vector` class.

5.1. Application-level benchmark

A small test application was written in order to test the evector implementation in a realistic situation. It uses a Monte Carlo method to calculate the ratio between the area of a square and the area of its inscribed circle. This ratio is subsequently used to estimate π. First, all random numbers are placed in the collection by using `push_back`. These are later reused to perform the area calculations. All tests were run on a 64 bit Opteron system equipped with two gigabytes of memory. The results can be seen in Fig. 2. The horizontal axis shows the size of the vector, while on the vertical axis the execution time of the program is plotted. The type parameter `double` denotes the built-in double precision type. The `Double` type is a class wrapper for `double`. As it has a custom constructor, it is not a

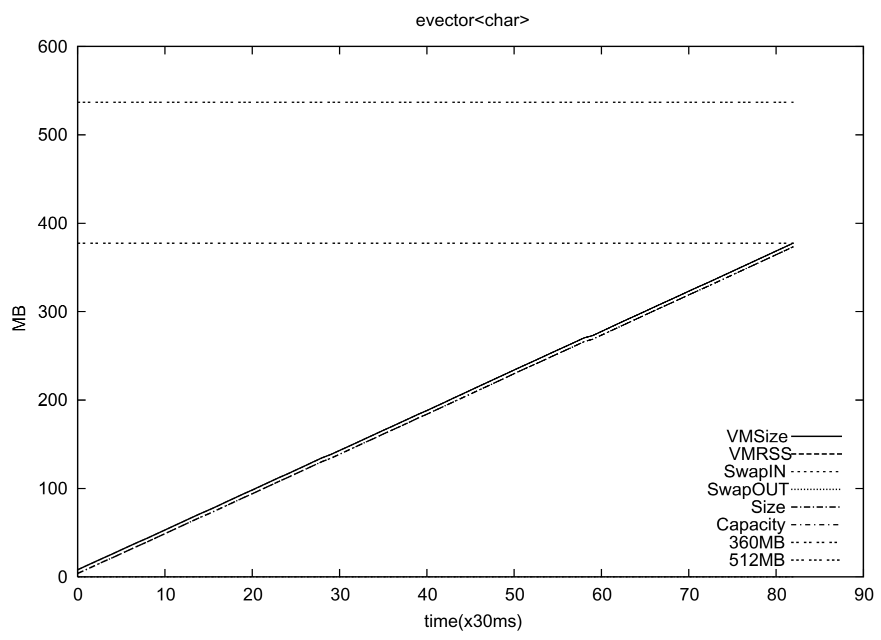

Fig. 3. `evector`, movable datatype.

POD type and will not be marked as movable. The suffix `can-move` indicates that the technique demonstrated in listing 1 was used to communicate that the type can be moved. A number of observations can be made:

- The overhead caused by the reallocation of the `std::vector` class can clearly be seen at 256, 512 and 1024 MB.
- Even though the test system has two gigabytes of memory, the system starts swapping the moment the vector size exceeds 1024 MB. This causes a large increase in execution time (and also decreases total system performance).
- As expected, no `evector` instance ever needs to reallocate as a 64 bit system has more than enough free virtual memory. The graph shows that for `evector` instances, extension happens in constant time.

5.2. *Detailed analysis*

In order to better illustrate the memory behaviour of both implementations, a monitor utility was written. It is embedded into a test application, and samples memory usage every 30 ms. Its information is collected from various system functions and the `/proc` file system. The monitor measures the total virtual memory size, the amount of virtual pages present in main memory, the moving of pages between swap and main memory and the current vector size and capacity. Another virtual memory feature, *page locking*, is used to pin the monitor application into main memory. This guarantees that the monitor will always be able to sample its data. Without these precautions, measurements could be distorted if pages containing monitor code are swapped out.

All tests were executed on a 32 bit machine with 512 MB of main memory. The test application simply fills a vector up to a predefined size (360 MB) by repeatedly appending elements at the back.

5.2.1. *Evector, movable datatype*

Figure 3 demonstrates the results for an `evector` holding a movable type. The graph nicely shows that the effective memory usage (VMSize) closely follows the size of the vector, which is at any time equal to its capacity. Increasing capacity is only done when required, and is quick and without memory overhead. Pages are never evicted to the swap area, which is to be expected as the total vector size (360 MB) is significantly less than the amount of available memory (512 MB).

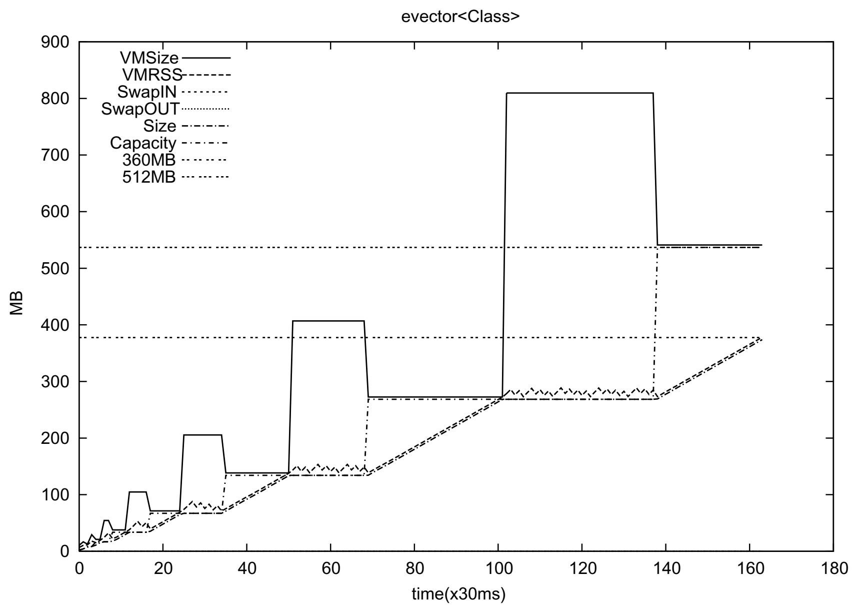

Fig. 4. `evector`, class datatype.

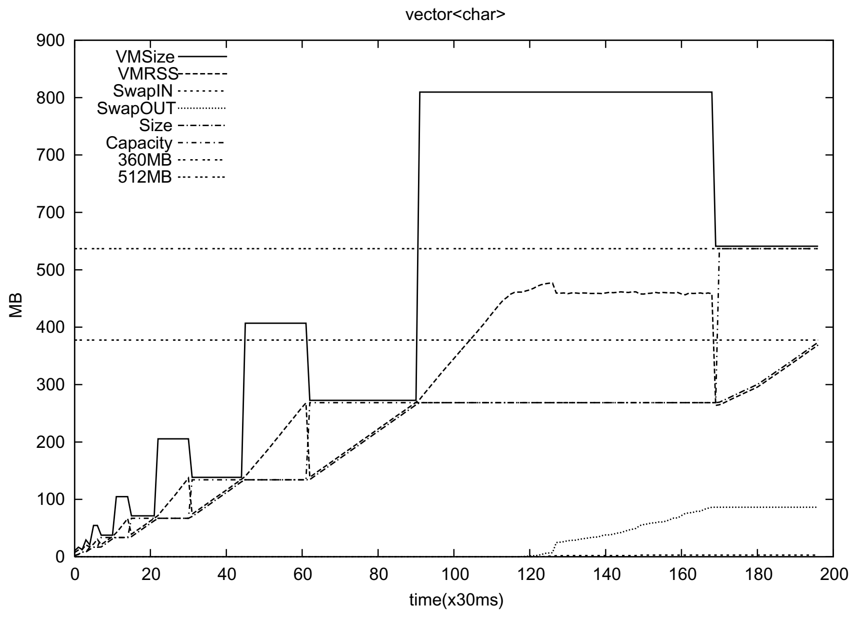

Fig. 5. `vector`, movable datatype.

5.2.2. Evector, non-movable datatype

When the `evector` stores a type that cannot easily be moved, another strategy is used. As with the normal `vector`, capacity is doubled when needed but the effective memory usage (VMRSS) is *never* (*much*) *larger than size of the elements it contains*. Figure 4 clearly demonstrates that although the system only has 512 MB of main memory, allocating more than 800 MB of *virtual* memory (VMSize) does not impede the OS.

Note that on 64 bit systems it does not matter if the stored type is movable or not. Because of the huge amount of available virtual memory on those systems, an `evector` instance will always overallocate and reallocation will

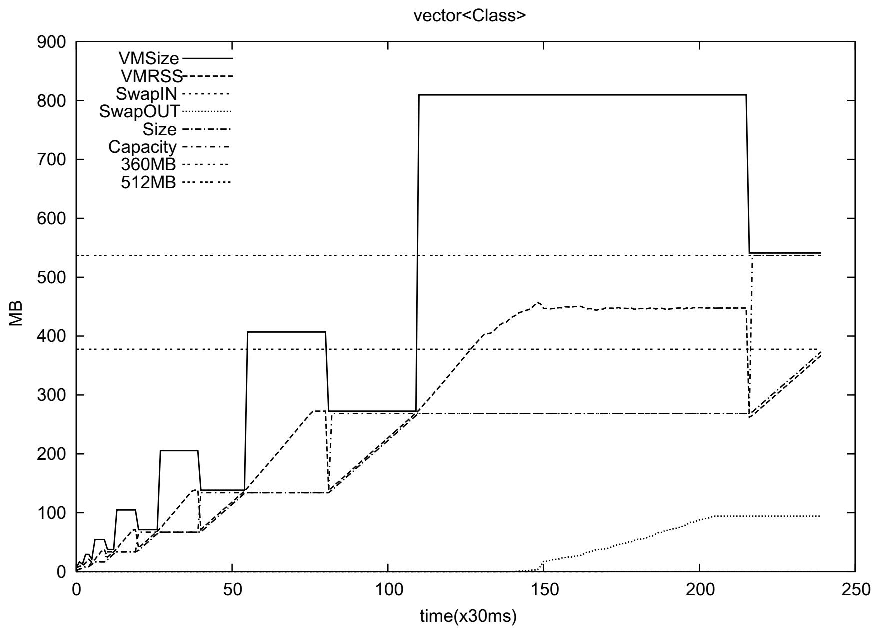

Fig. 6. `vector`, class datatype.

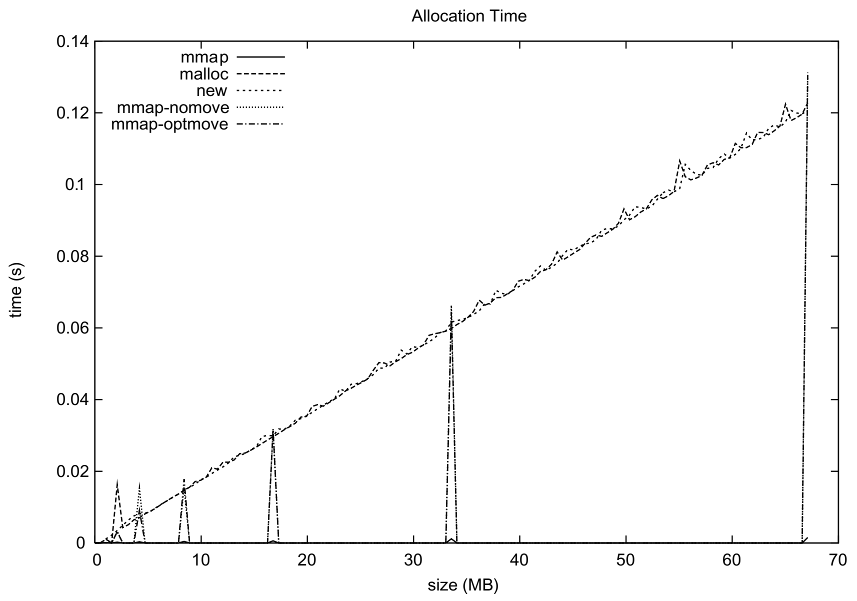

Fig. 7. Comparison of different memory allocators.

never occur.

5.2.3. *Vector, movable & non-movable*

The STL `vector` class does not adapt its behaviour to the type it holds. Consequently, Figures 5 and 6 are similar. Note that during the reallocation phase, the actual memory usage (VMRSS) is twice the size of the actual data stored within the vector, only to fall back to its normal level once reallocation is finished. As soon as the vector size exceeds 256 MB, an attempt is made to use *up to 512 MB* of memory! This is of course not possible, so the

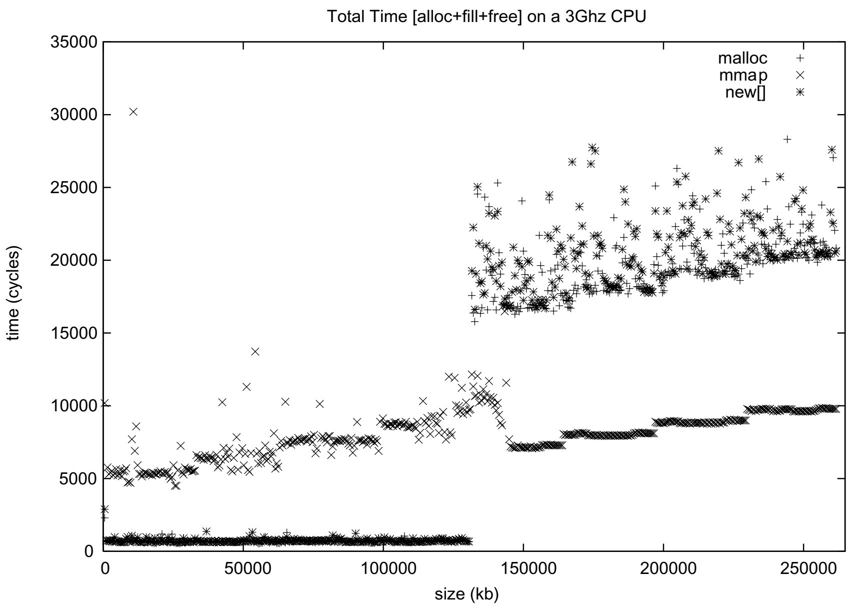

Fig. 8. Allocate & free performance.

system starts to swap out pages at a high rate, causing large delays and loss of reponsiveness. Also note that the runtime of the program more than doubled compared to the `evector` code. The situation would even be worse on a system that does *not* revert to `mmap` for large allocations (as the GNU C library does). In this case, more than 768 MB of memory would be required when creating a vector of little over 256 MB.

5.3. *Resize performance*

The allocation technique alone was also benchmarked. In this test, the following allocator techniques were compared:

malloc The standard C memory allocator. It uses the functions `malloc/realloc/free`. Modification of the base address is allowed.

new The standard C++ memory allocator. As there is no corresponding “resize” function, the normal practice of allocating a new buffer and copying the old data is used. (`new/delete`)

mmap The most flexible virtual memory technique. Modification of the base address is allowed. (`mmap/mremap/munmap`)

mmap-nomove As above, but no base address modification is allowed. When in-place extension is not possible, the same allocate-copy operation is used as is the case with “new”.

mmap-optmove As above, but this time the copy operation uses hints when copying from the old to the new location. Every time a block of one megabyte is copied, the OS is informed that its contents are no longer relevant.

For each allocation strategy a block of memory was created. Next, the block was continuously extended, and the time required for this operation was recorded. After successful extension, a byte was read from every page. The reason for this last test is to include the first-access overhead of pages allocated in a lazy fashion.

Figure 7 displays the time for both resizing the memory block and touching its pages. It is clear that it is dominated by the time needed to copy data from the old block to the new block. When the mmap-allocator is allowed to change the base address of a block, this copy is never needed and hence it performs much better than `malloc` or `new`. The peaks for `mmap-nomove` and `mmap-optmove` occur when the in-place extension failed, and a copy was needed. Note that they both perform the same, which shows that the overhead of `madvise` is negligible.

This test was executed on a machine with 2 gigabytes of memory and a pentium4 processor. The compiler used was gcc 3.3.6, with kernel version 2.6.13.

Table 1
System call overhead

	Execution time (ns)		
operation	min	avg	max
function call	30	32	46
`getpid`	165	174	308
`madvice`	281	293	398

5.4. Allocate & free

In this test the time needed to allocate, touch and free a memory block of a certain size is measured. Because of the small time scales involved, all times were measured using a cycle counter present in most modern CPUs. The cycle counter is a register which is incremented on each clock tick. It can be used to construct a highly accurate, extremely high resolution ($\frac{1}{3\times10^9}$ s) timer.

Figure 8 shows why a hybrid approach has most advantages. For small block sizes, the traditional allocators perform much better. However, due to practical reasons these fast algorithms cannot be used for all block sizes. The discontinuity around 131000 bytes for `malloc` and `new` is a consequence of this. For blocks larger than this size, different, slower algorithms are used.

5.5. System call overhead

An attempt was made to determine the extra overhead of switching to kernel mode. This happens every time the page tables have to be modified. We compared the minimum, maximum and average time needed to perform a normal function call, a simple system call (`getpid`) and the system call used to discard virtual page contents. The test system was a pentium4 at 3 Ghz running kernel 2.6.13. Table 1 lists our findings. When looking at the minimum time, we can see that a simple system call is about five times slower than a normal function call. The `madvise` system call, used when clearing an `evector` is almost ten times slower. However, all of these operations are still at least an order of magnitude faster than for example copying a page, or writing a page to swap space.

6. Conclusions & future work

6.1. Future work

Virtual memory could also be used to implement extremely fast sparse structures (like sparse matrices or hash tables). In some cases the speed benefits could be much larger than the memory overhead caused by the page granularity. However, this will most likely require 64 bit environments as large amounts of virtual memory are needed.

In situations where copying cannot be avoided, further optimisation is possible, although it requires type modification. By providing a special “swap” constructor, composite types can be “moved” to a new memory location without copy-constructing all of theirs class data. This practice has been described in [9].

On platforms that support them, *large pages* could be used to increase the efficiency of the TLB cache. This way, by allowing a single entry to describe more virtual memory, the effective cache size is increased. More research is needed to make sure this is really beneficial, as large pages also increase the granularity of the memory and could lead to increased overhead.

6.2. Conclusions

This article demonstrated that by making full use of the virtual memory facilities provided by the OS, increased application and system performance can be obtained. The `evector` class, a drop-in `vector` class replacement can – by making careful use of new techniques when appropriate – outperform its STL counterpart in every situation, while at the same time improving system-wide memory efficiency. The STL `vector` (and more general the STL library), indisputably revolutionary at the time of its introduction, needs to be updated to make full use of new language paradigms.

As most advantages offered by the virtual memory system can deteriorate gracefully – for example, the `madvise` call is only a *hint*, and the move capability can be simulated by a copy – similar functionality should be offered by the base language. This way, programs can make use of this in a portable way and can achieve much better performance and efficiency on platforms supporting it.

6.3. Source code

The `evector` source code can be obtained by sending a mail to Dries.Kimpe@cs.kuleuven.be.

References

[1] A. Stepanov and M. Lee, The Standard Template Library, HP Technical Report HPL-94-34, February 1995.
[2] International Standard: Programming Language – C++, ISO/IEC 14882:1998(E), 1998.
[3] N. Myers, *A New and Useful Template Technique: Traits*, C++ Report, June 1995.
[4] J. Maddock and S. Cleary, *C++ Type Traits*, Dr. Dobbs Journal, October 2000.
[5] *Boost C++ Libraries; Boost.TypeTraits*, http://www.boost. org.
[6] A. Alexandrescu, *Modern C++: Generic Programming and Design Patterns Applied*, 2001.
[7] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns*, Addison Wesley, 1995. ISBN 0201-63361-2-(3).
[8] B. Jacob, Virtual Memory Systems and TLB Structures, in: *The Computer Engineering Handbook*, V. Oklobdzija, ed., CRC Press: Boca Raton FL. 2002.
[9] A. Alexandrescu, *Generic<Programming>: Typed Buffers*, C/C++ Users Journal C++ Experts Forum, August 2001.
[10] M. Austern, *The Standard Librarian: What Are Allocators Good For?*, C/C++ Users Journal C++ Experts Forum, December 2000.
[11] *SGI STL Allocator Design*, http://www.sgi.com/tech/stl/alloc.html, 1994.
[12] Sun Developer Network, *The Java API Specifications*, http://java.sun.com/j2se/1.4.2/docs/api/java/lang/ref/SoftReference.html.
[13] M. Böhme and B. Manthey, The Computational Power of Compiling C++, *Bulletin of the EATCS* **81** (2003), 264–270.
[14] A. Brodnik, S. Carlsson, E.D. Demaine, J.I. Munro and R. Sedgewick, *Resizable Arrays in Optimal Time and Space, Proceedings of the 6th International Workshop on Algorithms and Data Structures* (*WADS'99*), Lecture Notes in Computer Science, Vol. 1663, Vancouver, British Columbia, Canada, August 11–14, 1999, 37–48.