`bitset<N>` in the Copenhagen STL

Jeppe Nejsum Madsen
Odensegade 15, st
DK-2100 Copenhagen
Denmark
`nejsum@diku.dk`

Abstract. This report describes the implementation of the template class `bitset<N>` in the Copenhagen STL.

1. Introduction

Taken from The Copenhagen STL website (`www.cphstl.dk`), the project's main purposes is:

– to study and analyze existing specifications for and implementations of STL to determine the best approaches to optimization,
– to design alternative/enhanced versions of individual STL components using standard algorithmic and performance engineering techniques, and
– to implement and document the new versions in C++.

This paper documents these activities for the STL template class `bitset` in the Copenhagen STL Project.

2. `bitset<N>` functionality

The `bitset<N>` class is a collection of bits. Unlike normal STL containers, the size remains fixed at N and it doesn't support the concept of iterators. Instead, `bitset`'s interface resembles that of unsigned integers. It supports bitwise operations such as `&=`, `|=` and `<<=`

In general bit 0 is the least significant bit and bit $N-1$ is the most significant bit.

The complete specification can be found in the C++ standard [International Organization for Standardization (ISO) 1998].

2.1 Unclear semantics in the standard

Note: Unfortunately, the author only had access to a draft version of the standard. Time did not permit verification against the final version.

In a number of places, the semantics of the operations specified for `bitset<N` are undefined, non-intuitive or plain wrong:

CPH STL Report 2001-11, April 2001.

1. The semantics of `operator[](size_t pos)` is not described. From the class specification, one can infer that it should return a `reference` presumably to the bit indicated by `pos`. But should range checking be performed on `pos`? Is it allowable to make a `const` overload that just returns the bit value?
2. The semantics of class `reference` is not described.
3. The section on constructors ([lib.bitset.cons.5]) gives the following description of the effects of constructing a `bitset<N>` from a string:

   > Determines the effective length `rlen` of the initializing string as the smaller of `n` and `str.size() - pos`. The function then throws `invalid_argument` if any of the `rlen` characters in `str` beginning at position `pos` is other than 0 or 1. Otherwise, the function constructs an object of class `bitset<N>`, initializing the first `M` bit positions to values determined from the corresponding characters in the string `str`. `M` is the smaller of `N` and `rlen`.

   This means, that constructing a `bitset<N>` from a string can throw an exception if the string contains invalid characters, even if those characters are never used for initialization.
4. The description of `operator~()` says: *Constructs an object x of class* `bitset<N>` *and initializes it with* `*this`. I.e. it never flips the bits.

3. Implementation

The standard doesn't explicitly impose any complexity restrictions on `bitset` besides the general container requirements which are listed in Table 1.

Method	Complexity
Copy constructor	linear
Comparison	linear
`size()`	constant

Table 1. General container complexity requirements applicable to `bitset`

It's clear that with only these requirements, `bitset<N>` can be implemented using e.g. `vector<bool>`. But since the number of elements in the container are fixed at compile time, better implementations are possible.

Since `bitset` resembles unsigned integers with (in theory) an unlimited number of bits, it seems reasonable to assume that bitwise operators has linear (in the case of operations on the entire container) or constant (in the case of single bit operations) complexity. This rules out more exotic implementations such as signature encoding [Mehlhorn et al. 1997] which, in return for equality comparison in $O(1)$, requires polylogarithmic time for any updates to the sequence.

3.1 Representation

In this implementation the collection is represented as an array of words, each having the type T. The number of words required are

$$\#words = \left\lceil \frac{N}{CHAR\_BIT \times sizeof(T)} \right\rceil$$

This representation uses $\#words \times sizeof(T) \times CHAR\_BIT - N$ extra bits. An invariant that is always maintained is that these bits are zero.

3.2 Algorithms

The implementation of the various methods in `bitset<N>` are mostly straightforward once the representation is decided. There's little room for algorithmic improvement; it's more an exercise in performance- and software engineering.

In the present implementation, most optimizations are left to the compiler using inline functions and loop unrolling. The only "trick" used is in `count()`, which uses a table to lookup the number of 1-bits in a single byte.

4. Benchmark results

The benchmark results from both the present implementation and SGI's implementation is compared on two different architectures and with several different operations. Nanna is an Intel Pentium III 550MHz running Redhat Linux 6.2. Bjarke is a Digital Alpha running Digital Unix 4.

As can be seen from the figures, performance is almost identical for both implementations except in two cases:

1. SGI's implementation is faster for small N. This is due to the fact that they specialize the implementation if the number of bits can be contained within a single word. The usefulness of this specialization is unclear: if the number of bits can be contained in a single word, why use bitset?
2. The shift operations seems to be slightly faster in the present implementation when running on the Intel x86 architecture.

Figure 4.1. Applying `operator&=` to `bitset` containing n bits, nanna

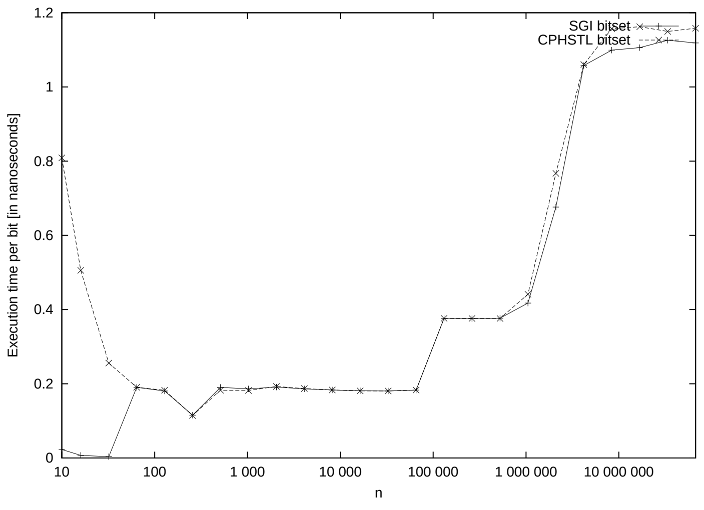


Figure 4.2. Applying `operator==` to `bitset` containing n bits, nanna

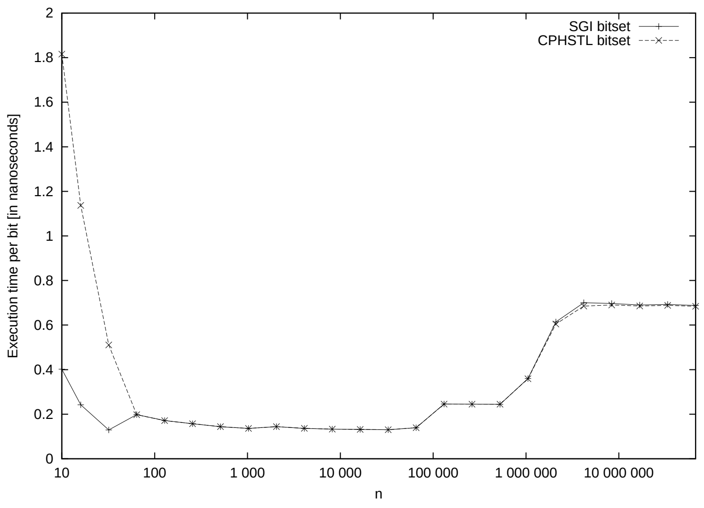


Figure 4.3. Applying `operator<<=(30)` to `bitset` containing n bits, nanna

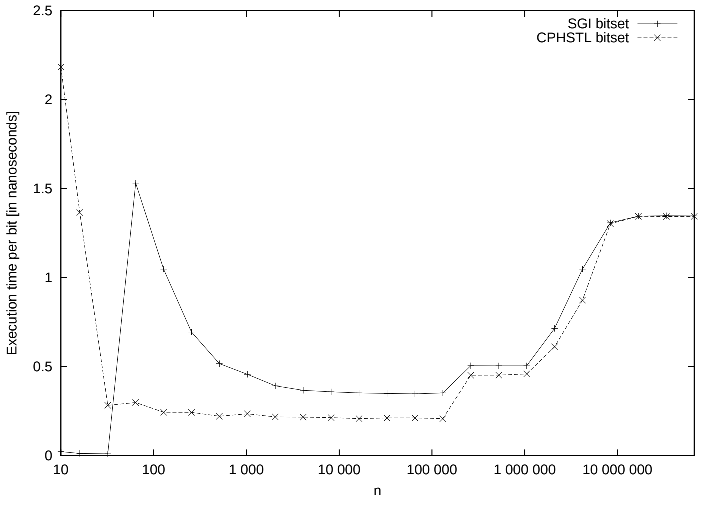


Figure 4.4. Applying `operator<<=(300)` to `bitset` containing n bits, nanna

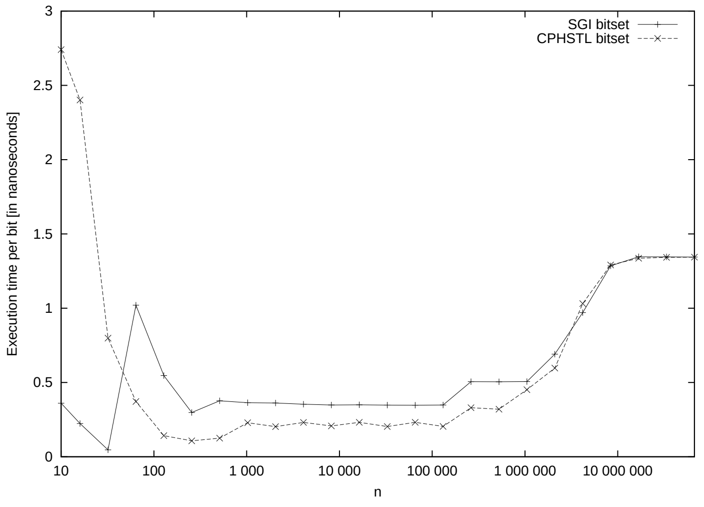


Figure 4.5. Applying `operator&=` to `bitset` containing n bits, bjarke

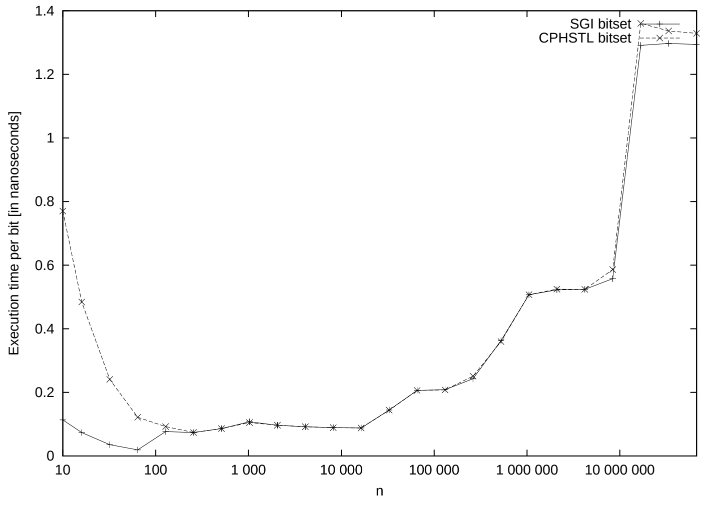


Figure 4.6. Applying `operator==` to `bitset` containing n bits, bjarke

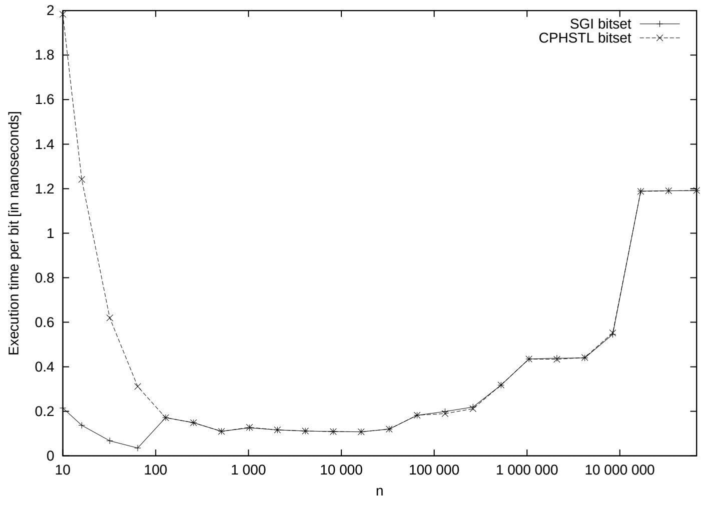


Figure 4.7. Applying `operator<<=(30)` to `bitset` containing n bits, bjarke

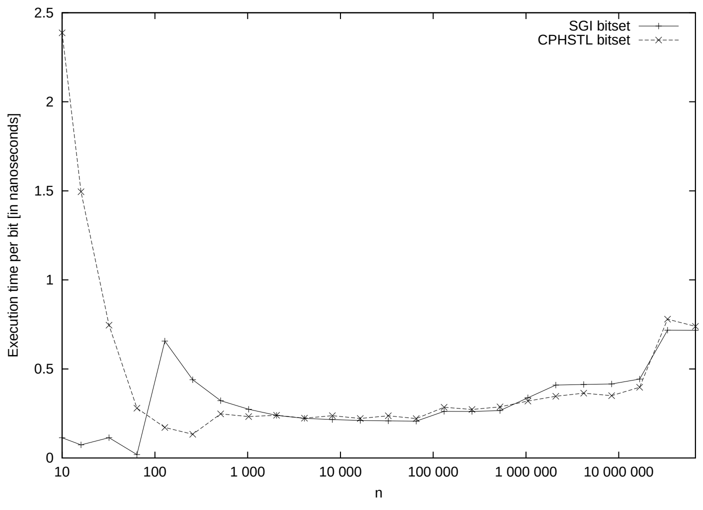
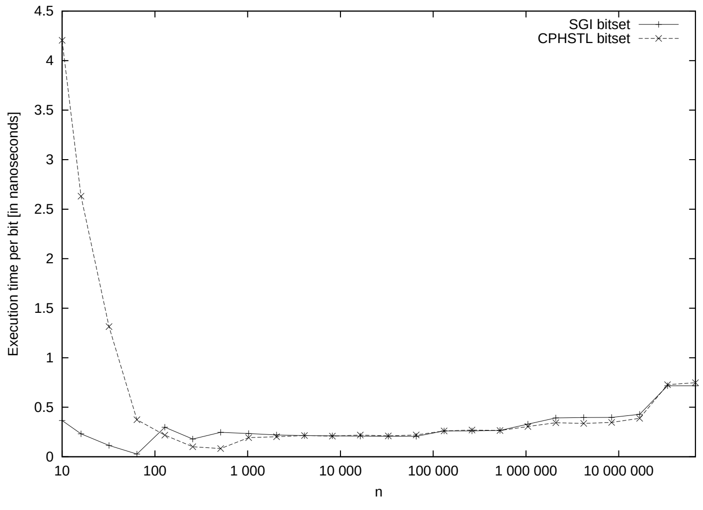


Figure 4.8. Applying `operator<<=(300)` to `bitset` containing n bits, bjarke



5. Software engineering aspects

Since the development of `bitset` is more an exercise in software engineering than anything else, it seems natural to comment on some of the issues that we encountered during the development period.

Standards conformance The standards conformance of the Gnu C++ compiler 2.95.2 (or rather, the supplied library) made it impossible to create `bitset` as described in the standard. In the Gnu library, iostreams are not templates, which is needed for the i/o operators.
Also, the `to_string()` method should be a template member function, which returns a `basic_string`. Gnu C++ refused to recognize the function, unless it was made a non-template member function returning a `string`.

Template instantiations It's clear that code that relies heavily on templates is still problematic for the compiler. The test suite for `bitset` instantiates 33 different test classes which in turn instantiates many `bitset` classes. Compiling this without any optimizations is not a problem, it completes within a minute. But compiling with `-O` takes more than 1 hour and requires more than 800MB of memory on a Sun Sparc running 400MHz.

6. Conclusion

The present implementation delivers a standard compliant implementation which matches, and sometimes improves, the runtime of the SGI implementation for the cases where a bitset cannot be contained within a single word.

References

International Organization for Standardization (ISO). 1998. *ISO/IEC 14882: Standard for the C++ Programming Language*. Genevé.

Mehlhorn, K., Sundar, R., and Uhrig, C. 1997. Maintaining Dynamic Sequences under Equality Tests in Polylogarithmic Time. *Algorithmica 17*, 183–198.