Benchmarks of Cache-Friendly Data Structures in C++

❑ tyler.a.young ❑ C++ ❑ January 29, 2019January 30, 2019 ❑ 5 Minutes

Suppose you're a savvy C++ developer who knows all about data-oriented design (https://www.youtube.com/watch?v=rX0ItVEVjHc) and the importance of optimizing for cache locality (https://tylerayoung.com/2017/01/23/notes-on-game-programming-patterns-by-robert-nystrom/#chapter17-datalocalityi.e.data-orienteddesign) if you want a prayer of running fast on modern hardware. Now suppose you want to go beyond basics—"just use `std::vector` " is a good starting point, but you want more!

I'm going to go over six containers here that are alternatives to their standard library analogs: two replacements each for `std::vector` , `std::set` , and `std::map` . I'll talk about the ideas behind them, then dive into a comparison of their performance in benchmarks.

This post was inspired by Chandler Carruth's CppCon 2016 talk, "High Performance Code 201: Hybrid Data Structures (https://www.youtube.com/watch?v=vElZc6zSIXM)." If you're interested in this sort of thing, absolutely watch his talk!

Three clever containers from LLVM

The most interesting containers to talk about come from the LLVM project. They are:

`llvm::SmallVector` (http://llvm.org/docs/ProgrammersManual.html#llvm-adt-smallvector-h)

This is a small-size optimized `std::vector` replacement. "Small-size optimization" (SSO) means that instead of keeping all its data on the heap (like `std::vector` does), this class preallocates some amount of data (the actual size of which is configurable via a template parameter) locally on the *stack*, and *only* performs a (slow!) heap allocation if the container grows beyond that size. Because `malloc` is slow, and traversing a pointer to get to your data is slow, the stack storage is a double-win. The hardware folks didn't spend the last 60 years making CPUs faster so that we kids could chase pointers all over the place!

All of this works best if your types are small. If the objects you're storing are are large (i.e., a significant portion of a cache line!), you may be better off storing pointers to them instead.

There is some fine print here: SmallVector can be awkward at interface boundaries, since the preallocated size is part of the template type. Along interface boundaries, you generally don't care how much preallocated storage the thing has!

LLVM provides an elegant solution, though. `SmallVector` inherits from a type-erased base class called `llvm::SmallVectorImpl` . You can't instantiate objects of `SmallVectorImpl` directly, but you *can* accept references to one and manipulate it just like you would a `std::vector` —it has `push_back()` , `reserve()` , etc.

So, you simply use `SmallVectorImpl` references at interfaces boundaries, like this:

```
//   BAD: Clients cannot pass e.g. SmallVector<Foo, 4>.
void BAD_hard_coded_small_size(SmallVector<Foo, 8> &out);
//   GOOD: Clients can pass any SmallVector<Foo, N>.
void GOOD_allows_any_small_size(SmallVectorImpl &out);
```

`llvm::SmallSet` (http://llvm.org/docs/ProgrammersManual.html#llvm-adt-smallset-h)

`SmallSet` is a small-size optimized `std::set` replacement, analogous to `llvm::SmallVector` above. It uses a simple linear search when you're below the preallocated size, and only moves to fancy, higher-overhead, guaranteed-efficient hash-based lookups at larger sizes. The only fine print on this one is that there's no iteration support.

`llvm::DenseMap` (http://llvm.org/docs/ProgrammersManual.html#llvm-adt-densemap-h)

This is LLVM's general-purpose `std::unordered_map` replacement. Unlike the standard map classes, it keeps all its data in one memory allocation (good for locality!), and it does away with buckets in favor of keeping keys and values next to each other in memory. Moreover, it allocates a large number of key/value pairs by default (64, in fact), so it's super fast at small sizes. (The downside of that, of course, is that it's memory inefficient if you're creating a lot of very small maps, or if your types themselves are large.)

The fine print on this one is: its iterators are invalidated after insertion (unlike `std::map`). This strikes me as mostly a theoretical problem, since you could of course just store the *keys* rather than iterators. (And unlike with vectors, I don't know that I've ever come across code that retained map iterators…)

Three pseudo-read-only alternatives ("No, seriously, *just* use a vector")

There are three other containers I'd like to suggest as alternatives to the typical standard library choices. These are array-backed implementations designed to be write-once or "write-infrequently"—that is, they deliberately eschew normal mutation operations (`push_back()` , `insert()` , etc.) and only support wholesale *replacement*. In exchange, you get:

- A dead-simple implementation.
- A single memory allocation—during `replace()` , it `malloc` s exactly as much memory as you need.
- Super fast iteration (you could even specialize them to have fixed-size, stack-based variants).
- Lookups on map and set types that are O(log *n*)—after all, if you're not modifying the data after initialization, you might as well sort it and do binary search lookups!
- An API that makes it clear that, hey, you *really* shouldn't be manipulating this data. (Very useful for containers that are *conceptually* constant, but which you can't actually `const` -construct for whatever reason—this happens more often than I would have expected 5 years ago.)

These are `FixedArray` , `ArrayMap` , and `ArraySet` in the benchmarks below. You can find their implementation in the ArrayTypes.h header in the accompanying Git repo (https://github.com/s3cur3/llvm-data-structure-benchmarks/blob/master/ArrayTypes.h).

So how fast are these containers, really?

[This GitHub repo (https://github.com/s3cur3/llvm-data-structure-benchmarks)] provides a benchmark that pits the LLVM containers against both the STL types and my "mostly-read-only" containers. I've used this to generate results both [in text format (https://github.com/s3cur3/llvm-data-structure-benchmarks/blob/master/scripts/llvm_data_structure_benchmark_results.txt)] (the raw output from running Google Benchmark) and as graphs below. A complete set of the (72) graphs generated by the script can be found [here (https://imgur.com/a/QljLj7R)].

The TL;DR from this is:

- `SmallVector` is a big win for a win for emplace/push_back at sizes up to the preallocated "small size," and not a loss beyond that
- `SmallVector` is a big win for random reads at sizes up to the preallocated "small size" *until* you get so many elements preallocated that you start passing beyond cache lines
- `DenseMap` inserts are "OMG fast"—way faster than `std::map` or `std::unordered_map`
- `ArrayMap` is *not* actually a win over `DenseMap` for reads, and it's only a win over `std::unordered_map` at the smallest sizes (like, *8* elements)
- `SmallSet` insertion: a win at the preallocated size, a wash at larger sizes compared to `std::set`
- `SmallSet::count()` : marginally faster than `std::set` at best; kind of a wash at larger ones— `ArraySet` on the other hand winds up way slower at large sizes.

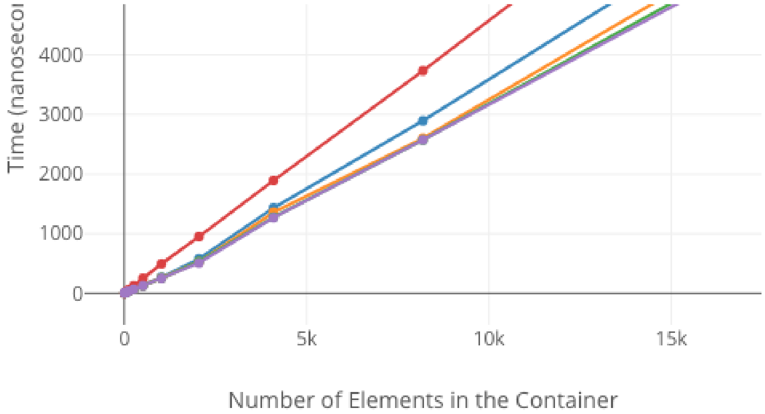
Note that all these numbers were obtained with data types between 4 bytes (i.e., an int) up to 64 bytes (i.e., 8 doubles). YMMV if you stick large objects in the structs. For the maps, I was using a 64-bit pointer as keys in all cases.

Following [the instructions in the repo's README (https://github.com/s3cur3/llvm-data-structure-benchmarks)], you can run the benchmarks on your own machine.

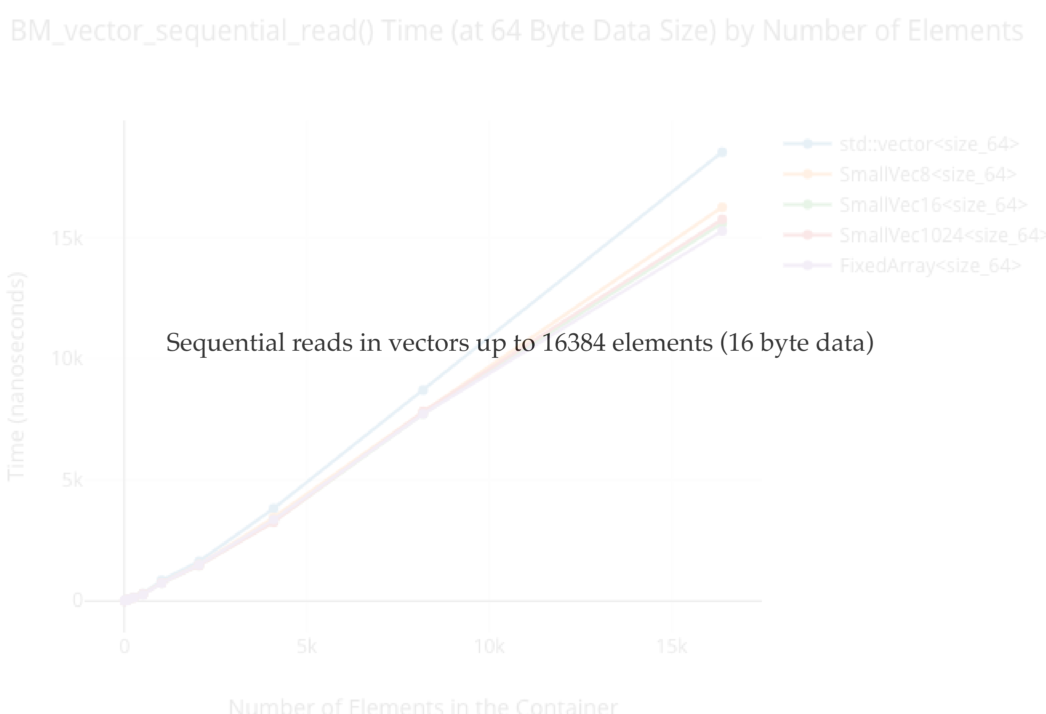
Benchmarks of Vector Alternatives

At large container sizes, it's a wash. No surprise there. What *was* surprising was that the 1024-element small-sized optimized vector does *not* seem to get a huge speedup at smaller sizes—it's only marginally faster than the size 8 and size 16 SSO vectors in the 256 element case.

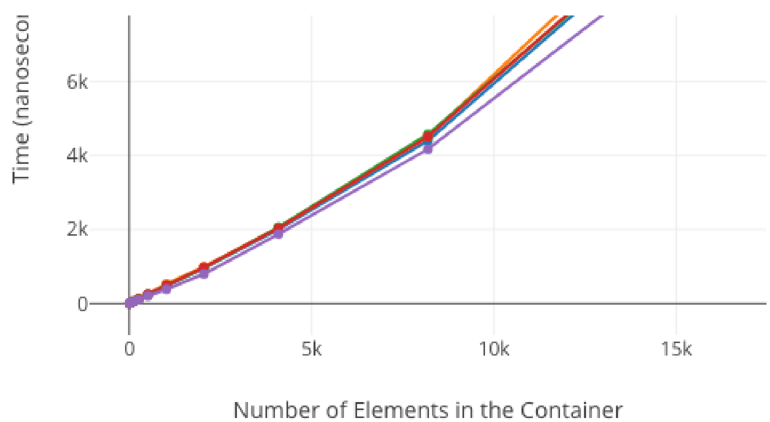
I do *not* have a good explanation for why the `FixedArray` 's read speed is way out front in the small size cases… except that maybe it's getting vectorization benefits due to just being "really damn simple"?



Sequential reads in vectors up to 16384 elements (16 byte data)



Sequential reads in vectors up to 16384 elements (64 byte data)

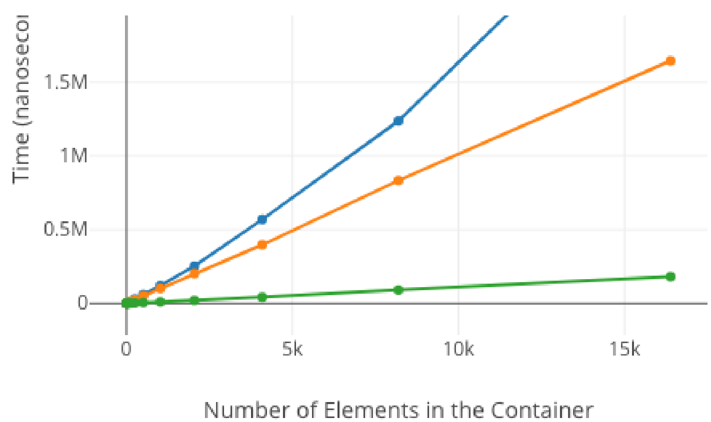


Random reads in large vectors (16 byte data)

Map Alternatives

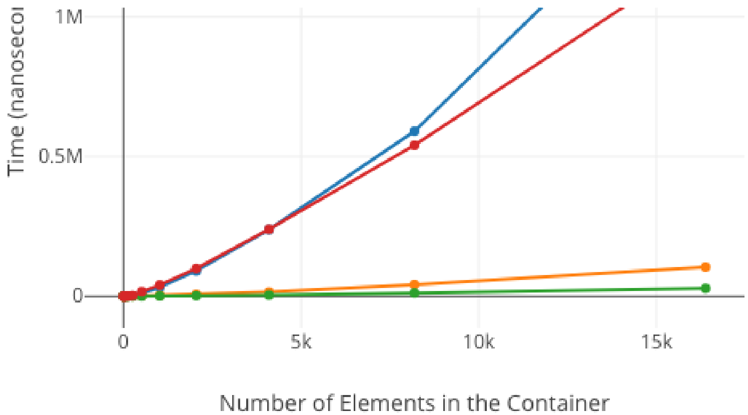
Note that inserting into the sorted-array-backed “map” (`ArrayMap`) isn’t included here, because if you’re inserting into it enough to be concerned, you’re using it wrong! 😉

Lookup in the `ArrayMap` is *shockingly* slow at large data sizes—again, I don’t have a good explanation for why this is.

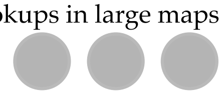


Insertion into large maps (16 byte data)

Insertion into large maps (64 byte data)



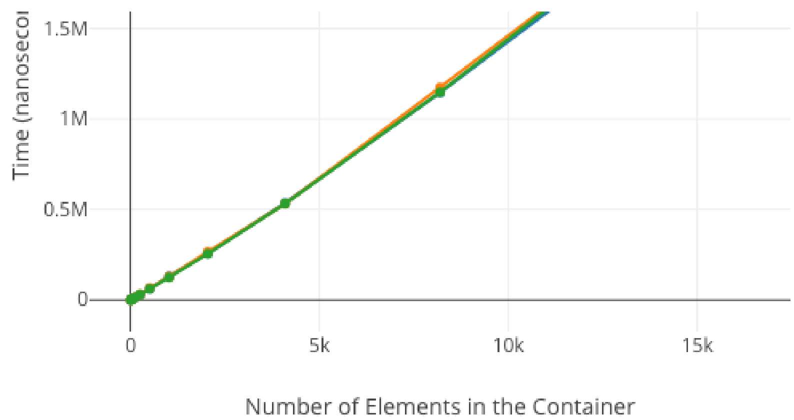
Random lookups in large maps (4 byte data)



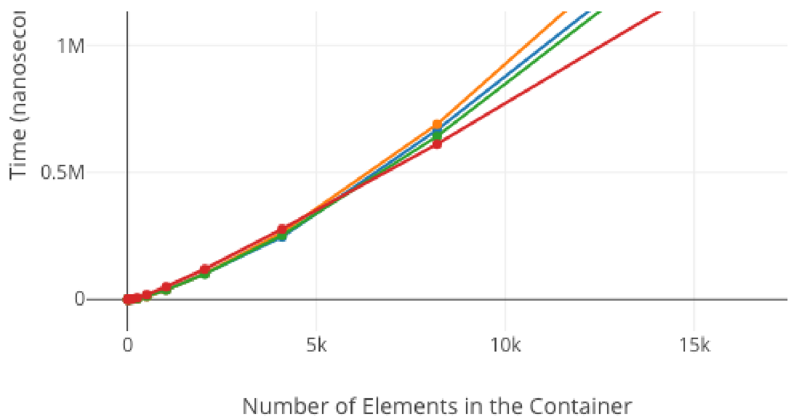
Set Alternatives

Again, insertion into the sorted-array-backed "set" (`ArraySet`) isn't included in the insertion graphs, because it's the wrong data structure to use unless insertions are massively infrequent.

As in the map case, the `ArraySet` 's lookups get "real slow" in some cases… for reasons I'm not clear on.



Insertion into large sets (16 byte data)



Random lookups in large sets (16 byte data)

Advertisements

Published by tyler.a.young

I'm a software developer working on the X-Plane flight simulator. @TylerAYoung on Twitter. View all posts by tyler.a.young

5 thoughts on "Benchmarks of Cache-Friendly Data Structures in C++"

Pingback: Benchmarks of Cache-Friendly Data Structures in C++ | My Tech Blog
Pingback: Benchmarks of Cache-Friendly Data Structures in C++ – Hacker News Robot
satish says:
January 30, 2019 at 12:25 pm
Interesting read ! Btw, in this sentence (first paragraph under "Three clever containers from LLVM
"):
… Because malloc is slow, and traversing a pointer to get to your data is slow, the SSO heap storage is a double-win. …

should this be "… stack storage is a double-win." ?

□ Reply

tyler.a.young says:
January 30, 2019 at 12:40 pm
Doh! Thanks for the heads up. Fixed now! 🙂

□ Reply

http://Behance.net says:
February 2, 2019 at 1:35 am
Fastidious answer back in return of this question with solid arguments and telling everything concerning that.

□ Reply