Table of Contents

Data Container Profiling

Introduction

This page provides some interesting results from profiling different container implementations. The profiling was done with some simple container implementations and a small standalone program outside the Gaudi framework. The main reason for not doing it inside Gaudi (yet) was to obtain some results quickly. However, this also allows to obtain some basic results without having to worry about possible side effects from the framework. The next step will be to repeat the exercise inside Gaudi, including data containers provided by the framework.

The Container Implementations

Six different data containers were implemented:

- `ContArrayStatKeepAssign`
- `ContArrayStatProtectedKeepAssign`
- `ContStdVector`
- `ContStdVectorKeepAssign`
- `ContStdVectorNew`
- `ContStdVectorPlacementNew`

Each container implements three methods: `reset()`, `append(const T&)` and `get(unsigned int)`. The `reset()` method prepares the container for the next event, `append()` adds a data instance to the container and `get()` allows access by index. With the exception of the first two, containers are constructed with an initial capacity which is increased if necessary during the test run. The two array based implementations have a fixed capacity. All implementations are templates, allowing to exercise them with different data types. Note however, that they are not designed for real life applications but merely to wrap the underlying container in a common interface for this specific test! The different container implementations are described in more detail below.

The underlying container is a simple array allocated in the constructor using operator `new []`. The capacity of the array is not allowed to grow and there are no safety measures implemented. I.e. attempts to add data beyond the capacity are not intercepted and will result in a crash. The `reset()` method simply sets the element counter to 0. The `append()` method adds data by assignment and increments the element counter. This is clearly not an implementation any sane person would use. It's sole purpose is to play the role of the lower limit.

Similar to `ContArrayStatKeepAssign` but intercepts attempts to add data beyond capacity. Capacity is never increased, however. The append method simply issues an error message and refuses to add more data.

The most straightforward implementation. The `reset()` method simply wraps `std::vector::clear()` and `append()` just forwards to `std::vector::push_back()`.

Uses a `std::vector` internally. But as opposed to `ContStdVector` manages the size of the vector itself and adds new elements by assignment.

Uses a `std::vector<T*>` internally. Manages the size of the vector itself. Data is added by explicitly calling operator `new`. Memory is freed with `delete` before new data is inserted but only for reused addresses. Of course all memory is freed when the container is destructed. The `reset()` method simply sets the element counter to 0.

Also uses a `std::vector<T*>` internally. But as opposed to `ContStdVectorNew` uses placement `new`. This is expected to be faster and also makes it unnecessary to call destructors in the `append()` method. (Well, strictly this is not true if the contained data class allocates recourses. But this is not the case with our simple `Coordinate` class.) Again, all memory is freed when the container goes out of scope and `reset()` just sets the element counter to 0.

The Profiling Procedure

I aimed for a setup that resembles the requirements in real event processing while being as lean as possible at the same time. To this end I implemented a simple data class `Coordinate` which holds three `double` data members. It features the usual accessors and constructors. This class is not really useful for anything, the point is that it is not POD and thus allows for a more realistic scenario.

The different container implementations are exercised by the class `ContainerTester`. This class implements a simple event loop. For each event the `reset()` method of all container implementations is called. A random number of `Coordinate` instances is created in each event. The minimum and maximum numbers of `Coordinate` instances in each event is configurable. The append methods of all containers are called for each instance of `Coordinate` in the event. In order to make all calls visible to the profiler, there is a wrapper method for each containers `append()` method. Otherwise inlined container specific implementations would not show up as function calls. Since there is a wrapper call for each container the comparison is fair, independent of inlining in the container implementation. Immediately after a `Coordinate` instance is added to the containers all containers are accessed and the retrieved value is compared to the original data. This ensures data integrity, i.e. that all containers actually do contain what they are supposed to.

A simple `main()` implementation instantiates a `ContainerTester` and calls its `run()` method. The callgrind tool is used for profiling. Visualisation is done using kcachegrind.

The source code of the standalone profiling suite can be downloaded here. The compiler flags used in the test are `g++ -g  -O3 -c -Wall -Werror -ansi -pedantic`. The test was performed on SLC 4.4 X86_64 running on a AMD64 3000+ system.

Results

And now the ranking:

1. `ContArrayStatKeepAssign`: 2.74%
2. `ContArrayStatProtectedKeepAssign`, `ContStdVector`: 3.47%
3. `ContStdVectorKeepAssign`: 6.03%
4. `ContStdVectorPlacementNew`: 6.4%
5. `ContStdVectorNew`: 40.54%

You can view the profiling diagram here.

Evaluation of `std::list`

On demand decoding of the raw buffer is anticipated at least for the OT. This suggests that our measurement container implementation should support performant insertion behaviour. Naturally, `std::list` comes to

mind. It has the advantage that iterators are not invalidated by insertion (although the semantics of an iterator range might change) and that insertion complexity is constant time. Random access is not possible, but this might not be an issue on its own. What we defintely need, however, is a binary search on the sorted container using `std::lower_bound()`. And this algorithm is expected to be much slower without random access.

The performance of `std::list` was tested with the same profiling procedure as the other containers. To this end I added a `ContStdList` container similar to `ContStdVector` to the test suite. The visualisation of the profile can be seen here. The bottom line is that `std::list` is roughly eight times slower than the corresponding `std::vector` implementation. This alone probably renders it unusable for the principal container implementation.

Furthermore I did a comparison of the performance of `std::lower_bound()` on `std::list` and `std::vector`. The test was performed by creating a `std::list` and a `std::vector` of size 1000 containing the numbers from 0-999. Then the `std::lower_bound()` algorithm was used 10000 times on both containers with random numbers in the range 0-999. This picture says it all. The call for `std::vector` does not show up, even with a callee threshold of 1%. So `std::lower_bound` is about a hundred times slower on `std::list` than on `std::vector`.

Conclusion

As expected, the completely unprotected static array implementation is the fastest. But it only represents a lower limit and is unusable in practice. The straightforward `std::vector` and the array implementation with boundary check share the second place. However, the latter is less flexible since does not implement dynamic capacity.

So we arrive at the conclusion that the best solution is actually the most straightforward one, using the STL `std::vector`. I like that result.

Concerning `std::list` it was shown that it is considerably slower compared to `std::vector` in `push_back()` and `clear()` operation. This is not surprising since `std::list` dynamically allocates memory for each element. Even worse, `std::list` is about a hundred times slower than `std::vector` in binary searches with `std::lower_bound()`. This most likely renders it unsuitable for our purposes and we have to find a different solution if we need performant insertion behaviour.

-- Main.krinnert - 04 June 2007

This topic: LHCb > ATeamDataContainerProfiling
Topic revision: r2 - 2007-06-04 - KurtRinnert