7 Years Later: On `vector<bool>` and optimized Standard Algorithms in libstdc++ for Google Summer of Code 2019

Date: 2019-08-18
Project: libstdc++, Programming Language C++
Audience: Free Software Foundation, libstdc++-dev
Reply-to: JeanHeyd Meneide<phdofthehouse@gmail.com>
Columbia University

In 2012, Howard Hinnant wrote a blog post about the performance characteristics and qualities of not only `vector<bool>` but the standard algorithms that underpin all containers [1]. By this time, libc++ added internal overloads to optimize `std::vector<bool>`'s iterator for various algorithms [2] including `std::find`, `std::fill(_n)`, `std::find`, `std::equal` and others. Things are not as far along in libstdc++. Only `std::fill` [3] has an overload in libstdc++'s library for `_Bit_iterator`s. Performance data captured in statistically rigorous benchmarks for various Standard Libraries – including VC++ and libstdc++ – show that `vector<bool>` and its bit iterators are not as optimized as they ought to be [4].

With the coming revival of bit utilities paper for the C++ Standard and the potential of a new suite of bit utilities coming from a header `<bit>` [5], the goal of this Google Summer of Code 2019 project will be to identify existing algorithms where libstdc++ will benefit from additional overloads based on using the bit iterators. This proposal also explores the fundamental appeal of broadening this class of optimizations to types that are not only represented by bit iterators or `std::vector<bool>`, but any type whose bits are trivially relocatable under the upcoming work of Arthur O'Dwyer that is gaining both LLVM Clang [6] and the GNU Compiler Collection (GCC) [7] implementation traction, as well as broader C++ community exposure [8].

1 Background

I am a student at Columbia University who spends their free time working on high-quality C++ libraries [9] and participating in C++ Standardization [10]. After engaging in a Library Working Group "Issues Processing" session during the 2018 Rapperswil-Jona C++ Standards Meeting [11] at the HSR Hochschule für Technik Rapperswil, I realized that I could help contribute to the Standard Libraries that make C++ both fast and easy to use.

I also wanted to try implementing the newer bits of the C++ Proposal for Bit Utilities, P0237 [5]. But before I do that, I realized that getting real-world experience with the existing bit utilities in an actual C++ Standard Library would give me a much greater perspective in helping to move those utilities forward and push a much more complete product into the C++ Standard.

My résumé is available upon request.

2 Phase 1: Updating tr2::dynamic_bitset using accumulated knowledge based on an updated P0237

Initial communication with P0237's author – Dr. Vincent Reverdy – has yielded that P0237 is likely to receive an update before GSoC 2019 comes out with a new approach to greatly expand the utilities of a `<bit>` header.

libstdc++ has a minimally-maintained dynamic bitset in <tr2/dynamic_bitset>, but needs to be more thoroughly tested and updated.

The new direction for P0237 – as has been explained – will be to allow for a container adapter which might sit on top of any container, allowing the user to pick the necessary underlying container which will provide for the (primitive) type that bit operations will be performed on. An early implementation of `std::dynamic_bitset<Container` will prove instrumental in vetting design decisions and optimization opportunities for C++, and prevent having to write Yet Another Owning Container With Slightly Different Conflated Semantics for Standard Library developers.

The full deliverable for Phase 1 will be an implementation and discussion of `template <typename Container> class std::dynamic_bitset;` for libstdc++.

1. Week 1 will be spent identifying the necessary requirements to place on Container (e.g., should it work with just `ContiguousSequenceContainer`, or with things like `std::list` that are just bidirectional sequences?). It will also be spent considering whether `std::dynamic_bitset<std::span<T>>` should be possible, and whether other non-owning ranges can be put into `std::dynamic_bitset` or if `template <typename Range> std::bitset_view<Range>` might be a valuable addition to the library.
2. Weeks 2-3 will be spent implementing the container, taking advantage of the requirements identified in Week 4.
3. Week 4 will be locating and potentially acting on optimization opportunities inside of `dynamic_bitset`.

The reason this portion is worked on first is because we will need to finalize and update `_Bit_iterator` and `_Bit_reference` to be shared between this implementation and `vector<bool>` for Phase 2.

3 Phase 2: Library Algorithms

At this stage, verification of the work done begins and iteration over the previous functions while we look to the standard library algorithms for optimization opportunities will begin. Because we will have a common bit iterator and bit reference nomenclature, we can specialize and overload C++ Standard Library algorithms to take advantage of both contiguous iterator and bit iterator structures. Verification will include:

1. Correctness - Algorithms will be tested against libstdc++'s preexisting tests for these algorithms to ensure that adding these overloads does not break the correctness of the algorithm's implementations;

2. Benchmarks - performance is correctness. Currently, there is a harness of benchmarks created specifically for this proposal. Hooking into libstdc++'s testing infrastructure will be vital in ensuring the longevity of these changes (and if not, a test suite with benchmark data is available at [4]); and,
3. Code review - the general quality review process that happens through mailing list discussion.

Notably, tests that we must pass are contained in the following files, as of the time of writing:

- testsuite/25_algorithms/stable_sort/3.cc
- testsuite/25_algorithms/adjacent_find/vectorbool.cc
- testsuite/25_algorithms/find/vectorbool.cc
- testsuite/25_algorithms/is_permutation/vectorbool.cc
- testsuite/25_algorithms/heap/vectorbool.cc
- testsuite/25_algorithms/find_end/vectorbool.cc
- testsuite/25_algorithms/find_if_not/vectorbool.cc
- testsuite/25_algorithms/iter_swap/20577.cc
- testsuite/25_algorithms/find_first_of/vectorbool.cc
- testsuite/25_algorithms/find_if/vectorbool.cc
- testsuite/25_algorithms/sort/vectorbool.cc

As stated in early communication [12], many of the bitset tests only deal with a small number of bits. Expanding this test suite will be part of the deliverable value of this proposal.

3.1 Phase 2 MVP: improved libstdc++ `std::find`, `std::equal`, `std::copy` and `std::count`

Improving these algorithms will follow in the same vein of the improvements hinted at in Howard Hinnant's 2012 article: we will use specialized built-in functions from GCC and Clang in order to provide several orders of magnitude speedup to many of these algorithms to help the standard library remain competitive with hand-rolled implementation.

This will take up weeks 5 through 7 of Phase 2, spending about one week per two algorithm with additional tuning for the last week.

3.2 Phase 2 Stretch Goals: enhanced libstdc++ `std::swap_ranges`, `std::rotate`, `std::is_sorted`, `iter_move`/`iter_swap` and `std::is_sorted_until`

These algorithms are a bit more complicated than the above algorithms and require a degree of additional finesse to ensure correctness. While I am familiar with the intrinsic functions required to make the Phase 1 Deliverables possible, the built-ins used here will undoubtedly be similar but require some research. There are also search patterns and other optimizations that can be applied even at this level, and delivering on these algorithms will be more of a stretch goal.

Note that optimizations will not be applied to `iter_move` and `iter_swap`, just changes to make sure they behave properly for all the standard algorithms (albeit it seems like libstdc++ already specializes one of them: ranges work will allow specializing the second for bit iterators).

This will take up any additional time leftover from implementing the Phase 2's MVP.

4 Phase 3: Research and Guideline development for `is_trivially_relocatable` usage in algorithms

`is_trivially_relocatable<T>` is a trait that is seen as the current logical evolution of wanting to support what has been typically referred to as "destructive move": types whose bits can be taken from one place to another and who (effectively) end the lifetime of their source. `std::is_trivial_v<T>` gave us the legal definition of a type that should be trivially relocatable (and thus bit-blast-able) from one memory location to another. The current rendition of `is_trivially_relocatable<T>` being worked on by Arthur O'Dwyer and Marc Glisse in libstdc++ gives us a trait that identifies whether the physical layout of a type is also bit-blast-able from one place to another. In libstdc++, the patch remains for a strictly internal library trait. Early work suggests it is a good area for future-looking library improvements.

Phase 3 will be all about analyzing where and how in the Standard Library – in particular, algorithms such as `std::equal`, `std::find`, `std::fill`, `std::copy`, etc. can use this information to perhaps work down to either using intrinsic functions for copy operations or directly applying `std::memcmp`, `std::memchr`, `std::memcpy`, and other high-performance bit-based functions and intrinsic functions.

Should this proposal be accepted, Phase 3 will consist of:

1. Research - contacting Arthur O'Dwyer and engaging other Standard Library maintainers to help identify places where `is_trivially_relocatable<T>` can bring benefit to libstdc++;
2. Implementation - attempting proof of concept code rewritten to take advantage of `is_trivially_relocatable<T>` for various primitive types in libstdc++'s internal mechanisms; and
3. Review - provide feedback (possibly integrated into P1144 [13] and a new revision of P0237 [5]) that lays out the merits and usefulness of `is_trivially_relocatable<T>` for Standard Library implementations.

This is the most research-heavy portion of the proposal. A significant amount of proof of concept work has already been performed by Marc Glisse and Arthur O'Dwyer in this area for some standards containers: extending it and recognizing when it could apply to algorithms with ranges and iterators seems to be the next logical step for ensuring maximal performance in the Standard Library (if the compiler doesn't optimize such already).

Phase 3's MVP will be showing proof of communication and feedback with Arthur O'Dwyer, Vincent Reverdy and the chosen mentors about the benefits this can bring to the Standard Library. The STRETCH GOAL will be having the feedback incorporated into respective Standards papers and information left in libstdc++'s documentation about the research efforts, particularly places of useful or interesting design expansion not identified in the early weeks of Phase 1.

Any leftover work from Phase 2 will be checked here for correctness and performance, and tests will be enhanced.

5 Summary

Summarizing, this proposal seeks to:

1. Improve libstdc++'s handling of bit utilities to be on part with libc++'s while also teaching myself what a real-world usable Standard Library would require for bit utilities;
2. Produce an implementation of an adaptable P0237;
3. Research more closely additional places that the potential trait `std::is_trivially_relocatable` might bring to the standard library; and,
4. Use that information to help guide libstdc++'s future optimization development and to provide proper feedback to C++ Standardization with real-world implementation and usage experience.

I humbly request the opportunity to work on this for the greater C++ and open source communities as part of the Google Summer of Code 2019.

References

[1] Howard Hinnant. On vector<bool>. `https://howardhinnant.github.io/onvectorbool.html`, 2012.

[2] Chandler Carruth LLVM, Marshall Clow and Eric W. Fiselier. libc++ __bit_iterator source code. `https://github.com/llvm-mirror/libcxx/blob/7c3769df62c0b3820130aa868397a80a042e0232/include/__bit_reference#L390`, 2018.

[3] Free Software Foundation. libstdc++ _bit_iterator source code. `https://github.com/gcc-mirror/gcc/blob/41d6b10e96a1de98e90a7c0378437c3255814b16/libstdc%2B%2B-v3/include/bits/stl_bvector.h#L411`, 2018.

[4] ThePhD. Bit - benchmarks and data for various standard libraries. `https://github.com/ThePhD/gsoc-2019-bit`, 2019.

[5] Vincent Reverdy and Robert J. Brunner. Fundamental bit manipulation utilities. `https://wg21.link/p0237`, 2017.

[6] Arthur O'Dwyer. Compiler support for p1144r0 "___is_trivially_relocatable(t)". `https://reviews.llvmN.org/D50119`, 2018.

[7] Arthur O'Dwyer Marc Glisse. Group move and destruction of the source, where possible, for speed. `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=87106`, 2019.

[8] Arthur O'Dwyer Howard Hinnant, Jon Kalb and Phil Nash. Episode 40 - moving and relocation guide. `https://www.youtube.com/watch?v=8u5Qi4FgTP8`, 2018.

[9] ThePhD. sol2. `https://github.com/ThePhD/sol2`, 2019.

[10] ThePhD. C++ standardization efforts. https://thephd.github.io/portfolio/standard, 2019.

[11] HSR Hochschule für Technik Peter Sommerlad. N4673 - spring 2018 jtc1/sc22/wg21 c++ standardization committee meeting. http://open-std.org/JTC1/SC22/WG21/docs/papers/2017/n4673.pdf, 2017.

[12] Jonathan Wakely. Re: [gsoc] bit iterators, trivial relocatability, and performance. https://gcc.gnu.org/ml/libstdc++/2019-02/msg00010.html, 2019.

[13] Arthur O'Dwyer. Object relocation in terms of move plus destroy. https://wg21.link/p1144, 2019.