Performance of C++ Bit-vector Implementations

Vreda Pieterse
FASTAR Research Group
Dept. of Computer Science
University of Pretoria
South Africa
vpieterse@cs.up.ac.za

Derrick G. Kourie
FASTAR Research Group
Dept. of Computer Science
University of Pretoria
South Africa
dkourie@cs.up.ac.za

Loek Cleophas
Department of Mathematics
and Computer Science
Eindhoven University of
Technology, The Netherlands
loek@loekcleophas.com

Bruce W. Watson
FASTAR Research Group
Dept. of Computer Science
University of Pretoria
South Africa
bruce@brucewatson.com

ABSTRACT

This paper describes an experimental study to compare the performance of various dynamically resizable bit-vector implementations for the C++ programming language.

We compare the `std::vector<bool>` from the Standard Template Library (STL), `boost::dynamic_bitset` from Boost, `Qt::QBitArray` from QT Software, and BitMagic's `bm::bvector<>` with one another.

We also compare `std::vector<char>` from the STL with these because it is a dynamically resizable vector implementation that has been suggested to be an acceptable alternative for `std::vector<bool>`.

We describe the test data and the methods that were applied to measure memory use and processing time. This lays a foundation for comparing other parts of the different C++ libraries.

The results are presented and discussed in terms of the differences in the implementations of these data structures. Although the results reported in this article is specific to the mentioned C++ libraries, the techniques used to measure and compare the performance of the different libraries go beyond C++ bit-vectors and may be used more generally.

Categories and Subject Descriptors

E.1 [**Data Structures**]: Arrays; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Performance evaluation*

General Terms

Measurement, Performance

Keywords

C++; bit-vectors; benchmark; Standard Template Library

1. INTRODUCTION

There are many applications that can apply bit-vectors to implement functionality. Application areas for bit-vectors include graphics, hardware programming, encryption, networking and databases. For C++, there is a number of different bit-vector implementations available to choose from. This article takes a look at a few available implementations of bit-vectors of which the size can be specified and altered at runtime, and compares their performance using data sets of varying sizes.

This type of comparison has probably been performed many times in industrial settings. However, they are likely to be conducted in a non-formal way. A meticulous approach has been taken to measuring time and space performance and this approach is outlined in this article. It is hoped that this will not only enhance the credibility of our cross-comparative results, but that it will also serve as a model for future benchmarking endeavours.

Section 2 acquaints the reader with the different bit-vector implementations included in this study, while Section 3 describes the algorithm that was used to benchmark these implementations. Section 4 discusses details of the experimental environment regarding the platform that was used and the software that was implemented to conduct this experimental study. Section 5 describes how the data was gathered and analysed, and shows the results. The concluding Section 6 offers some insights about the bit-vector implementations used, based on their observed performance differences.

2. BIT-VECTOR IMPLEMENTATIONS

This section discusses some, easily obtainable, implementations of bit-vectors that support bit-wise operations on large bit-vectors of which the size may be varied at run time. Since we are interested in bit-vectors that are dynamically extendable, we do not consider those whose size is fixed at compile time such as the STL's `std::bitset`.

2.1 std::vector<bool>

In the C++ 03 standard [13] the `vector` class template includes a special template specialisation for the `bool` type. This specialisation is provided to optimise for space allocation. Its bit-manipulation functions are limited. The only member function that was added to those already in `std::vector<T>` is `flip()`. This member function is overloaded and can thus be used to invert all the bits of a vector at once or to invert a single specified bit. There is no `set()` or `reset()` as would be expected in a data structure that is implemented for the purpose of doing data manipulation at bit level. It also does not provide for any bit-wise operations such as `operator|` or `operator>>` on whole vectors. However, the `operator[]` is provided. This can be used in combination with the `flip()` operation on a single bit to implement the desired results. This forces the programmer to iterate through the bits in the vector to achieve bitwise operations.

Meyers [17] advised against its use. According to Meyers "its elements are not individually addressable and hence it does not meet the requirements of a container", despite the fact that the `operator[]` is provided. Furthermore, Meyers is of the opinion that its iterators do not meet the iterator requirements. This has broken user code in the field in mysterious ways [24]. The space optimisation intended by the implementation of `std::vector<bool>` exacts a hefty toll: a quirky interface, speed overhead, and incompatibility with the STL [14].

Following the issue that was filed in 1996 regarding the above mentioned flaws of `std::vector<bool>`, it was proposed in 2007 that it be deprecated or removed [8]. However, in November 2009 this issue was labeled as *"not a defect"* (NAD) [12], in other words, `std::vector<bool>` will remain part of the C++ Standard for the foreseeable future. Furthermore, a proposal that was made in 2006 to replace it with an alternative [1], was re-opened as a feature request in September 2009 [11], albeit no longer with the intention to replace it. This was pre-emptively filed as *"NAD Future"*, in other words, the addition to the standard of the proposed `std::dynamic_bitset` that addresses all the mentioned shortcomings, is postponed to an unspecified future date and it will most likely not be added in the near future. A dynamic bit-vector that closely resembles the ill fated proposed `std::dynamic_bitset` can be found in the Boost library (see Section 2.3).

Despite its obvious shortcomings and evidence that it is not widely used [8], `std::vector<bool>` was included in our experiments for the sake of completeness.

2.2 std::vector<char>

`std::vector<char>` is a dynamically resizable vector of addressable characters that can easily be cast to `bool`s. Owing to the controversy around `std::vector<bool>`, programmers were advised to use `std::vector<char>` instead [14, 17]. When using this implementation one does not have direct access to the rich set of bit-wise manipulations such as those offered in dedicated bit-vector implementations. However, the specifications of the required operations are fairly straightforward. `std::vector<char>` provides an interface that is similar to bit-vector implementations on the concept level. The `std::vector<char>` implementation was included in the experiments because it is widely accepted as an alternative for `std::vector<bool>` and because we were particularly interested in comparing its performance with that of `std::vector<bool>` to determine if it is indeed a viable alternative.

2.3 boost::dynamic_bitset

Boost is a collection of free, open-source, peer-reviewed C++ libraries intended for eventual inclusion in the C++ STL. The project was originally proposed on May 6, 1998 by Beman Dawes [6]. Proposed libraries are accepted into Boost only after undergoing a formal review, during which members of the Boost mailing list may comment on their evaluation of the proposed library [7].

The `boost::dynamic_bitset` class, despite being called a bitset, is a class for manipulating bit-vectors. It has been included in the Boost library since October 2002 [21]. It provides accesses to the value of a bit-vector's individual bits via an `operator[]` and provides all of the bit-wise operators that one can apply to built-in integers, such as `operator&` and `operator<<` on whole bit-vectors as well as all relevant operators with individual bits. The number of bits in the vector is specified at runtime via a parameter to its constructor.

`boost::dynamic_bitset` closely resembles the proposed dynamic bit-vector, `std::dynamic_bitset`.

2.4 Qt::QBitArray

Qt is a C++ class library to support the development of cross-platform applications with graphical user interfaces (GUIs). Qt is primarily developed and maintained by the developers at Qt Software, a unit within Nokia. It is licenced under both open source licenses (LGPL and GPL), as well as a commercial license. Therefore, Qt can easily be used for a wide variety of project types [19].

Apart from providing a rich set of GUI classes, Qt also redefines almost all C/C++ standard data structures and provides for a wide variety of new ones. Most of the class implementations in Qt that replace classes of the STL, have similar interfaces to the corresponding classes in the STL. Sometimes they add some functionality, but they are mostly there to simplify their usage in collaboration with the sophisticated GUI classes in Qt. The Qt class that corresponds with the proposed dynamic bit-vector `std::dynamic_bitset` that was mentioned in Section 2.1 is `Qt::QBitArray`. It is a special bit-array that provides functionality to access individual bits, and perform bit-operations on them with operations such as `operator[]`, `setBit()` and `toggleBit()`. It also provides functionality to perform bit-operations on entire arrays with operations such as `operator&`, `operator|` and `operator^`[4]. However, it does not support bit-wise shift operations.

The popularity of C++ as a programming language combined with the fact that cross-platform applications with GUIs have become the norm, contributed to wide adoption of Qt as an application development environment of choice. Qt is used to implement prominent applications such as KDE, Google Earth, Skype, Adobe Photoshop Album and VirtualBox. For these reasons we have included Qt in our comparison.

2.5 bm::bvector<>

BitMagic is an Open Source, free C++ library designed and developed to implement efficient platform independent bit-vectors[16]. The founders of BitMagic identified the need to implement more sophisticated data compression in the internal representation of bit-vectors. Their main aim is to provide a tool to implement data-intensive technologies such as streaming media and large database applications. It was launched in 2002.

The implementation of BitMagic's `bm::bvector<>` is a dynamically sized bit-vector that applies several types of on the fly adaptive compression. Owing to the sophisticated internal representation, it is not required that the programmer specify the size of the bit-vector that is represented by a `bm::bvector<>`. The actual memory used by its implementation is adapted to best suit the data at hand. Because the size of a `bm::bvector<>` is internally managed, no public operation to resize it is provided. The `bm::bvector<>` has a protected member function that is used by other public operations to automatically maintain

```
{$R \subseteq U \times U$}
{$n = |U| \in \mathbb{N}^+$}
var    $M_0 \in \mathbb{B}[n,n]$
var    $M_1 \in \mathbb{B}[n,n]$

for $i, j : i, j \in \mathbb{N}_n$  $\rightarrow$  $M_0[i,j], M_1[i,j] := u_i R u_j,\ 0$ rof
do $M_0 \neq M_1$  $\rightarrow$
    $M_1 := M_0$
    for $i : i \in \mathbb{N}_n$  $\rightarrow$
        for $j : j \in \mathbb{N}_n$  $\rightarrow$
            if $\neg M_0[i,j]$  $\rightarrow$  skip
            [] $M_0[i,j]$  $\rightarrow$  for $k \in \mathbb{N}_n$  $\rightarrow$  $M_0[i,k] := M_0[i,k] \vee M_0[j,k]$ rof
            fi
        rof
    rof
od
```

Figure 1: Baker's algorithm for calculating transitive closure

optimal size.

`bm::bvector<>` provides by far the richest set of operations when compared to the other data structures discussed in this article. All bit-wise operations can be performed on whole arrays, subsections of arrays and individual bits. It also provides iterators.

The advances in hardware that in turn allow an ever growing amount of data to be processed, resulted in the processing of very large data sets to be commonplace. The ability to optimise compression of these huge data sets has become a necessity. Experiments with the `bm::bvector<>` were therefore included. We expected that its performance would compare favourably to the corresponding data structures in the C++ STL and the Boost library.

3. THE EXPERIMENTAL ALGORITHM

The purpose of this article is to compare the performance of different implementations of dynamic C++ bit-vectors in a situation that resembles as close as possible a *real world* application. Since some of the data structures included in our study do not support bitwise shift operations, we avoided their use. For this reason we selected an algorithm for calculating the transitive closure of a binary relation represented as a vector of bit-vectors. The algorithm requires storage and manipulation of multiple bit-arrays and uses a representative set of the bit-wise operations found in all the implementations of bit-vectors we included in our study. During the initialisation phase it makes heavy use of set and reset operations on individual bits. During the calculation phase it relies on intensive inspection of individual bits, repetitive comparison of whole arrays, as well as recurring execution of bitwise-OR operations on whole arrays. Although no bitwise-AND operations are required in our chosen algorithm, we deem the algorithm adequate based on the assumption that the performance for bitwise-AND and bitwise-OR operations would be similar.

The transitive closure R^+ of a relation $R \subseteq U \times U$ is the smallest subset of $U \times U$ such that R^+ is transitive and $R \subseteq R^+$. There are many known algorithms for calculating transitive closure, of which the one proposed by Warshall [26] is probably the most widely used. However, a primitive algorithm described by Baker [2] was deemed to be more suitable for our experiment. In pilot experiments with different algorithms to calculate the transitive closure of a relation, the performance of Baker's algorithm was less sensitive to variations in the density and distribution of 'on'-bits of its input data than some of the other algorithms to calculate the transitive closure of a relation. This is the case because this primitive algorithm systematically works toward the solution without taking any advantage of any knowledge about the underlaying data. This algorithm is listed in Figure 1 using the Guarded Command Language (GCL) defined by Dijkstra [9]. In this listing $\mathbb{B}[n,n]$ is the set of all $n \times n$ Boolean matrices. If $M \in \mathbb{B}[n,n], M$ has n rows and n columns numbered $0, 1, 2, \ldots (n-1)$. $M[i,j]$ refers to the bit in row i and column j of M. The body of the loop in this algorithm is essentially three levels of nested loops, each requiring n operations, yielding a complexity of $O(n^3)$. It can be shown that the algorithm will terminate after at most $log_2 n + 1$ iterations. Therefore, the complexity of the algorithm is $O(n^3 log_2 n)$.

4. EXPERIMENTAL ENVIRONMENT

4.1 Language and Compiler

The experimental algorithm was implemented in C++. A separate implementation was created for each of the data structures mentioned in Section 2.

Each individual C++ implementation of the algorithm was compiled using g++ from the GNU Compiler Collection (GCC) for compiling C++ programs as it is included in the Minimalist GNU for Windows (MinGW) Version 3.81 with patches from Eli Zaretskii (March 2008) [23].

GCC provides optimising options which aim to increase the speed, or reduce the size, of the executable files it generates. In each case, the compilation of our code was done with the maximum level of optimisation enabled by using the `-O3` option.

4.2 Hardware

The experiments were executed on an Intel Centrino Duo laptop with 1 GiB RAM. The processors are Genuine Intel(R) CPU's T2300 @ 1.66 GHz. Its operating system is Micosoft Windows Vista Business Version 6.0 (Build 6001: Service Pack 1).

Prior to running the experiments all applications were closed and all memory resident processes that are not needed, were stopped. Further, the Task Manager was used to manually set the process affinity of all the remaining background processes to one of the CPU's and

Table 1: Number of memory blocks needed to execute the program implementing each of the data structures

		Δ +								
Data Structure	Δ	327	328	329	330	331	332	333	334	335
`std::vector<char>`	0	61%	38%	1%						
`std::vector<bool>`	7	66%	29%	5%						
`boost::dynamic_bitset`	9	66%	32%	2%						
`bm::vector<>`	12	63%	34%	3%						
`Qt::QBitArray`	275	1%	15%	11%	9%	14%	16%	22%	9%	3%

that of the benchmark system to the other CPU. Because the experiments tended to utilise the selected processor to maximum capacity, risking overheating and non-optimal performance, we swapped the choice of CPU for the background processes and our benchmark system from time to time.

4.3 Software

A system was designed and implemented to perform these benchmarks. To support repeatable experiments, the system can generate or import data sets. These data sets are retained in a repository and can easily be accessed for subsequent executions. The graphical user interface for this system was created using the non-commercial version of Qt 3.2.1. Our aim was to be able to record the performance metrics of a selected implementation of an algorithm. This was achieved by creating a distinct process that is dedicated to the execution of the specific implementation of the algorithm with a known data set selected from the repository of data sets.

Previously the time stamp counter (TSC) was widely used to perform high-resolution timing. This is no longer the case since technology that changes the frequency of the CPU is in use in many high-end desktop PCs [25]. Furthermore, when executing on systems with multiple CPUs, and hibernating operating systems, the TSC cannot be relied on to provide accurate results. For our benchmarking we decided to take a bold step and implement an alternative strategy that would not try to eliminate these effects, but rather incorporate them. In our opinion this technique, described here, enabled us to implement a platform independent benchmarking system that produces realistic measurements.

A `Qt::QTime` object contains a clock time expressed as number of hours, minutes, seconds, and milliseconds. A time stamp can be retrieved by reading the current time from the system clock in 24-hour clock format using the static `currentTime()` function. The execution time of a process is calculated by measuring a span of elapsed time with the help of `start()`, `restart()`, and `elapsed()` functions provided in the `Qt::QTime` class.

Time is measured in milliseconds using the `msec()` function. The execution time of the process is measured by signaling the moment all data has been read from the input buffer as its starting point, and the moment the first output character is produced as its finishing time. According to the Qt documentation, the accuracy of these functions depends on the accuracy of the underlying operating system. The operating system used in our case (Windows Vista) reported at 15-millisecond accuracy [20]. The execution times recorded in our benchmarks ranged from 109 to 623993 milliseconds. Therefore, the possible loss of accuracy owing to the 15 millisecond granularity, was deemed insignificant.

The process whose memory usage is to be measured completes its execution just before the output phase commences. It is at this moment that the memory consumption of the process is measured. The Qt method called `processIdentifier()` returns a pointer to a struct containing process information. Among others this struct provides a handle that can be used to call a C++ function defined in the Process Status API `<psapi.h>`, which in turn updates some memory counters related to the process. The memory counters use bytes as their units of measurement. From these memory counters, the peak working set size was taken to represent the memory usage of the process.

5. BENCHMARKING

5.1 Calibration

The test environment was calibrated before the performance of the processes that implement the algorithm was benchmarked. To do this each of our implemented programs were executed one hundred times with an empty data set. Its memory usage was measured each time. These data were analyzed to determine the memory overheads of these implementations.

The operating system allocated memory to the processed in 4096 byte sized blocks. The smallest amount of memory blocks used by any of these processes was 327 blocks. This was the memory allocated to the program that implemented `std::vector<char>`. It used 327 blocks for 61% of the runs with this program. For 38% of the runs of this program, it used 328 blocks, and 329 blocks for the remaining 1% of the runs. This is shown in the first row of Table 1.

The programs that implemented `std::vector<bool>`, `boost::dynamic_bitset` and `bm::vector<>` had a similar pattern of memory usage. For all these processes roughly 65% of the runs used the lowest number of memory blocks recorded for that specific process. However, the implementation of each of these structures had a different base value. They respectively needed 7, 9 and 12 additional memory blocks when compared to the number of memory blocks used by the program that implemented `std::vector<char>`. These values are shown in a column labeled Δ. For example, the program that implemented `std::vector<bool>` used $327 + 7 = 334$ memory blocks for 66% of its runs and similarly 335 blocks for 29% of its runs, and 336 blocks for 5% of its runs.

The different memory usage pattern and large Δ-value of the program using `Qt::QBitArray` can be explained by the fact that this program is dependent on a pre-compiled run time library that resides in a separate file as opposed to the other programs that are self contained. Based on these observations the base memory overhead for this experimental study was declared to be 327 memory blocks.

We assume that the additional memory blocks, represented by its Δ-value in each case is needed for algorithmic enhancements, included in the program, to enable the manipulation of the data stored in the data structure. We deem this additional memory to be part of the payload to be accounted for when using the data structure.

5.2 Experimental Data

We performed a number of preliminary benchmarks with data sets of different sizes and different densities and observed that the relative performance of the different data structures did not change significantly if data sets with different densities were used. Therefore, we decided to keep a fixed density for this benchmark.

Dahlberg [5] benchmarked transitive closure algorithms with *real world* data. The sizes of her data sets ranged from 88×88 to 987×987. These had densities ranging from 0.2% to 3.4%. The average density of these data sets was 1.1%. Our generated data was chosen to be in the same range as the data used by Dahlberg in terms of size and density.

Fifty synthetic data sets were generated, each representing a two-dimentional array of bits. The smallest data set was 100×100 and the largest 1000×1000. All the bits in an array were initialised to '0', and then a simple algorithm that uses the random function provided in the STL was used to set random bits to '1' until the array was 1.1% filled. Five arrays of each size were generated. These data sets were stored and used in our benchmark.

5.3 Performance Data

Each program was executed ten times with each of the data sets, measuring both the CPU-time and memory usage for every run. The average over all the runs for each data set size was taken for each program. The variation of memory usage between different runs on the same data set was similar to the variations that were observed during calibration. The time performance was observed to be particularly stable between different runs of a chosen program on the same data set and with minor variations between data sets. No outliers were observed with respect to memory usage or performance times.

5.4 Time Performance

The compilation of the code used in the benchmarks was done with level `-O3` optimisation enabled. Turning on optimization when compiling, makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program. The highest level of optimisation for the gcc compiler is turned on with the -O3 flag. It enables all optimisations specified by the lower optimisation levels as well as a number of additional ones. It will for example integrate functions into their callers when their body is smaller than expected function call code. It may also move branches with loop invariant conditions out of the loop, with duplicates of the loop on both branches. Another common optimisation is to reorder the instructions of innermost loops achieving the overlap of different iterations or to switch an inner and its outer loop when the switch can potentially reduce cache misses [22].

Figure 2 shows the performance times of the process for each of the selected data structures for arrays ranging from 100×100 to 1000×1000. Since the complexity of the algorithm used in these benchmarks is $O(n^3 log_2 n)$, one expects to see exponential growth in performance time. However, it is evident that the growth in performance time of some of the used data structures are consistently worse than others.

The execution times achieved using `std:vector<bool>` and `std::vector<char>` grows extremely fast for arrays larger than 200×200. Their performance times for larger arrays are outside the scope of Figure 2. The implementation using `std::vector<bool>` took more than 10 minutes to execute a 1000×1000 array, and the implementation using `std::vector<char>` took almost 2.5 minutes.

The observed time performance of the implementation using `bm:bvector<>` is much more acceptable than the execution times of the implementations respectively using `std::vector<bool>` and `std::vector<char>`. However, it is still not very usable for this application. Although it succeeded to complete its calculation of the transitive closure of a 1000×1000 array in about 15 seconds on average, its performance is still much worse than the time performance of the implementations using `Qt::QBitArray` and `boost::dynamic_bitset`.

5.5 Memory Usage

Figure 3 shows the memory usage of the algorithm implemented using each of the data-structures. The memory usage is shown for arrays 100×100 to 1000×1000. The memory usage is measured in Kibibytes(KiB). Memory usage of more than 1.8 MiB is not shown. It is clear that `bm:bvector<>` performed significantly worse than the rest. The memory requirements for `std::vector<bool>`, `std::vector<char>` and `boost::dynamic_bitset` are very similar for small arrays and insignificantly small when compared to the memory requirements of `bm:bvector<>` and `Qt::QBitArray`. The apparently constant memory usage of `Qt::QBitArray` can be explained by the roughly 990 KiB it needs for algorithmic support. It seems as if most of the actual data that is manipulated with this implementation fits within the memory that had to be allocated to accommodate its algorithmic support. It can be observed that at array size of 300×300 the `Qt::QBitArray` application is gradually requiring more memory for the data that can not be squeezed in the space allocated for its algorithmic support. It seems as if the array data has an influence on memory usage only for array sizes of 800×800 and larger. Since the data used in these benchmarks are of order $O(n^2)$ one expects to see quadratic growth in memory usage which can be observed for all data structures. For `std::vector<bool>`, the growth is quadratic too, although it seems to be almost linear on this scale.

6. DISCUSSION

The relative time performance of the programs that implemented the data structures is shown per data structure in Table 2. The position of each data structure when manipulating a 1000×1000 array is shown. 1 means the program executed the fastest, and 5 means it executed the slowest.

Table 2: Relative time performance of the programs that implemented the data structures

Data Structure	Rank
`std::vector<bool>`	5
`std::vector<char>`	4
`boost::dynamic_bitset`	1
`Qt::QBitArray`	2
`bm::vector<>`	3

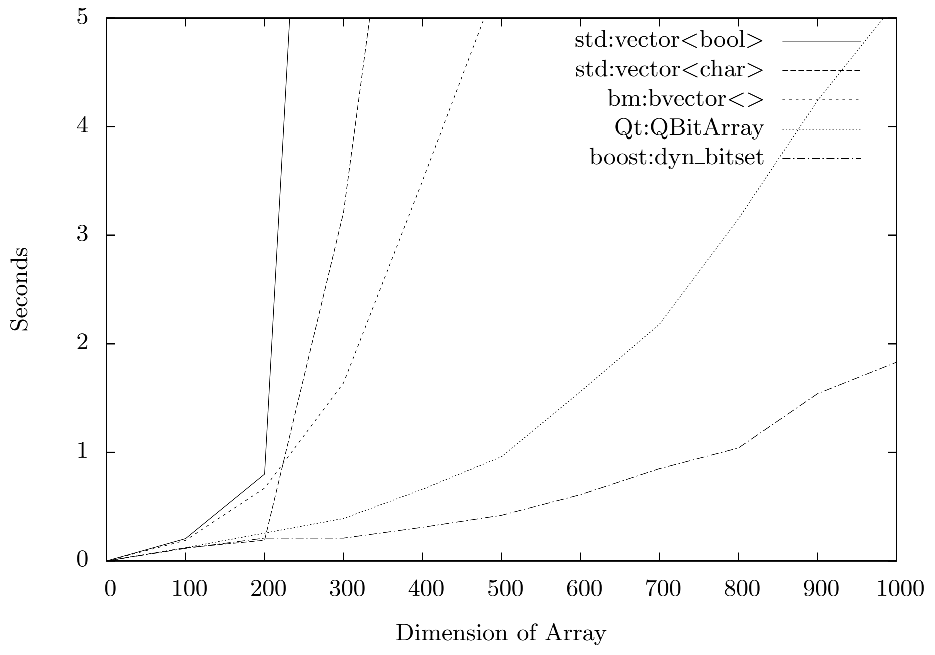

Figure 2: Time performance for arrays 100×100 **to** 1000×1000

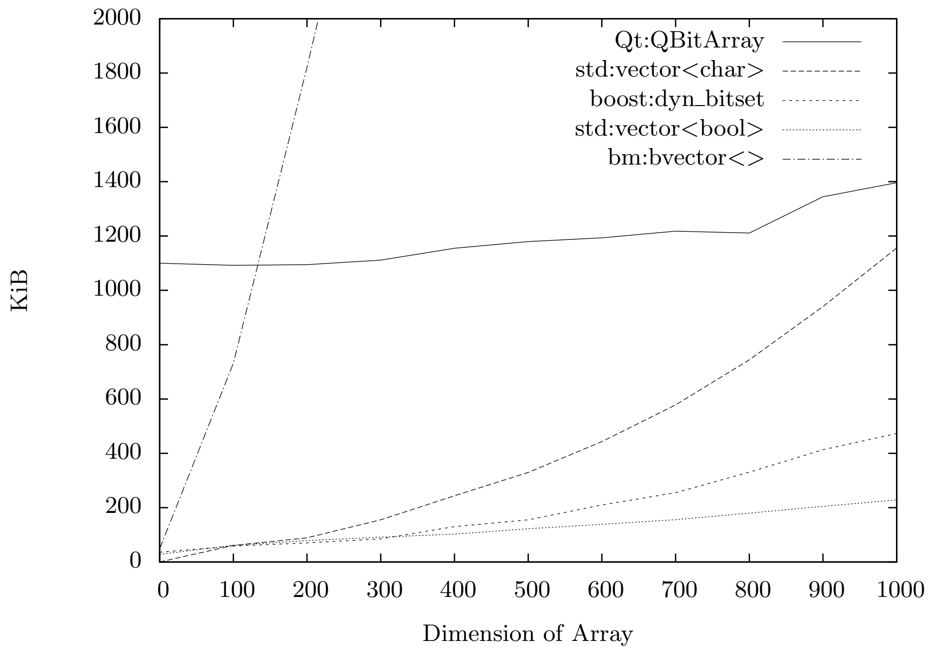

Figure 3: Memory usage for arrays 100×100 **to** 1000×1000

The relative memory usage of the programs that implemented the data structures is shown per data structure in Table 3. The position of each data structure when manipulating a 1000 × 1000 array is shown. 1 means the program used the least memory, and 5 means it used the most memory of these five.

Table 3: Relative memory usage of the programs that implemented the data structures

Data Structure	Rank
`std::vector<bool>`	1
`std::vector<char>`	3
`boost::dynamic_bitset`	2
`Qt::QBitArray`	4
`bm::vector<>`	5

The results of the `bm:bvector<>` were disappointing. Although it maintained acceptable performance times, its memory usage is exorbitant. In private e-mail conversation with a developer a BitMagic [15], this memory consumption is attributed to meta data regarding the characteristics of the vector that is warranted by impressive gains when manipulating much larger vectors with large portions that are very dense and other portions that are very sparse. It is obvious that this data structure was intended to be used in situations that are quite different from our application.

The `std::vector<bool>` ranks first with respect to memory performance. This is testimony of its successful space optimisation, achieved by storing the data in a format where all the memory that is allocated is purely used to store the data in its raw format without any meta information, thus eliminating all possible memory overheads. As a result, it ranks last with respect to time performance. This shows that the space-time tradeoff is huge. Because no meta information is stored, individual bits are accessed programatically by unpacking the byte that contains the bit that is accessed. Despite the impressive memory performance of `std::vector<bool>`, its dismal time performance renders it unusable in large-scale applications.

The `std:vector<char>` uses no data compression. Each bit is stored in its own computer word, i.e 16 or 32 bits depending on the architecture used. Respectively only 6.25% and 3.125% of the allocated memory carries significant data for 16 bit and 32 bit architectures. This results in very fast access times to individual bits. Because of this, it performed well when compared to `std::vector<bool>` in terms of execution time. However, owing to its wasteful memory usage its memory usage for larger arrays grows quite fast. Furthermore, when compared to the other data structures its time performance ranks very low despite its fast access to individual bits. This can be attributed to the fact that there are no bit-wise manipulations defined that can be used to manipulate whole vectors. This forces the programmer to iterate through the vectors and apply the required operations on the individual bits.

The `Qt::QBitArray` is a special byte array that can access individual bits and perform bit-operations on entire arrays of bits. Its implementation uses techniques similar to `std::vector<bool>` to compress and store data in its raw format with minimal overhead and to gain access to individual bits. However, it capitalises on the bitwise operations that are defined on primitive data types to implement the bitwise operations on whole arrays, resulting in achieving the best of both worlds. Consequently it performed very well in terms of time. Its relatively bad performance with regard to memory usage is contrary to what would be expected when inspecting the code. This anomaly can be explained in terms of how we defined base memory usage. As stated in Section 5.1 we include the memory allocated to the process, in addition to the memory size that is used by the process implementing `std::vector<char>`, to be part of the memory used for data manipulation. Therefore this seemingly excessive memory usage can be attributed to its larger process footprint owing to having to link to the entire Qt library. It is expected that its memory consumption will come closer to the other implementations, or even be more efficient, when the data size grows large enough for this constant larger footprint to be negligible.

The `boost::dynamic_bitset` uses template programming in its constructors allowing the programmer to specify the block size and an allocator. If not specified, the defaults render the data structure that to a large extent resembles `std::bitset`. However, it is much more versatile because a bit-vector object of this class can be resized at runtime because functions such as **`push_back()`**, **`append()`** and **`resize()`** are defined. It also uses similar techniques as **`std::bitset`** to implement all the bit-wise operations. Because it stores the data in raw format without much overhead its memory performance ranks high in comparison with the other data structures. Furthermore, because it capitalises on the operations that are defined in **`std::bitset`** to implement its bit-wise operations, its time performance ranks the highest of the data structures that are discussed in this study.

7. CONCLUSION AND FUTURE WORK

We have shown that `boost::dynamic_bitset` is considerably more efficient than most of the other implementations in terms of execution speed, while the implementation using `std::vector<char>` outperformed the other implementations in terms of memory efficiency. Our results clearly illustrates the time-space tradeoff that is at play when implementing software. We provide detailed information that developers may use as guide for choosing an appropriate C++ library for the implementation dynamic bit-vectors.

To provide broader comparison of bit-vector implementations, these experiments should be repeated with more bit-vector implementations such as those offered by [3, 10, 18] as well as others that may appear in the public domain in the future.

The use of other algorithms that perform a wider variety of bit-operations can potentially highlight more subtle differences between these bit-vector implementations. The use of algorithms from different application domains such as graphics, near-realtime network and cluster management, statistical computing and others may shed light on the relative usefulness of the different implementations as it may differ from one application domain to the other.

The role of the operating system, the hardware and the compiler should also be investigated. Different CPU types as well as variations in operating systems may have an impact. Furthermore, the performance of implementations using the different libraries may be influenced by the choice and version of compiler. Therefore these experiments should be repeated on different platforms and by using different compilers.

8. REFERENCES

[1] J. Allsop, A. Meredith, and G. Prota. Proposal to add a Dynamically Sizeable Bitset to the Standard Library Technical Report. Revision 1. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2050.pdf`, June 2006. [Online: Accessed 2009/07/31].

[2] J. J. Baker. A note on multiplying Boolean matrices. *Communications of the ACM*, 5(2):102, 1962.

[3] S. Beyer. Bit::Vector. `http://webscripts.softpedia.com/script/Development-Scripts-js/C-C-Library/Bit-Vector-26508.html`, 2009. [Online: Accessed 2009/08/15].

[4] J. Blanchette and M. Summerfield. *C++ GUI programming with Qt 4*. Pearson Educational, Upper Saddle River, NY, 2006.

[5] M. Dahlberg. Efficient algorithms for computing transitive closure in cwb: Implementation and comparison of several variations. Master's thesis, Swedish Institute of Computer Science, Sweden, September 1991.

[6] B. Dawes. Proposal for a C++ Library Repository Web Site. `http://www.boost.org/users/proposal.pdf`, May 1998. [Online: Accessed 2009/07/31].

[7] B. Dawes. Boost Formal Review Process. `http://www.boost.org/community/reviews.html`, 2000. [Online: Accessed 2009/07/31].

[8] B. Dawes. Library Issue 96: Fixing vector<bool>. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2160.html`, January 2007. [Online: Accessed 2009/07/31].

[9] E. W. Dijkstra. *A discipline of programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976.

[10] M. Dipperstein. ANSI C and C++ Bit Manipulation Libraries. `http://michael.dipperstein.com/bitlibs/`, June 2008. [Online; accessed 5-September-2009].

[11] H. Hinnant. C++ Standard Library Active Issues List (Revision R67). `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2009/n2948.html`, September 2009. [Online: Accessed 2010/04/09].

[12] H. Hinnant. C++ Standard Library Closed Issues List (Revision R68). `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2009/n3013.html`, November 2009. [Online: Accessed 2010/04/09].

[13] ISO/IEC. *ISO/IEC 14882:2003 - Programming languages – C++*. ISO, Geneva, Switzerland, 2003.

[14] D. Kalev. What You Should Know about vector<bool>. `http://www.informit.com/guides/content.aspx?g=cplusplus&seqNum=98`, March 2004. [Online: Accessed 2009/07/30].

[15] A. Kuznetsov. Re: Experiments with BitMagic. private e-mail communication <anatoliy_kuznetsov@yahoo.com>, September 2009.

[16] A. Kuznetsov, M. Shemanarev, I. Tolstoy, E. Lewis, and O. Khovayko. BitMagic. `http://bmagic.sourceforge.net/`, n.d. [Online: Accessed 2009/07/31].

[17] S. D. Meyers. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library* . Professional Computing Series. Addison-Wesley, Boston, 4 edition, 2004.

[18] H. Mostafa. Bits Array Encapsulation. `http://www.codeproject.com/KB/cpp/BitArray.aspx`, December 2004. [Online; accessed 5-September-2009].

[19] Nokia Corporation, Inc. Qt - A cross-platform application and UI framework. `http://www.qtsoftware.com/`, n.d. [Online: Accessed 2009/07/31].

[20] J. Reesig. Accuracy of JavaScript Time. `http://ejohn.org/blog/accuracy-of-javascript-time/`, November 2008. [Online: Accessed 2009/08/18].

[21] R. Rivera. Boost Version History. `http://www.boost.org/users/history/`, n.d. [Online: Accessed 2009/07/31].

[22] R. M. Stallman. Using the GNU Compiler Collection. http://gcc.gnu.org/onlinedocs/gcc.pdf, 2008. Published by GNU Press [Online; accessed 05-September-2009].

[23] C. Strauss, E. Boyd, and K. Marshall. MinGW - Minimalist GNU for Windows. `http://sourceforge.net/projects/mingw/`, 2008. [Online: Accessed 2009/07/31].

[24] H. Sutter. vector<bool>: More problems, better solutions. `http://www.gotw.ca/publications/N1211.pdf`, 1999. [Online: Accessed 2009/07/30].

[25] C. Walbourn. Game Timing and Multicore Processors. `http://msdn.microsoft.com/en-us/library/ee417693(VS.85).aspx`, 2005. [Online; accessed 12-Nov-2009].

[26] S. Warshall. A Theorem on Boolean Matrices. *Journal of the ACM*, 9(1):11–12, 1962.