A fast vector with mremap

(Szybki vector z mremap)

Wojciech Oziębły

Praca inżynierska

Promotor: dr Marek Szykuła

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

8 lutego 2019

Abstract

One of the most commonly used data structure in C++ is a *vector*, which is a wrapper of a dynamically allocated array. Aside from automatic memory management, it allows to dynamically increase the size of the held array, with reallocation of the contained memory block when needed. The STL implementation of this data structure (*std::vector*) executes this reallocation utilizing the scheme *allocate*, *move*, and *deallocate*. Due to the importance of a vector and its application in almost every C++ project, there also exist a number of other implementations utilizing various optimizations. In this work, we develop a novel vector implementation whose main idea is to directly use `mremap` syscall of a Linux system. It is then possible to check whether the operating system can expand the specified memory block in place, without copying the memory, or effectively reallocate the memory block by changing the virtual address mapping. This can yield a significant efficiency improvement compared to the traditional copying approaches, but requires suitable low-level management in order to satisfy the general specification of a vector used with non-trivial data types. We perform a detailed benchmark of our vector with the four most popular available implementations. The results show that our reallocation idea together with carefully optimized code yields the fastest vector implementation in a typical vector usage.

Wektor jest najczęściej wykorzystywaną strukturą danych w języku C++. Opakowuje on dynamicznie allokowaną tablicę. Oprócz automatycznego zarządzania pamięcią, pozwala on na dynamiczne zwiększanie rozmiaru trzymanej tablicy, realokując trzymany blok pamięci kiedy jest to wymagane. Podstawowa implementacja tej struktury znajdująca się w bibliotece standardowej (*std::vector*) wykonuje tę realokację wykorzystując schemat *alokacja*, *przeniesienie*, *dealokacja*. Z powodu znaczenia tej struktury dla niemal każdego projektu w C++, istnieją również różne inne implementacje zoptymalizowane na różne sposoby. W tej pracy proponujemy nową implementację wektora, gdzie główną ideą jest bezpośrednie użycie wywołania systemowego `mremap` w systemach Linux. Dzięki temu jest możliwe sprawdzenie czy system operacyjny potrafi rozszerzyć podany blok pamięci w miejscu bez kopiowania pamięci lub przenieść blok w czasie stałym zmieniając tylko mapowanie jego adresu wirtualnego na fizyczny. To może dać znaczący wzrost wydajności w porównaniu z tradycyjnymi podejściami kopiującymi, ale wymaga odpowiedniego niskopoziomowego zarządzania by zachować zgodność implementacji z ogólną specyfikacją wektora używanego z nietrywialnymi typami danych. W pracy wykonujemy szereg testów wydajności nowego wektora razem z czterema popularnymi ogólnodostępnymi implementacjami. Wyniki wskazują, że zastosowany pomysł na realokację pamięci w połączeniu ze starannie zoptymalizowanym kodem daje najszybszą implementację wektora w typowych zastosowaniach.

Contents

Chapter 1

Introduction to the vector data structure

A *vector* is the most commonly used and general data structure in C++. Its purpose is simple: management of a continuous dynamic array of objects. Because of its universal use and general applicability, a vector is required to be fast, memory efficient, and reliable. It is supposed to work effectively as a container for billions of `int` variables, as well as for a few large objects of an abstract structure. Almost all software written in C++ relies heavily on vectors, hence an efficient implementation of this data structure is highly desirable.

A vector stores all objects in a contiguous memory block, which allows constant time access to the elements without any intermediate computation, i.e. accessing is reduced to indexing a raw array. Vectors allow dynamically adding and erasing elements, where the storage is automatically expanded when needed. To do this efficiently, the capacity of a vector is usually greater than the actual number of stored elements. The key parameter of any vector implementation is the *growth factor*, which defines the expansion rate of the capacity when additional memory is required. With such an allocation strategy, adding an element to the back of the vector has amortized constant time. The most common value of the growth factor is 2. Yet, as the reallocation process is often very costly, it renders vector not suitable for certain tasks, e.g. real-time systems or low memory consuming programs.

Despite that the vector idea is simple, there are several details that can affect its efficiency. Indeed, because it is so crucial data structure, a number of different implementations are used, believing that they are more efficient than the standard implementation.

1.1 Existing vector implementations

1.1.1 std::vector

The most common implementation is the standard one provided in STL. Although its technical details may differ among C++ compilers, they generally follow the same pattern. With the default allocator, all allocations are done with `operator new` and deallocations with `operator delete`. A reallocation is done using the scheme *allocate, move, deallocate*, where the *allocate* and *deallocate* phases are done by the given allocator. The growth factor is constant and equal to 2.

1.1.2 folly::fbvector

`Folly::fbvector`[3] is a part of *folly* library developed by Facebook. It has a similar interface for the allocator to `std::vector`, yet by default, it utilizes `jemalloc` for allocations, deallocations, and reallocations[4]. It is worth noting that `folly::fbvector` utilizes `jemallocs xallocx` function, which tries to reallocate memory in place. The growth factor depends on the size of the current array. The initial growth is at least 64 bytes, probably to fill a whole cache line. For not in-place reallocations (small ones) and big memory blocks (at least $4096 \cdot 32$ bytes), the growth factor is equal to 2. Otherwise, it is equal to 1.5, as this allows to reuse previously allocated memory. `Folly::fbvector` developers believe that such a strategy is more memory friendly and efficient[4]. Additionally, `folly::fbvector` uses `memcpy` to move objects with a type decorated with `folly::IsRelocatable`.

1.1.3 boost::container::vector

`Boost::container::vector`[5] is the least specialized version of a vector in `boost::container`. It has lower exception guarantees in order to improve the performance of the container[6]. As a default allocator, it uses the boost version of `dlmalloc`[7], which allows expanding memory block forward as well as backward. To achieve this, the allocator stores a chain of allocations instead of a single allocation block. The growth factor by default is equal to 1.5, but it is possible to change its value at compilation time. In my opinion `boost::container::vector` has the most complex implementation out of all the vectors considered in this chapter.

1.1.4 eastl::vector

`Eastl::vector`[8] is a part of the Electronic Arts Standard Template Library (EASTL) developed by Electronic Arts company. EASTL was designed especially as a game development library. It is considered to be more suited for console platforms[9] than other STL implementations. The default allocator in EASTL requires the user to define a global `eastl` version of `operator new` that would be used for allocations. The *Eastl::vector* implementation is simple and similar to that of `std::vector`, as it also utilizes *allocate*, *move*, *deallocate* scheme. Yet, it contains only an EASTL version of STL functions. The growth factor is constant and is equal to 2.

Chapter 2

Rvector implementation

`Rvector`[10] implements the whole `std::vector` interface specified in C++17 with a few minor differences. The most important one is that `rvector` does not have any exception guarantees. Yet, as *gcc* (and *Clang*) follows *Itanium ABI*[11] in regard of exception handling, other vector implementations have a guarantee of zero overhead when exception throwing does not occur, which is the case during the benchmarks. The main idea of `rvector` is the use of syscall `mremap` to do reallocations. To make it possible, all allocations of size greater than the *page size* (4KB is a normal page size on most of the architectures) are done using syscall `mmap`. Allocations of a smaller size are done using `malloc` to reduce the space consumption overhead; in this case, the standard *allocate*, *move*, *deallocate* scheme is utilized.

```
template<typename T>
T* allocate(size_type n) {
    if(n > map_threshold<T>)
        return (T*) mmap(NULL, n*sizeof(T),
                PROT_READ | PROT_WRITE,
                MAP_PRIVATE | MAP_ANONYMOUS,
                -1, 0);
    else
        return (T*) malloc(n*sizeof(T));
}
```

Listing 2.1: rvector allocation.

In `rvector`, array handling depends on element type *type traits*. It is especially important whether a certain type is *trivially movable* or not. A trivially movable type is that having a *default moving constructor* To achieve compilation time dispatching that automatically detects this property of the type used with `rvector`, we use SFINAE feature with the following policies:

```
template<typename T, typename R = void>
using T_Move = std::enable_if_t<
                std::is_trivially_move_constructible<T>::value, R>;
template<typename T, typename R = void>
using NT_Move = std::enable_if_t<
                !std::is_trivially_move_constructible<T>::value, R>;

template<typename T>
using T_Move_a = std::enable_if_t<
                std::is_trivially_move_assignable<T>::value>;
template<typename T>
using NT_Move_a = std::enable_if_t<
                !std::is_trivially_move_assignable<T>::value>;

template<typename T>
using T_Destr = std::enable_if_t<
                std::is_trivially_destructible<T>::value>;
template<typename T>
using NT_Destr = std::enable_if_t<
                !std::is_trivially_destructible<T>::value>;
```

Listing 2.2: SFINAE policies.

`Rvector` uses `mremap` in two different ways. For trivially movable types, `MREMAP_MAYMOVE` flag is passed, which allows `mremap` to reallocate memory block to other address. When expansion in-place is not possible, `mremap` does reallocation by changing page table mapping from virtual address to memory page[1], which is very efficient for big memory blocks. Thanks to `MREMAP_MAYMOVE` flag, this reallocation is always successful.

```
template<typename T>
T_Move<T, T*> realloc_(T* data,
                        size_type length,
                        size_type capacity,
                        size_type n) {
    // move between malloc and mmap allocations
    if((n > map_threshold<T>) != (capacity > map_threshold<T>)) {
        T* new_data = allocate<T>(n);
        memcpy(new_data, data, length * sizeof(T));
        deallocate(data, capacity);
        return new_data;
    }
    else {
        if(capacity > map_threshold<T>)
            return (T*) mremap(data, capacity*sizeof(T),
                               n*sizeof(T), MREMAP_MAYMOVE);
        else
            return (T*) realloc(data, n*sizeof(T));
    }
}
```

Listing 2.3: rvector trivial type reallocation.

For non-trivially movable types, `MREMAP_MAYMOVE` cannot be used. Without this flag, `mremap` tries to expand the memory block in-place. This operation will fail when there is not enough free space in front of the provided address. In that case, a standard reallocation is done.

```
template<typename T>
NT_Move<T, T*> realloc_(T* data,
                        size_type length,
                        size_type capacity,
                        size_type n) {
    if(capacity > map_threshold<T>) { // try mremap fast reallocation
        void* new_data = mremap(data, capacity*sizeof(T),
                                n*sizeof(T), 0);
        if(new_data != (void*)-1)
            return (T*) new_data;
    }
    T* new_data = allocate<T>(n);
    std::uninitialized_move_n(data, length, new_data);
    destruct(data, data + length);
    deallocate(data, capacity);
    return new_data;
}
```

Listing 2.4: rvector non-trivial type reallocation.

The growth factor is equal to 2. The allocation size is also fixed with the following function:

```
template <typename T>
size_type fix_capacity(size_type n) {
    // minimal allocation is 64 bytes as in folly::fbvector
    if(n < map_threshold<T>)
        return std::max(64/sizeof(T), n);
    // if requested capacity is greater than page size,
    // it is rounded to the next multiple of page size
    return map_threshold<T> * (n/map_threshold<T> + 1);
}
```

Listing 2.5: fix capacity.

`Rvector` is equipped with a few additional functions:

- `fast_push_back`, `fast_emplace_back` – does not check the capacity bound. They can be used in *reserve*, *fill* pattern, where we already know that enough capacity has been ensured for the forthcoming `push_back` operations.

- `safe_pop_back` – it is `pop_back` which checks if the target vector is empty; if so, it does nothing.

2.1 The package

The implementation together with the benchmark package is available at:
`https://github.com/Bixkog/rvector`
The requirements are:

- *gcc* in version at least 7.4.0, and
- (to use the following instruction) *cmake* in version at least 3.5, *git*, and *make*.

```
git clone --recursive https://github.com/Bixkog/rvector.git
cd rvector
sudo bash install.sh
```

Listing 2.6: Installation

After installation, you can add `#include <rvector/rvector.h>` to your source files and ensure that C++17 is enabled. You may also skip the installation and directly copy `rvector.h` and `allocator.h` into your project.

To run the tests and the benchmarks you can do the following:

```
cd rvector
mkdir build && cd build
cmake ..
make
./runUnitTests
./runBenchmarks
```

Listing 2.7: Benchmarks and unit tests

Chapter 3

Benchmarks environment

3.1 Unit tests

Each public function has been unit tested using `gtest` library[12]. The tests are run with a few different object types, with one of them being custom `TestType` designed to check the correctness of object creation and destruction in `rvector`.

```cpp
struct TestType
{
int n;
int* p;
static int aliveObjects;

TestType(int a = 5,
         int b = 1) noexcept
: n(a),
p(new int(b)) {
    aliveObjects++;
}

TestType(const TestType& other)
    noexcept
: n(other.n),
p(new int(*other.p)) {
    aliveObjects++;
}

TestType(TestType&& other)
    noexcept
: n(other.n),
p(other.p) {
    aliveObjects++;
    other.p = nullptr;
}

~TestType() {
    delete p;
    aliveObjects--;
}

TestType& operator =
    (const TestType& other)
    noexcept {
    n = other.n;
    if(!p) p = new int();
    *p = *other.p;
    return *this;
}

TestType& operator =
    (TestType&& other) noexcept {
    n = other.n;
    std::swap(p, other.p);
    return *this;
}
...
};
```

Listing 3.1: TestType

The tests contain trivially and non-trivially movable types, and small (`malloc` allocations) and big (`mmap` allocations) sizes, to check all branches of `rvector` memory handling.

3.2 System environment

All tests were performed on an Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz processor with 8GB of RAM. The operating system was Ubuntu 16.04.5 LTS, kernel version Linux 4.4.0-141-generic. The compiler was g++ 7.4.0 with optimization level: -O3.

3.3 VectorEnv

In order to test the efficiency of vectors in complex, more realistic situations, we designed a dedicated vector benchmark environment template.

```
template <template<typename> typename V, typename... Ts>
class VectorEnv;
```

Listing 3.2: VectorEnv declaration.

For a given container type `V` (e.g. `std::vector, rvector`) and element types `Ts`, it creates the environment `std::tuple<V<V<Ts>>...> env`. It contains all the vectors created during the benchmark. With the environment created, a simulation can be run for a given number of iterations. In each iteration, for each element type, a random action is dispatched. Actions complexity depends on the iteration number so that late iterations are more expensive.

```
void RunSimulation(int iter = 1000) {
    for(int i = 0; i < iter; i++) {
        BenchTimer bt("Simulation");
        (dispatch_action<Ts>(i), ...);
    }
}
```

Listing 3.3: RunSimulation.

To gather time data, we used custom class `BenchTimer`, where objects of that type utilize constructor and destructor methods to record the time spent in their scope.

3.4 Simulation actions

A simulation consists of the following *actions*:

construct action

```
template <typename T>
void construct_action(int i) {
    auto& typed_env = std::get<V<V<T>>>(env);
    std::uniform_int_distribution<> q_dist(1, 3);
    std::uniform_int_distribution<> size_dist(1, i +
     10);

    int q = q_dist(gen);

    while(q--) {
        int size = size_dist(gen);
        typed_env.emplace_back();
        BenchTimer bt("construct");
        while(size--)
            typed_env.back().emplace_back();
    }
}
```

Listing 3.4: Construct action.

This creates a few small vectors. The elements are `emplaced` one by one in order to check the speed of small memory block reallocations.

push_back action

```
template <typename T>
void push_back_action(int i) {
    auto& typed_env = std::get<V<V<T>>>(env);
    if(typed_env.size() == 0) {
        construct_action<T>(i);
        return;
    }
    std::uniform_int_distribution<> q_dist(1,
    typed_env.size() / 3 + 1);
    std::uniform_int_distribution<> pick_dist(0,
    typed_env.size()-1);
    std::uniform_int_distribution<> size_dist(1, i *
     100);
    int q = q_dist(gen);

    BenchTimer bt("push_back");
    while(q--) {
        int pick = pick_dist(gen);
        int size = size_dist(gen);
        while(size--)
            typed_env[pick].emplace_back();
    }
}
```

Listing 3.5: Push_back action.

This adds many elements to a part of the environment (up to 1/3 with replacement). The other actions randomize the process in a similar manner.

pop_back action This pops elements (up to the size of the picked vector) from a part of the environment (up to 1/3 with replacement).

copy action This copies up to three vectors (with replacement) of the environment.

insert action This inserts elements (into a random position of a picked vector, up to the iteration number) into a part of the environment (up to 1/3 with replacement).

erase action This erases elements (between random positions in a picked vector) from a part of the environment (up to 1/3 with replacement).

3.5 Experiments

For each tested vector implementation, we run the following experiment function with a few different element types (*Ts*).

```
template <template<typename> typename V, typename... Ts>
void experiment(std::string name,
                int max_it = 1500,
                int tests = 10) {
    for(int seed = 12345512; seed < 12345512 + tests; seed++) {
        VectorEnv<V, Ts...> v_env(seed);
        v_env.RunSimulation(max_it);
    }
    // data gathering and saving
    ...
}
```

Listing 3.6: Experiment function.

Figure 3.1 shows the average number of bytes required by the environment (the sum of the lengths of all vectors multiplied by the size of the element types) and Figure 3.2 shows the maximum vector size at specific iteration. As both values are increasing with iterations, benchmarks with more iterations will show the efficiency of the vector operations on more fragmented memory and on longer vectors.

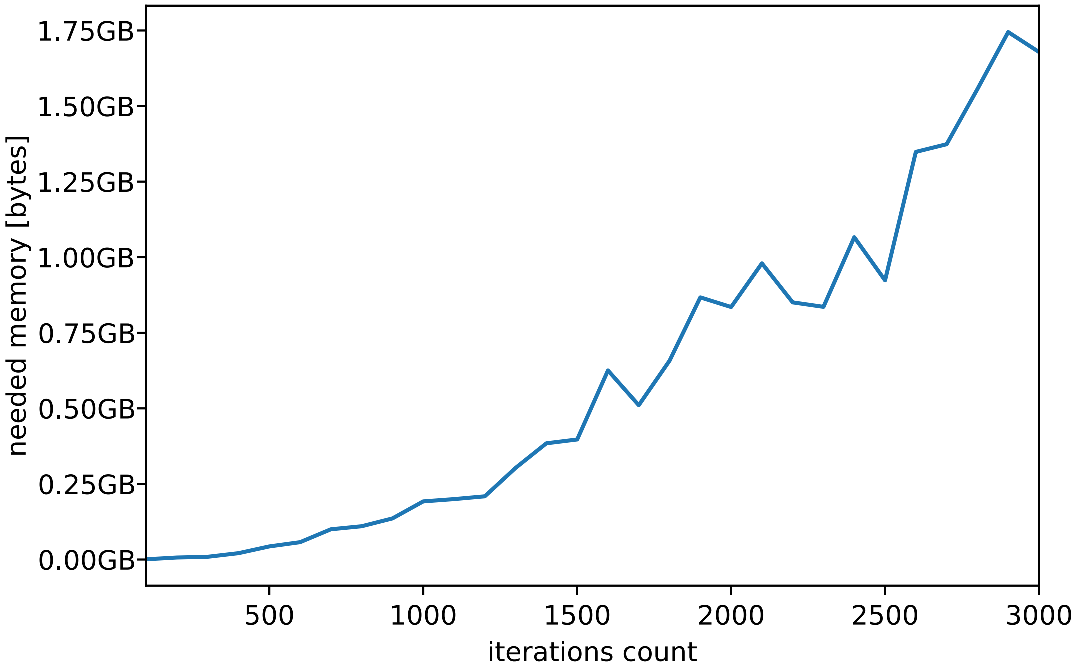

Figure 3.1: Required memory for type `int` [bytes].

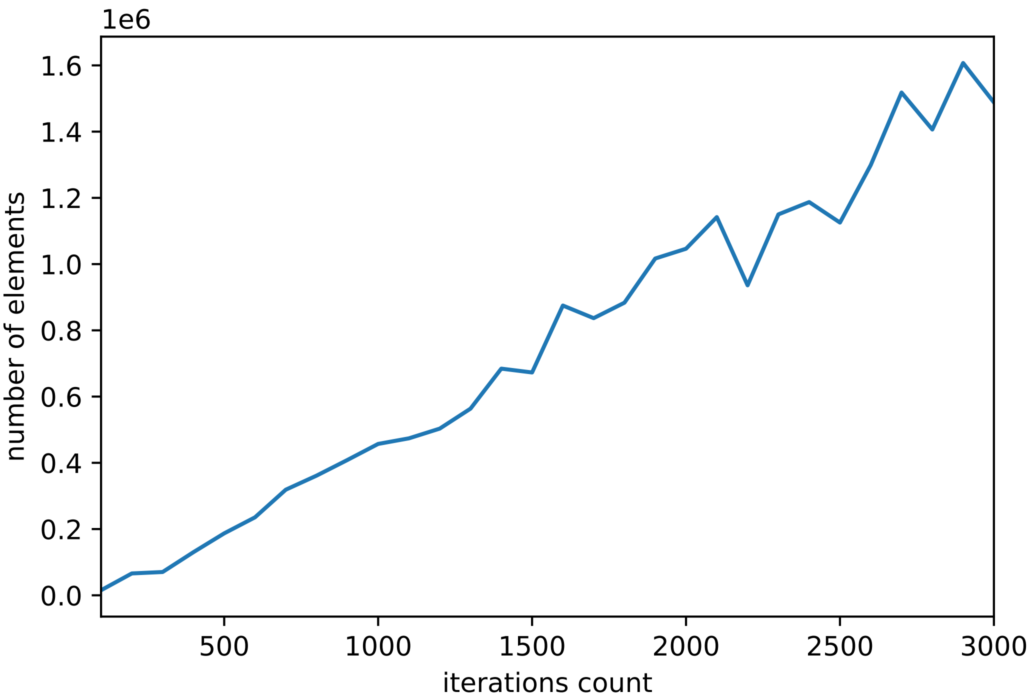

Figure 3.2: Maximum vector size for type `int`.

Chapter 4

Benchmarks results

Vector implementations that were considered in benchmarks are `boost::container::vector`, `eastl::vector`, `std::vector`, `folly::fbvector` and `rvector`. In the benchmarks, we used `libstdc++` implementation[2] of `std::vector`. `Eastl::vector` was provided with a standard allocator. All the vectors had the growth factor equal to 2, except `folly::fbvector` which has a custom dynamic *growth factor*.

4.1 Simple push_back benchmark

First of all, vectors have been tested for efficiency of a simple push_back loop execution. Figures 4.1 and 4.2 show the results of the benchmark that pushed back a certain number of elements to an initially empty vector. The tested element types were `int` and `std::string` (which was empty). In both cases, `rvector` did better than the other implementations when the number of elements pushed back was significant. For 100 million push_backs, for element type `int`, it was 105% faster than the second best vector `eastl::vector`. For `std::string` the difference was around 30%, thus less than for `int`. Yet, this shows that `rvector` in-place reallocation optimization alone provides a notable advantage over the other implementations.

Push_back is the most important use case of the vector data structure, yet this benchmark does not indicate vectors efficiency in more complex environments and programs when there are more vectors and other operations are also used. Hence, we need to check how it would behave when e.g. push_backs were not performed in a single time point, memory was fragmented, or allocations were done by other objects. To check vectors effectiveness in such situations, described in previous chapter `VectorEnv` was used in the next benchmarks.

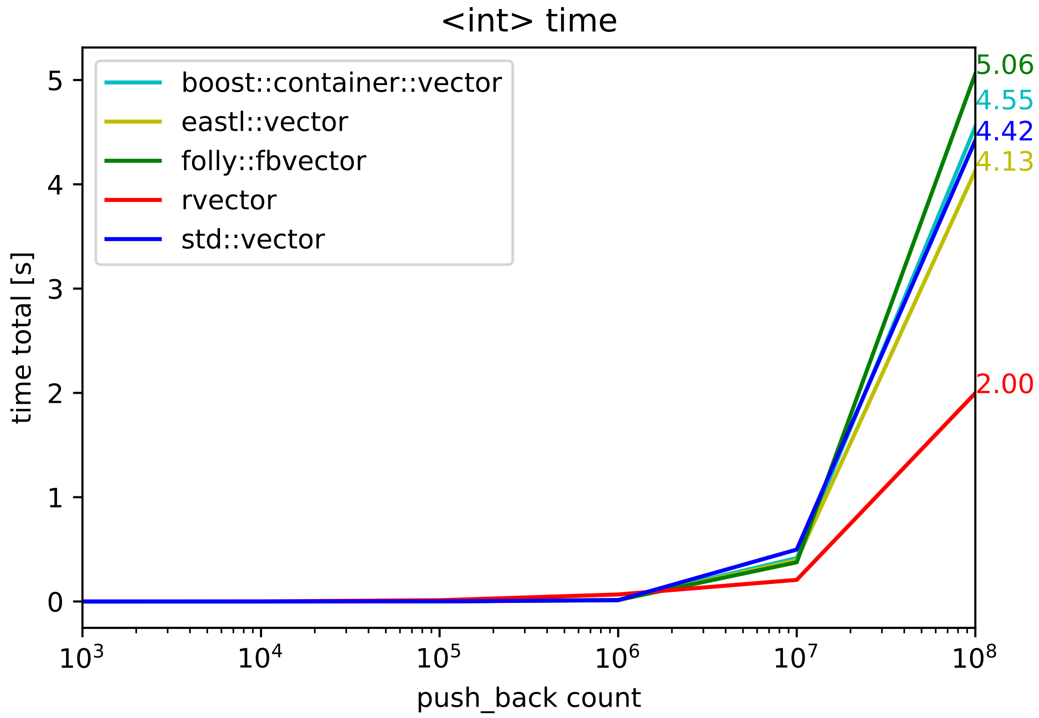

Figure 4.1: Push_back loop execution time for type `int`.

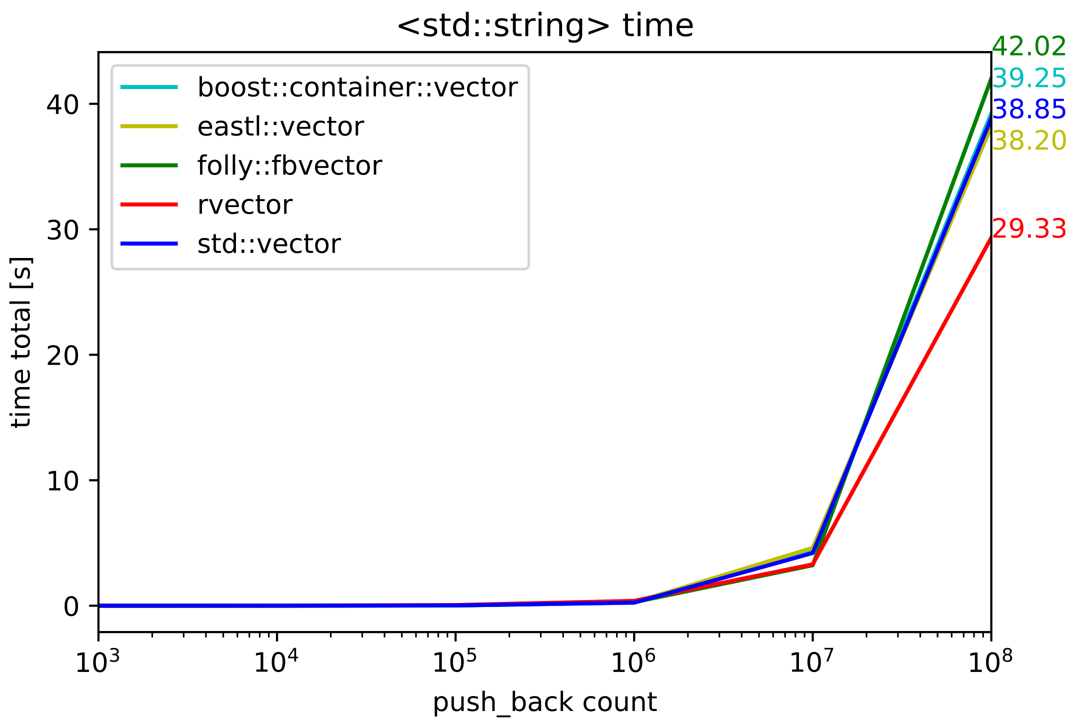

Figure 4.2: Push_back loop execution time for type `std::string`.

4.2 *VectorEnv* benchmarks

Using `VectorEnv` framework, we performed experiments for each of the tested vectors with the following element types. A *trivial type* is that it is trivially copyable and movable and has a default constructor [13].

- `int`
  The size of `int` is 4 bytes and it is a trivial type.

- `std::array <int, 10>`
  The size of `std::array <int, 10>` is 40 bytes and it is a trivial type.

- `std::string`
  The size of `std::string` is 32 bytes and it is not a trivial type. During the benchmarks, only empty strings are considered, so std::string objects do not allocate memory.

- `TestType`
  The size of `TestType` is 16 bytes and it is a non-trivial type. It is worth noting that each constructor and assignment operator has `noexcept` attribute, which allows vector implementations to use moving constructor instead of copy in case of reallocation. Also, the moving constructor is much faster than the copy constructor as it does not allocate memory.

- `std::string, int, std::array<int, 10>`
  In this case, three element types are tested at the same time.

4.3 Element types: int

Integer is one of tested element types to check how efficient a vector is for trivial, small objects. The benchmarks have shown that *std::vector* with a simple implementation containing intrinsic operations do much better than the more complex vectors (Figure 4.3).

For the most important vector action which is `push_back`, `rvector` does better than all the other vectors (Figure 4.4). Yet, the difference is not as large as for the other element types. Because in this benchmark vectors do not operate on big memory blocks, even for a large number of elements, reallocation optimizations do not induce much speed up.

Notably, `Rvector` falls behind the other implementations for the copy action (Figure 4.5). As for `int` element type (and all the other trivially copyable types), the copy constructor is equivalent to a single allocation and `memcpy`. During the benchmarks, `rvector` mainly uses `mmap` to allocate a memory block for a copy, which is much slower than an allocation with memory managers, which often do not have to do any *syscall* due to their local memory areas taken from the system in advance. Figure 4.7 shows that this is indeed the main factor of `rvector` copy being slower for element type `int`.

It turns out that *folly::fbvector* and *boost::container::vector* implementation of *pop_back* checks whether an object is trivially destructible at runtime instead of at compilation time. The results of *pop_back* action benchmark (Figure 4.6) show that the branching at *pop_back* makes it much slower when operating on trivial types.

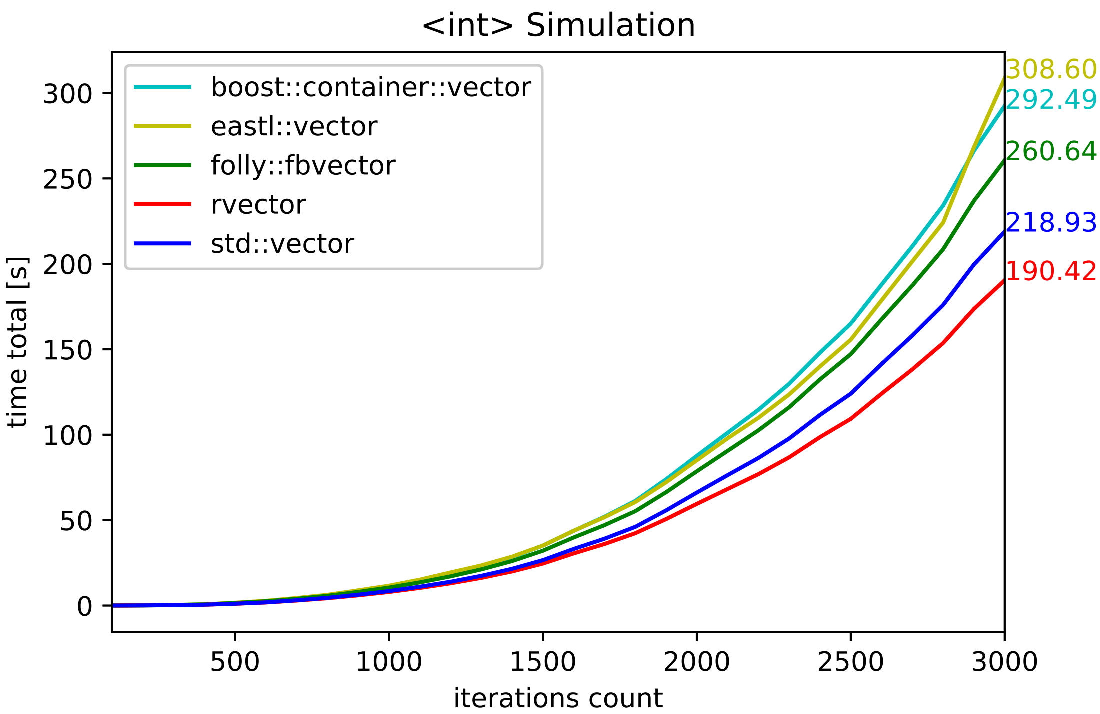

Figure 4.3: Total simulation time for element type `int`.

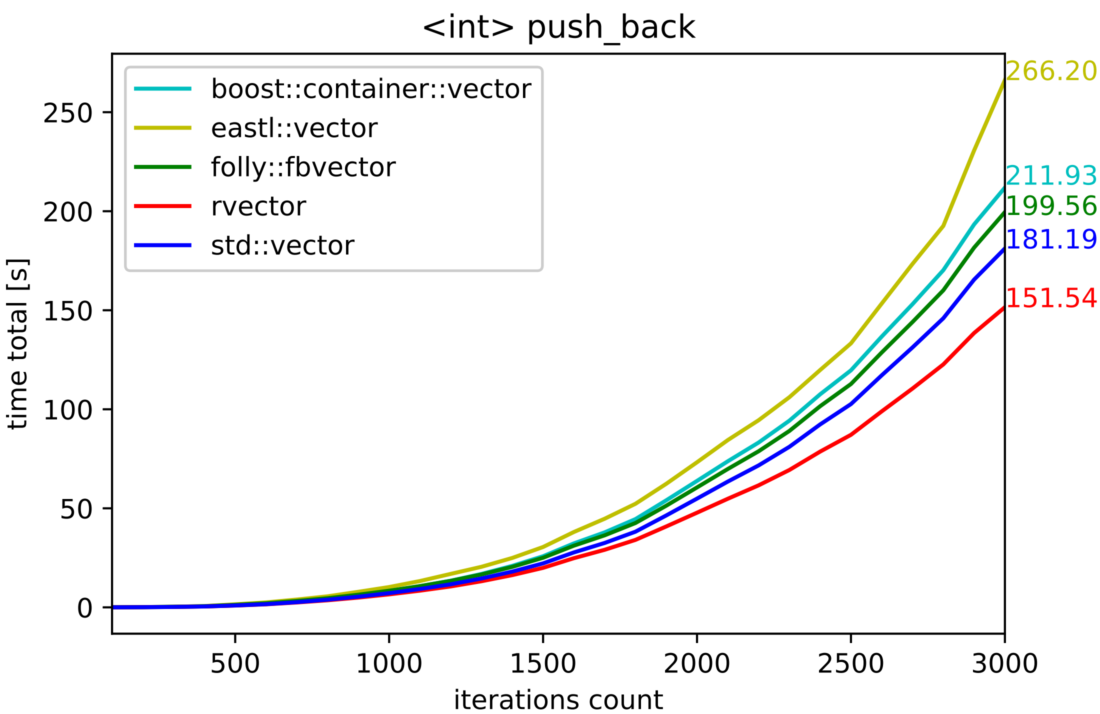

Figure 4.4: Push_back operation time for element type `int`.

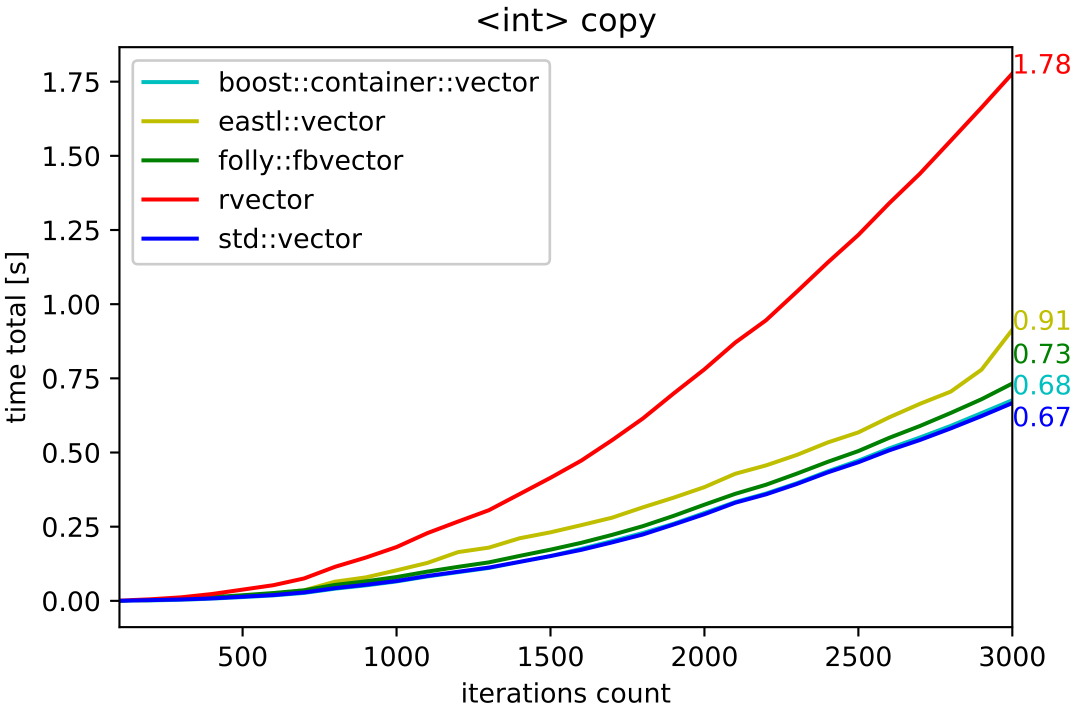

Figure 4.5: Copy action time for element type `int`.

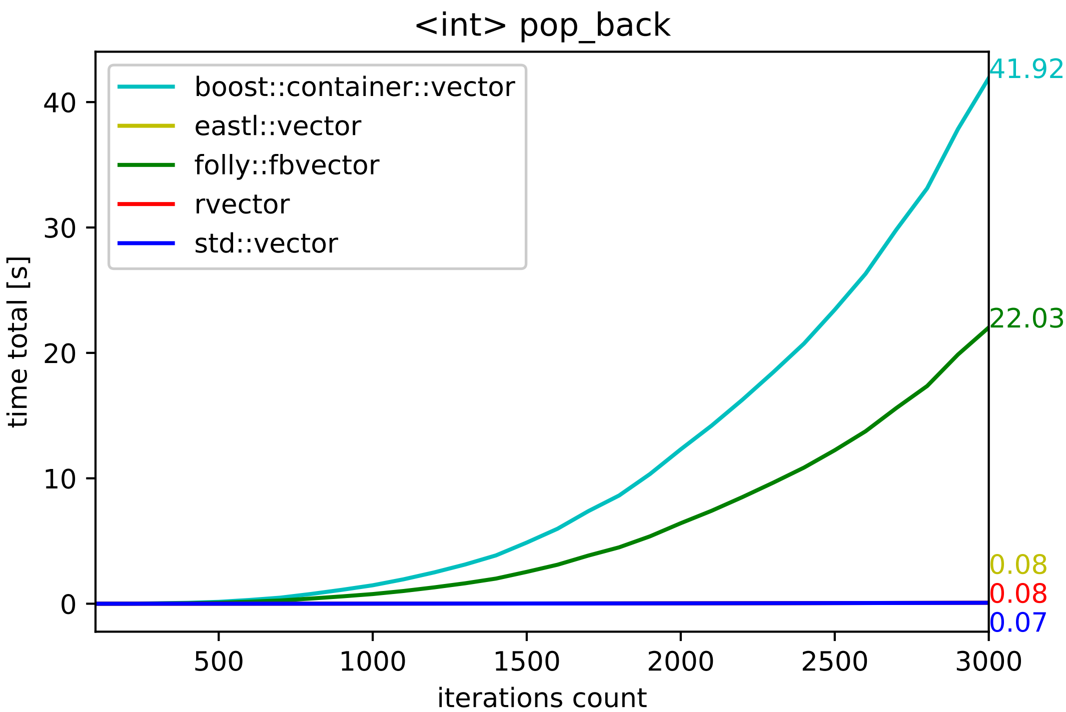

Figure 4.6: Pop_back action time for element type `int`.

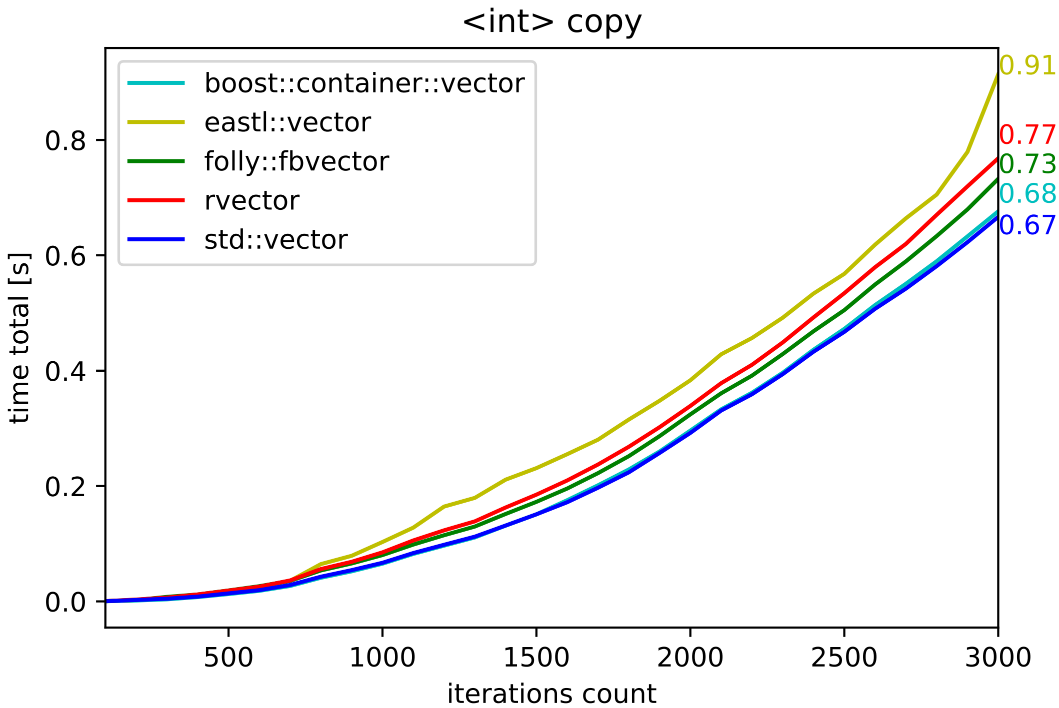

Figure 4.7: Copy action time for element type `int`. `Rvector` without `mmap`.

4.4 Element type: std::array

In order to test vectors on larger trivial objects, we used `std::array<int, 10>` as the element type. Figure 4.8 shows the total simulation time for this element type, and Figures 4.9–4.12 show the results for the separate operations.

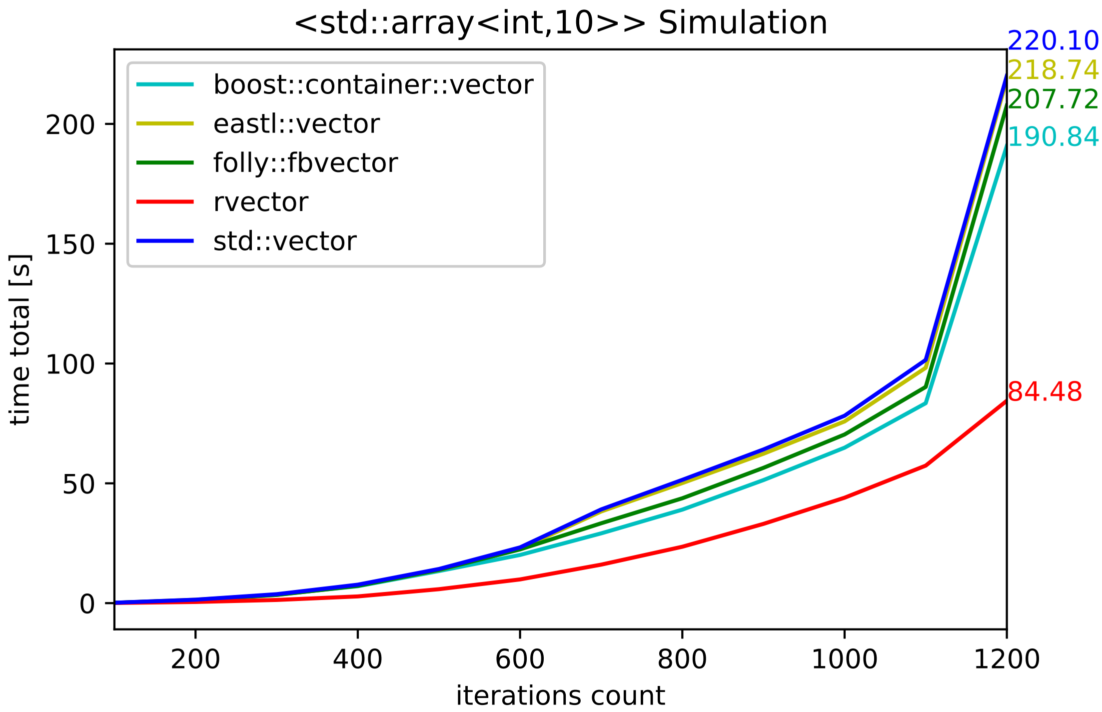

Figure 4.8: Total simulation time for element type `std::array<int, 10>`.

As it can be seen in Figure 4.9, `rvector` did all `push_back` operations in 54 seconds, yet runner-up `boost::container::vector` did them in 145 seconds. Reallocations using `mremap` are much faster for large memory blocks, making `rvector` far better than all the other implementations.

The results indicate that `rvector` is much more efficient when operating on large memory blocks and during *throttling*. *Throttling* is a situation of the system when programs take more memory than it is available, thus a swap file is used. It implies a much bigger *page fault* count, causing the whole system to slow down. A significant loss of efficiency of all vectors (except `rvector`) at late iterations indicates that the system has run out of physical memory. An explanation why `rvector` behaves better, in this case, is that `mremap`, in order to reallocate, does not have to read that memory, which is likely to be stored in the swap file. This behavior can be also seen for the other element types.

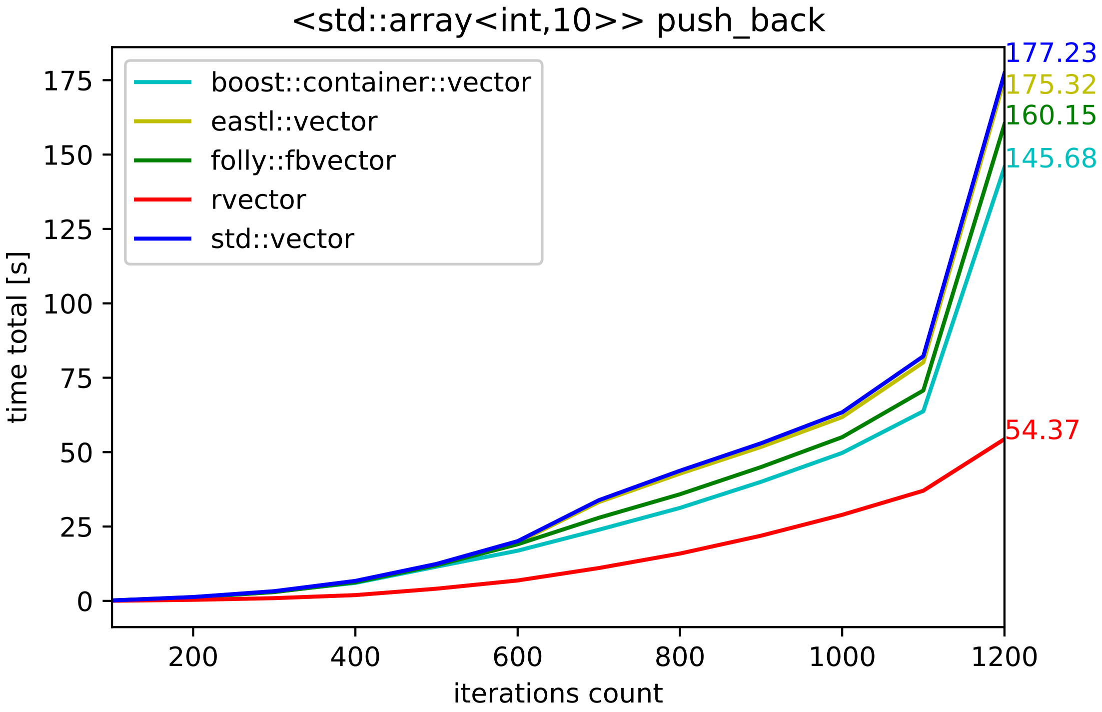

Figure 4.9: Push_back operation time for element type `std::array<int, 10>`.

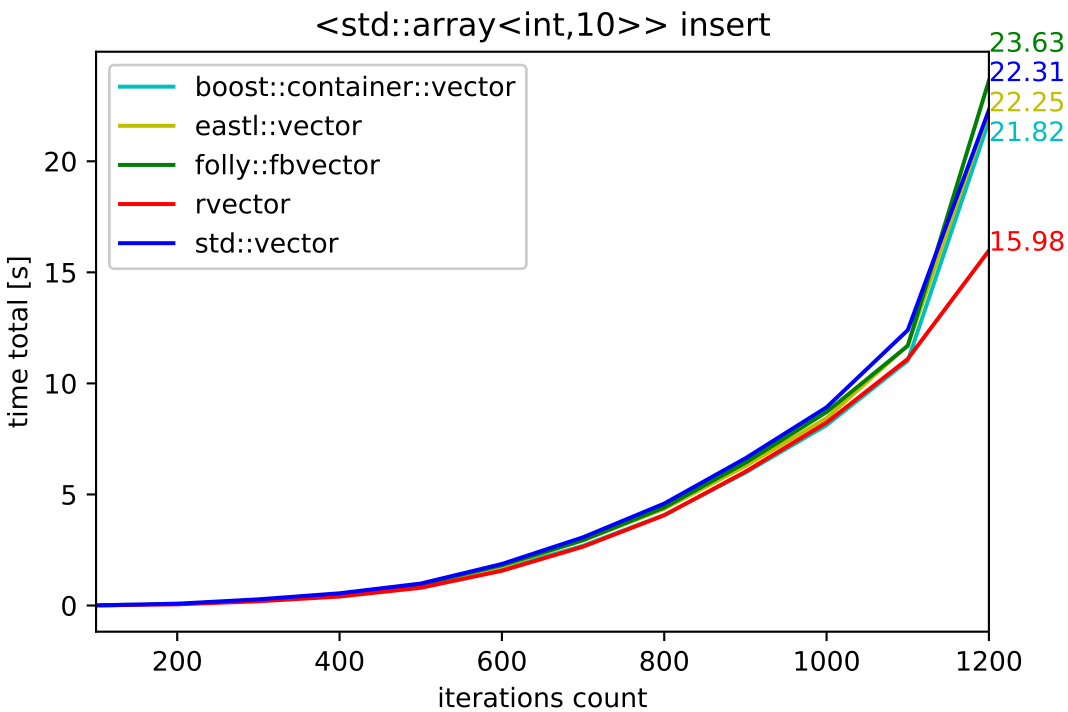

Figure 4.10: Insert operation time for element type `std::array<int, 10>`.

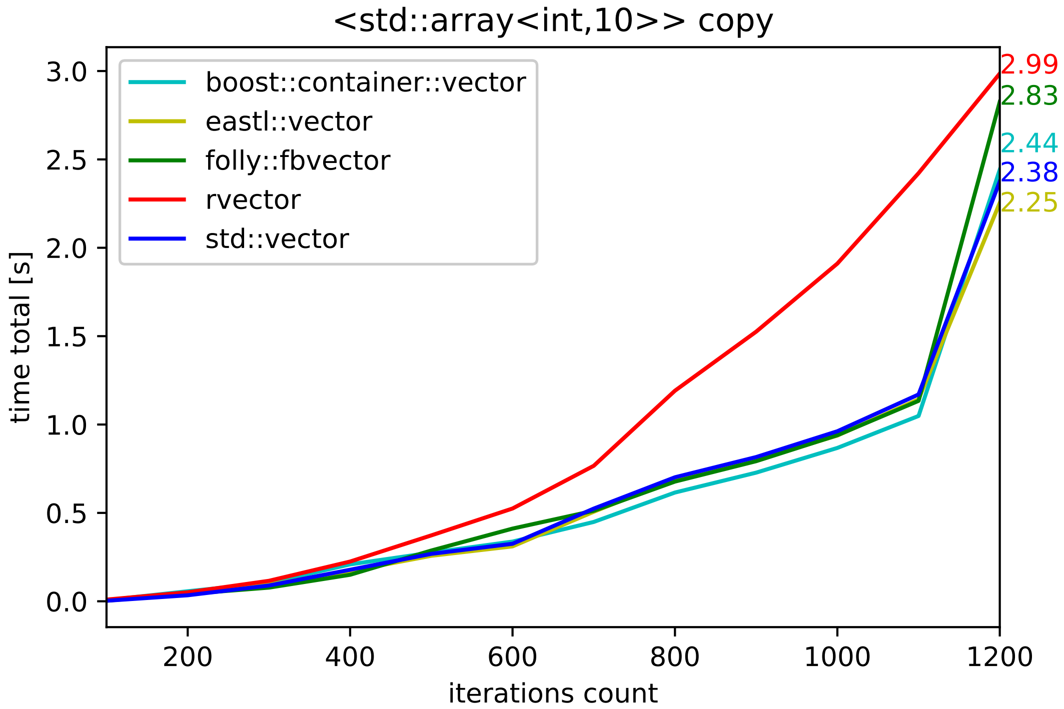

Figure 4.11: Copy operation time for element type `std::array<int, 10>`.

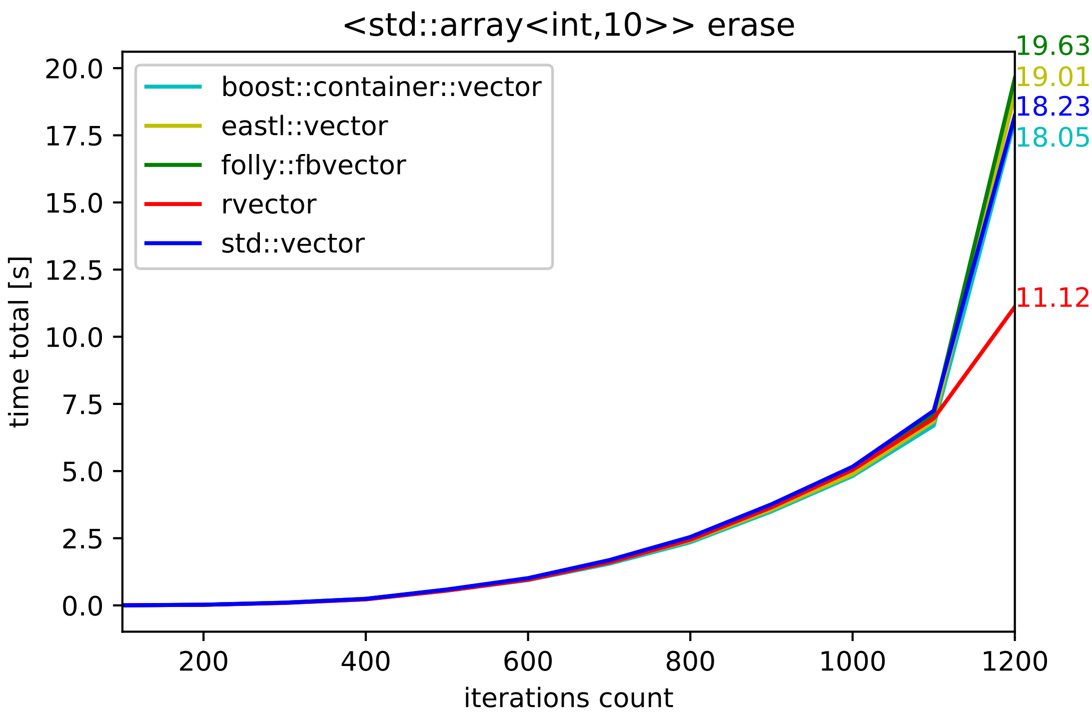

Figure 4.12: Erase operation time for element type `std::array<int, 10>`.

4.5 Element type: std::string

As `std::string` is not a trivial type, all vectors are obliged to move them with the move constructor, and not with `memcpy`. Hence, `rvector` will try to expand memory in place using `mremap` without `MREMAP_MAYMOVE` flag. On the one hand, the fast reallocation will not always occur, as it did with trivial types. On the other hand, regular reallocations are more expensive, because the objects must be moved with their move or copy constructors, not just with a simple `memcpy`. As it can be seen in Figures 4.13–4.17, the speed up induced by an in-place expansion with `mremap` is significant, and that throttling takes place after around $1,300$ iterations. During the early iterations, there is no significant difference between `boost::container::vector` and `rvector`, yet during throttling `rvector` is almost 100% more efficient than the runner-up.

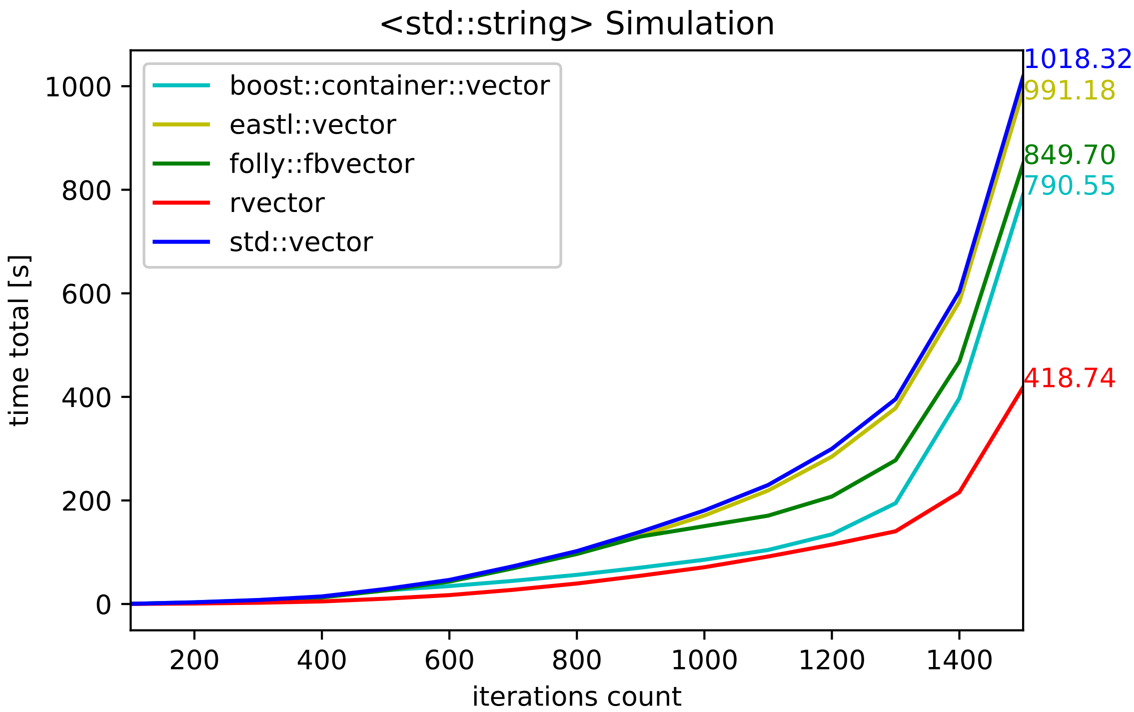

Figure 4.13: Total simulation time for element type `std::string`.

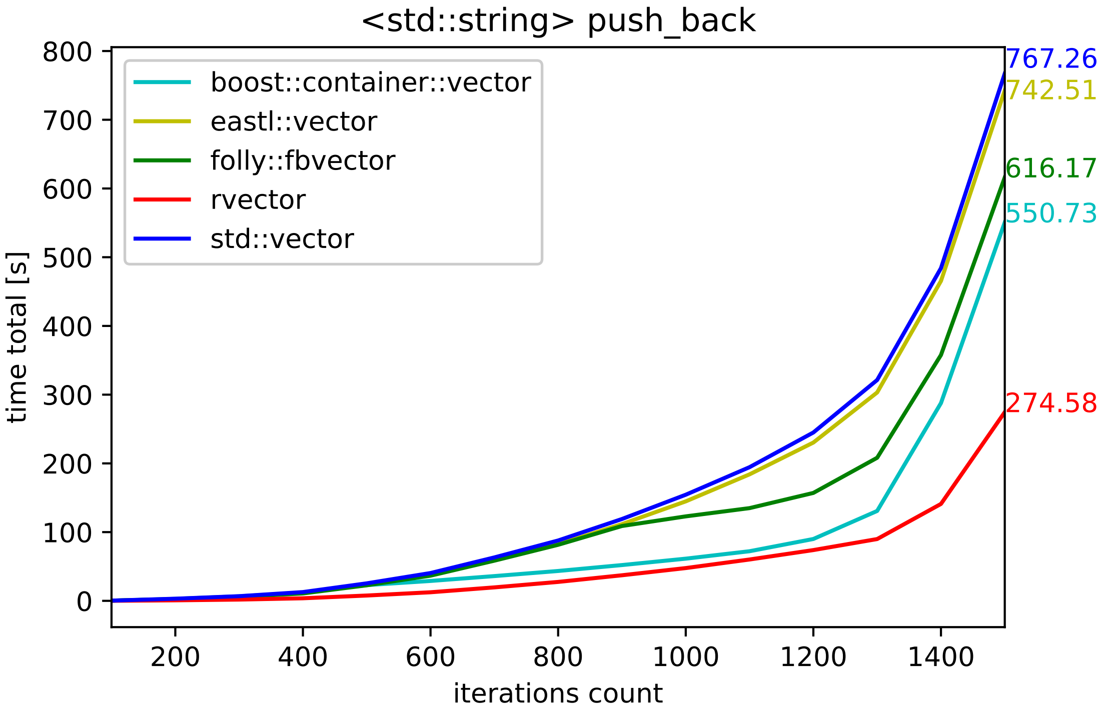

Figure 4.14: Push_back operation time for element type `std::string`.

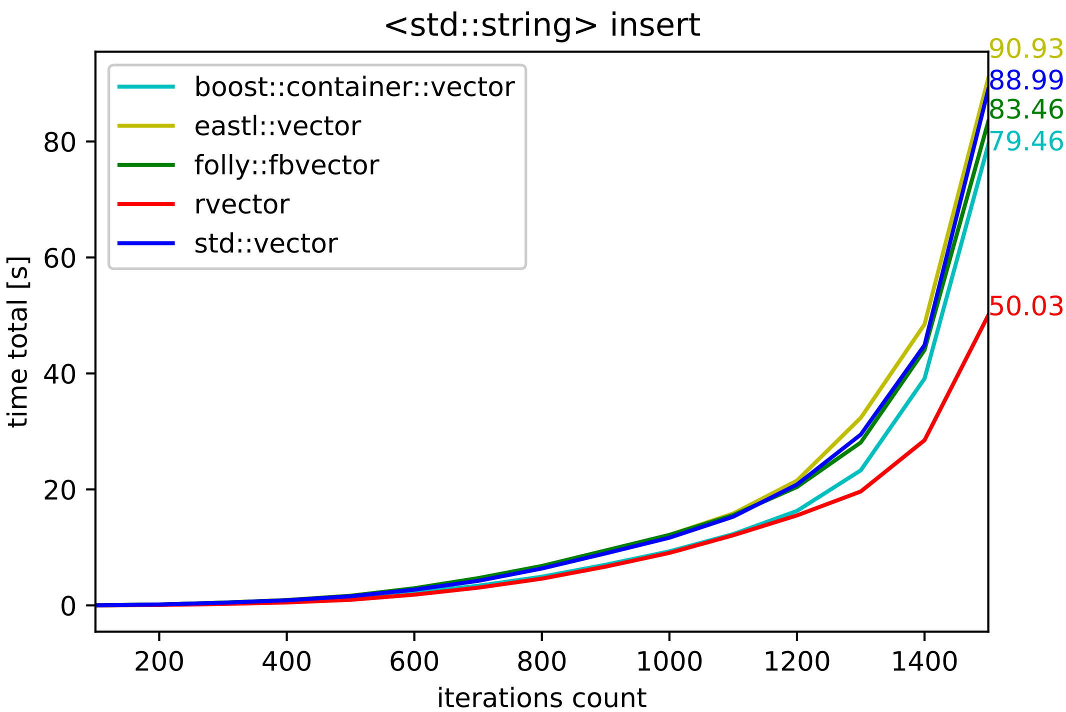

Figure 4.15: Insert operation time for element type `std::string`.

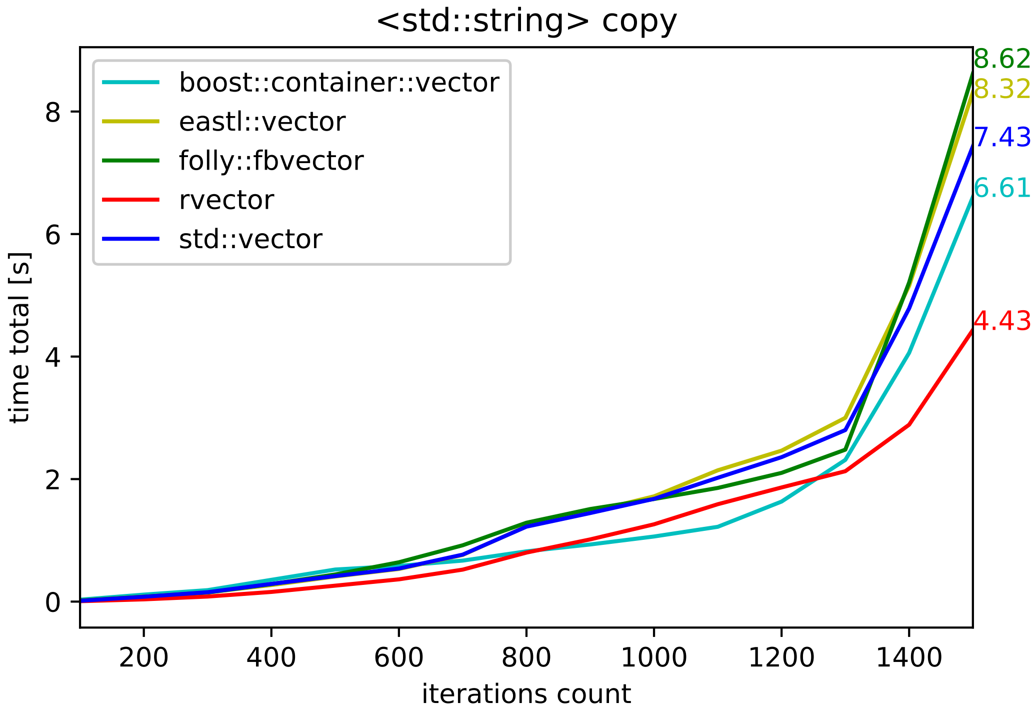

Figure 4.16: Copy operation time for element type `std::string`.

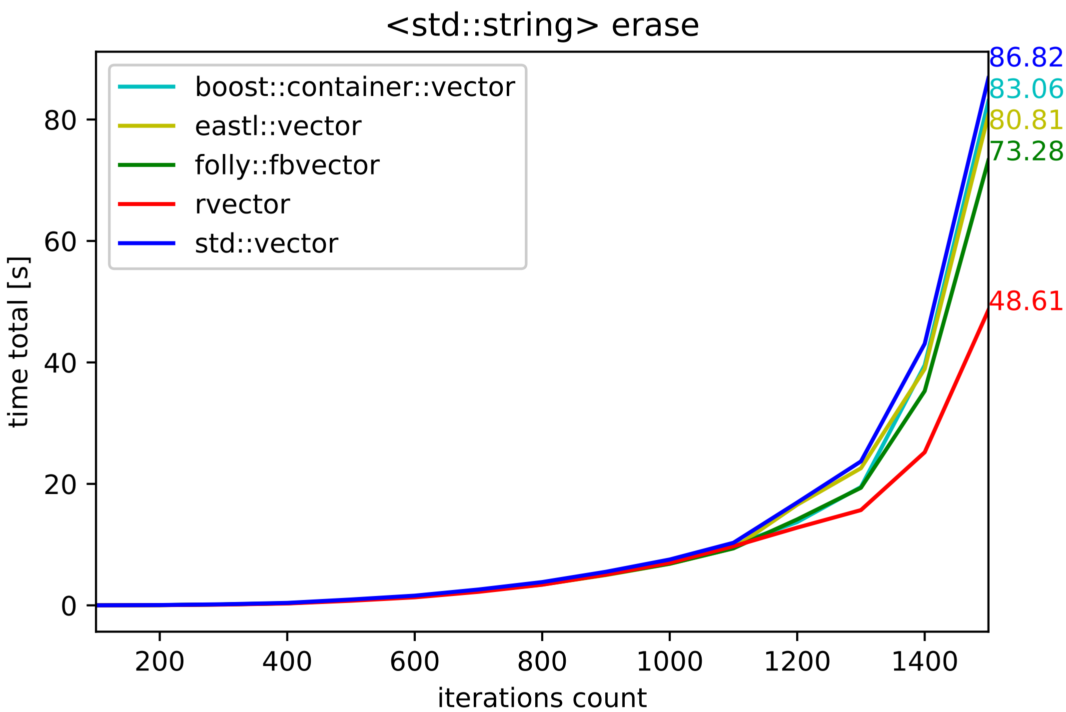

Figure 4.17: Erase operation time for element type `std::string`.

4.6 Element type: TestType

`TestType` class has been defined in Listing 3.1. It is a non-trivial type with the move constructor significantly faster than the copy constructor, as it does not any allocate memory.

The combined simulation is shown in Figure 4.18, and the results for particular action can be seen in Figures 4.19–4.22. It turns out that the gain from `mremap` is not as significant as for empty `std::string`, yet it still becomes greater when larger memory blocks are reallocated. The reason for such behavior may be increased fragmentation of memory, induced by `TestType` allocations. As it can be seen in Figure 4.25, the benchmark results on `std::string` long enough to allocate memory on the heap look similar to that in the case of `TestType`.

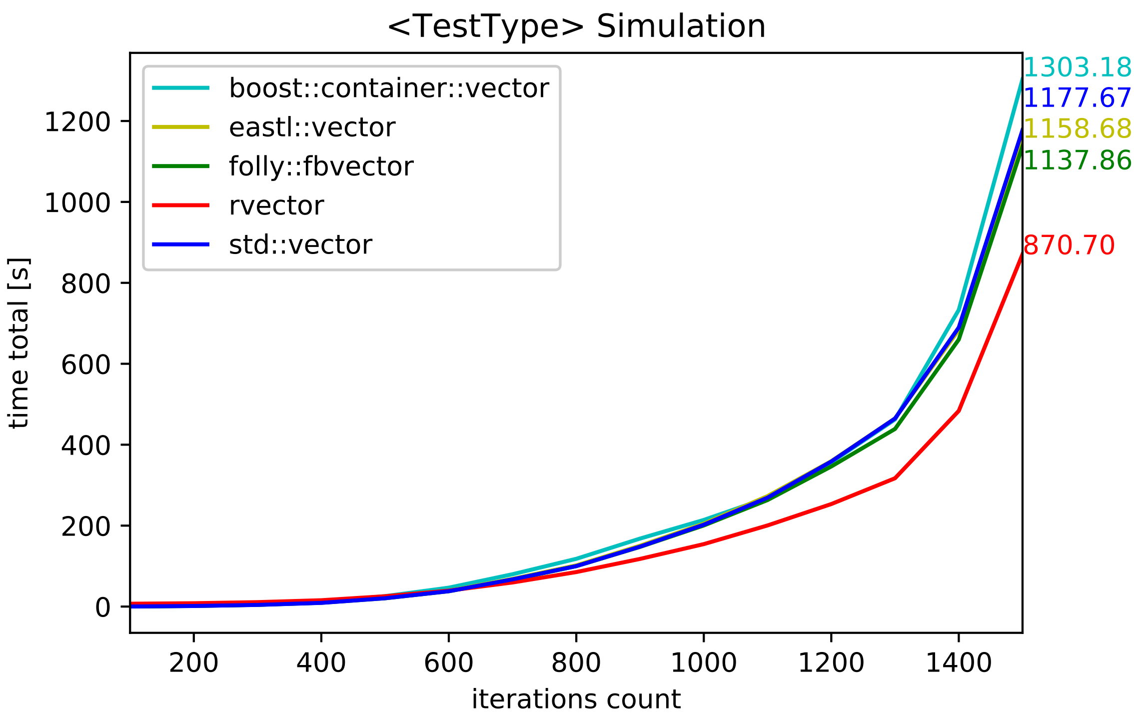

Figure 4.18: Total simulation time for element type `TestType`.

The `construct` action requires a separate comment. In this case, `rvector` got worse results than all the other vectors. Using a profiler, we found out that `malloc` functions called by `rvector` required around 60% more CPU cycles to finish than for `std::vector` calls. On the other hand, `rvector` did significantly better on the `copy` action. This behavior is most likely connected with the use of `mmap`. To test this hypothesis, we performed a benchmark of `rvector` with `mmap`/`mremap` optimization turned off. The results of a construct and copy action on this setup argue for that, and they can be seen in Figures 4.23 and 4.24, respectively.

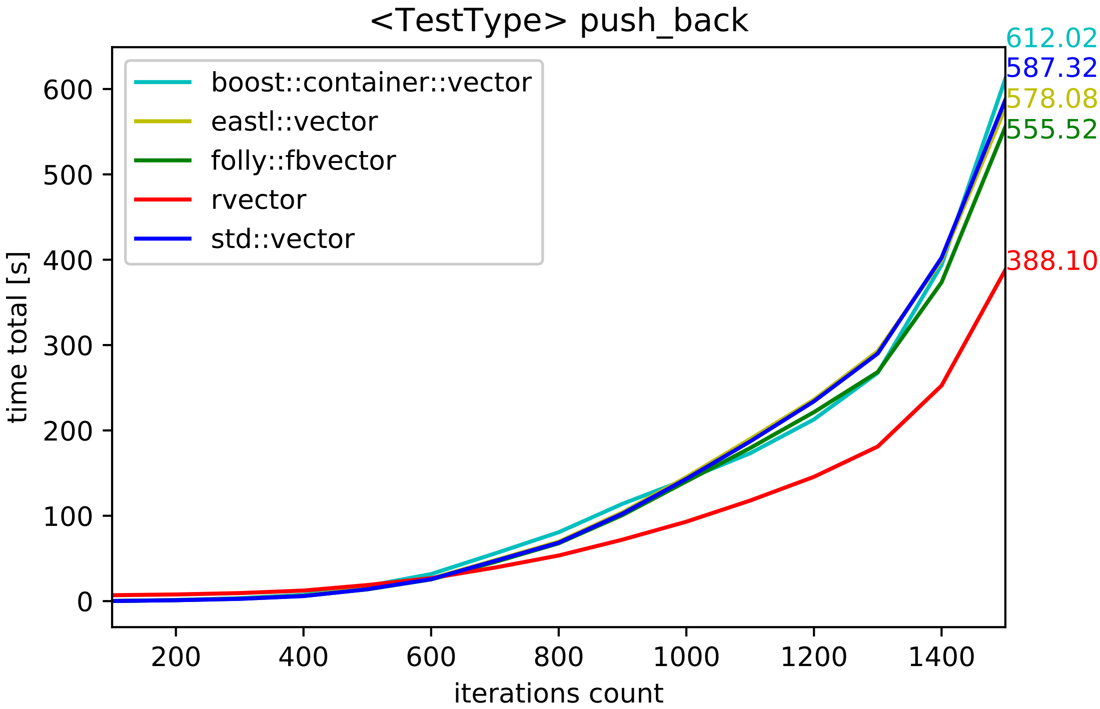

Figure 4.19: Push_back action time for element type `TestType`.

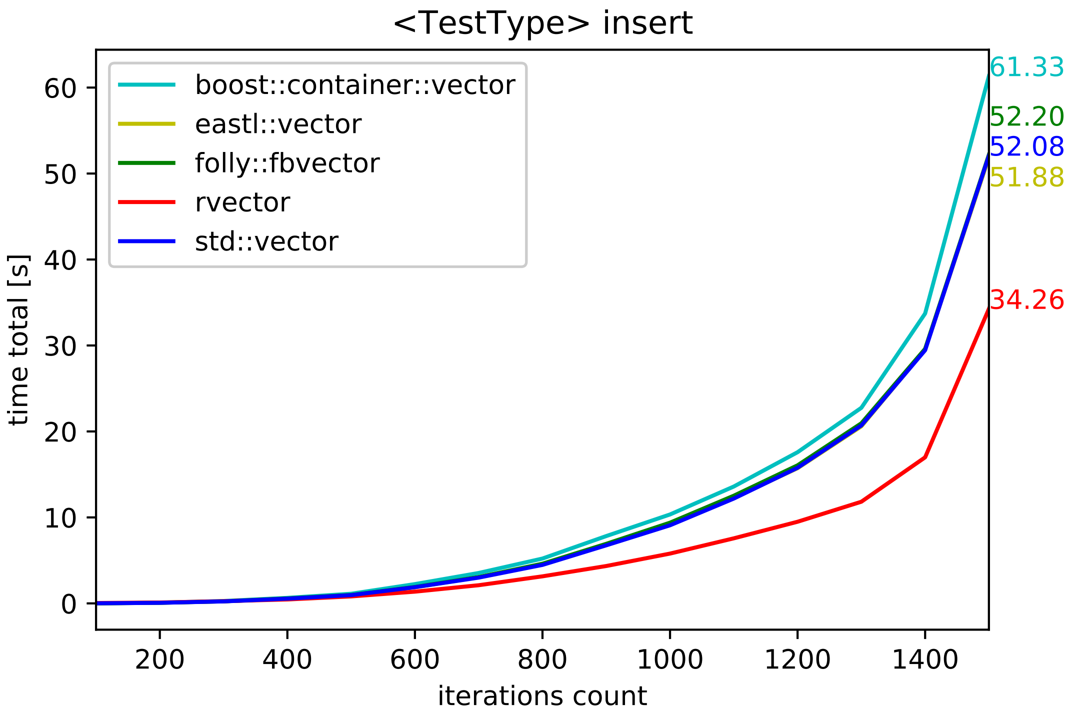

Figure 4.20: Insert action time for element type `TestType`.

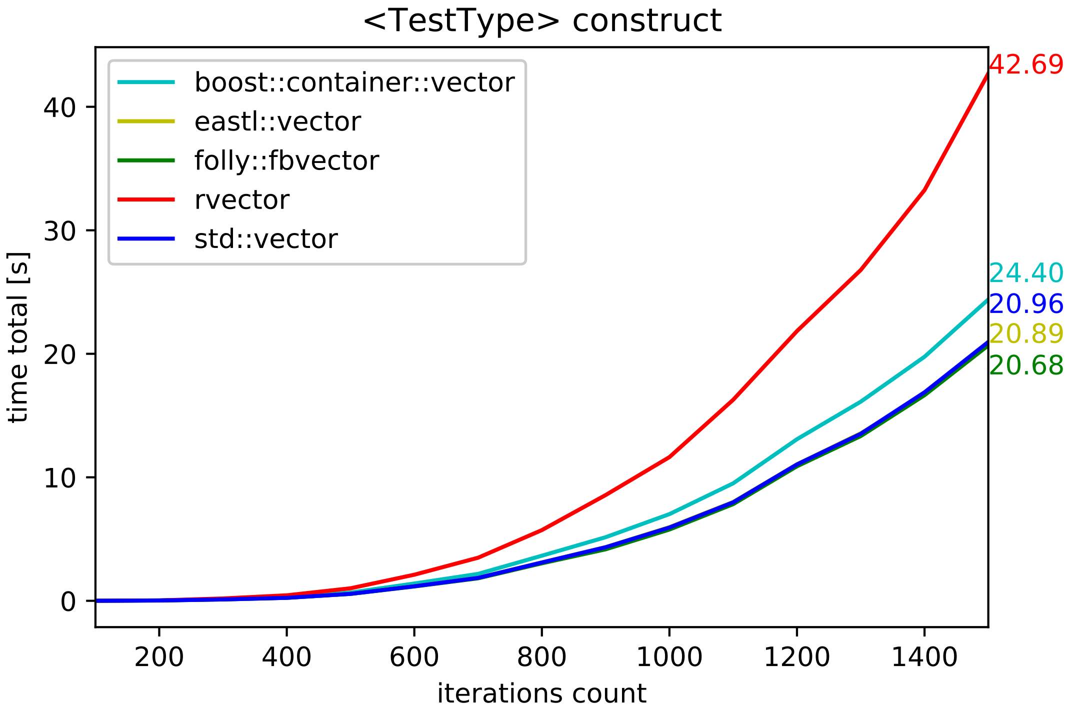

Figure 4.21: Construct action time for element type `TestType`.

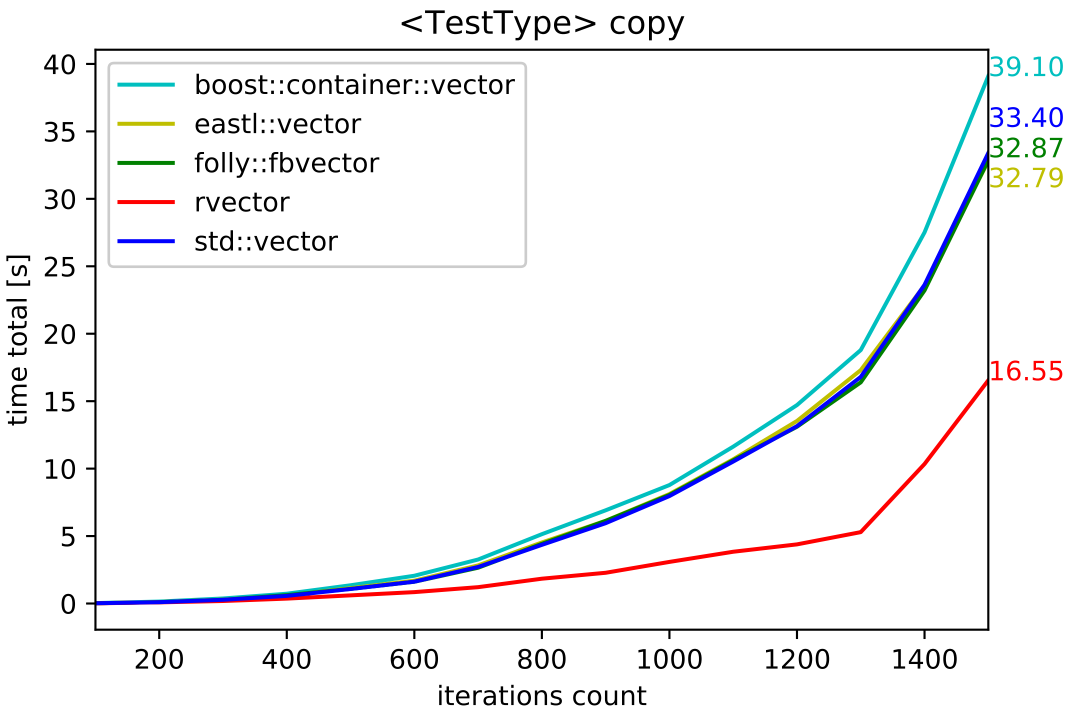

Figure 4.22: Copy action time for element type `TestType`.

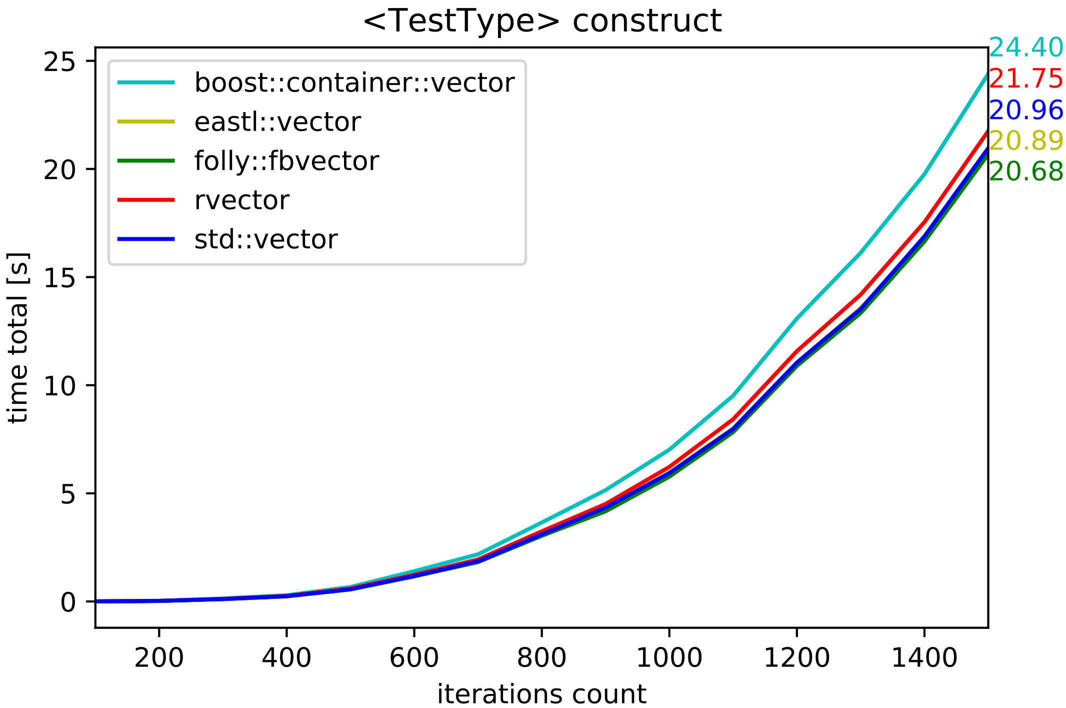

Figure 4.23: Construct action time for element type `TestType`. `Rvector` without `mmap`.

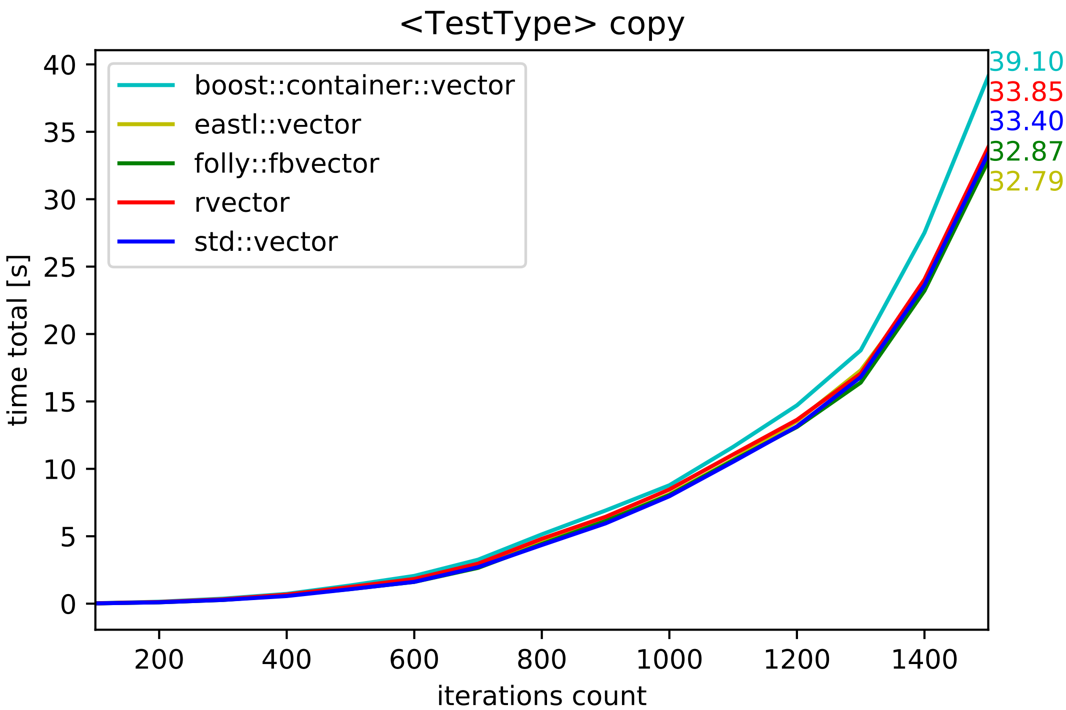

Figure 4.24: Copy action time for element type `TestType`. `Rvector` without `mmap`.

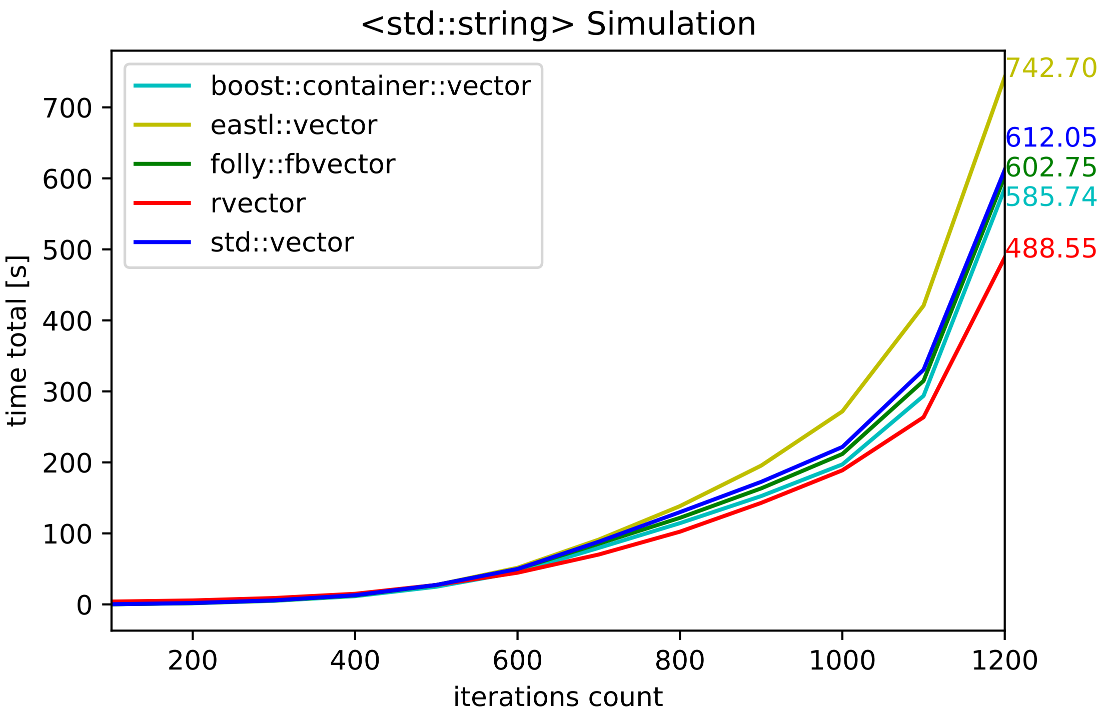

Figure 4.25: Total simulation time for element type `std::string` that allocates memory.

4.7 Element types: std::string, int, std::array

In this benchmark, three different element types have been tested at the same moment in a single tuple. It makes it the most time and memory consuming of all the previous benchmarks presented. A single simulation in the peak moment had a working set of around 11GB. As it is significantly more than the available physical memory (8GB), the simulation shows the efficiency of vectors during throttling.

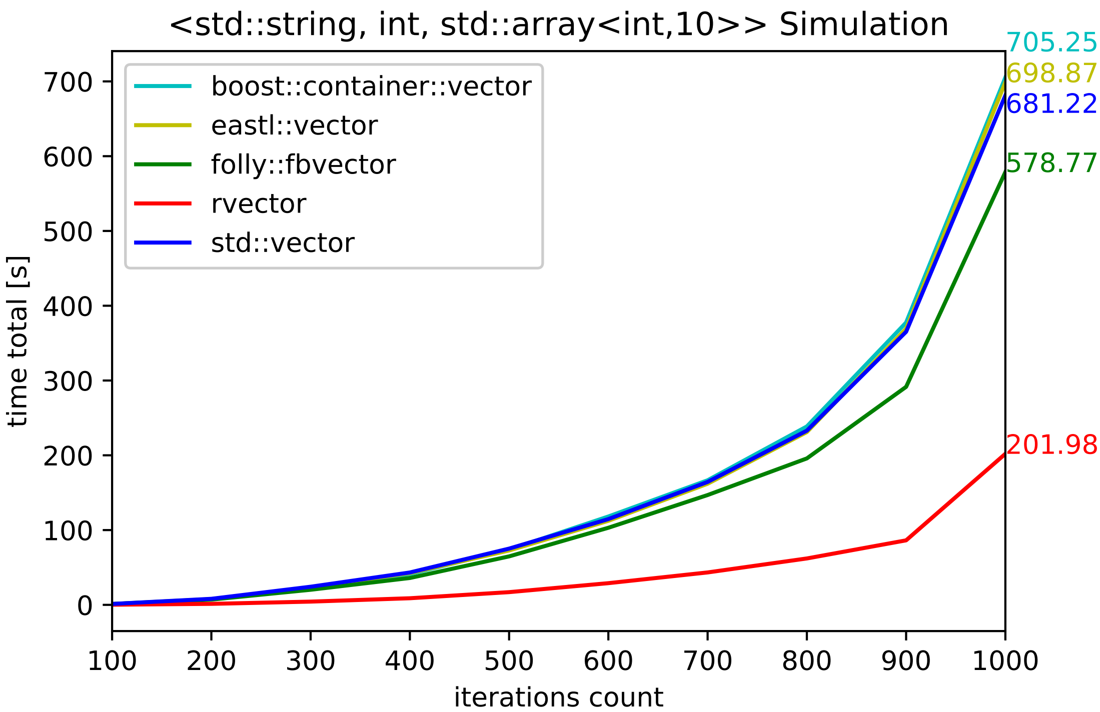

Figure 4.26: Total simulation time for element types `std::string`, `int`, `std::array`.

As it can be seen in Figure 4.26, `rvector` gets a significant advantage over the other vector implementations. Every action except the `construct` is done faster with `rvector`, which can be seen separately on Figures 4.27–4.32. Notably, even such actions as `erase` and `pop_back`, which do not induce reallocation, took less time to execute with the use of `rvector`. It is probably due to `rvector` being more cache/memory friendly. In contrast with the other vectors, when `rvector` reallocates, it often does not wipe a large part of cache as it does not copy the memory. This later causes faster execution of other operations, even if they do not involve reallocation.

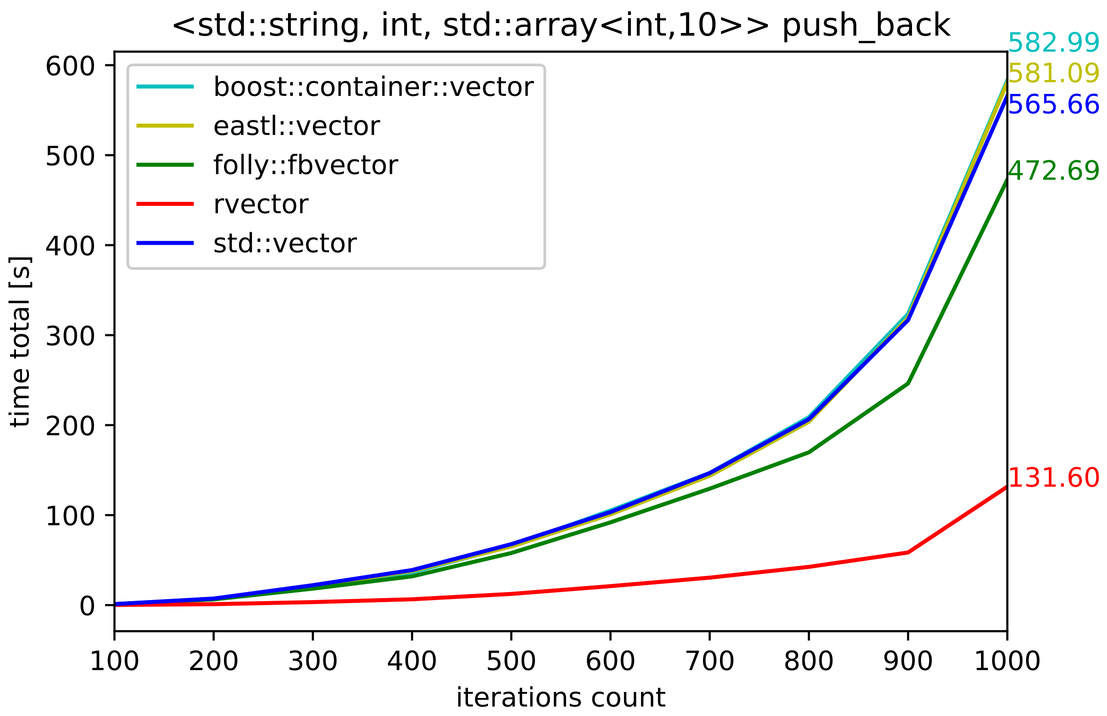

Figure 4.27: Push_back action time for element types `std::string`, `int`, `std::array`.

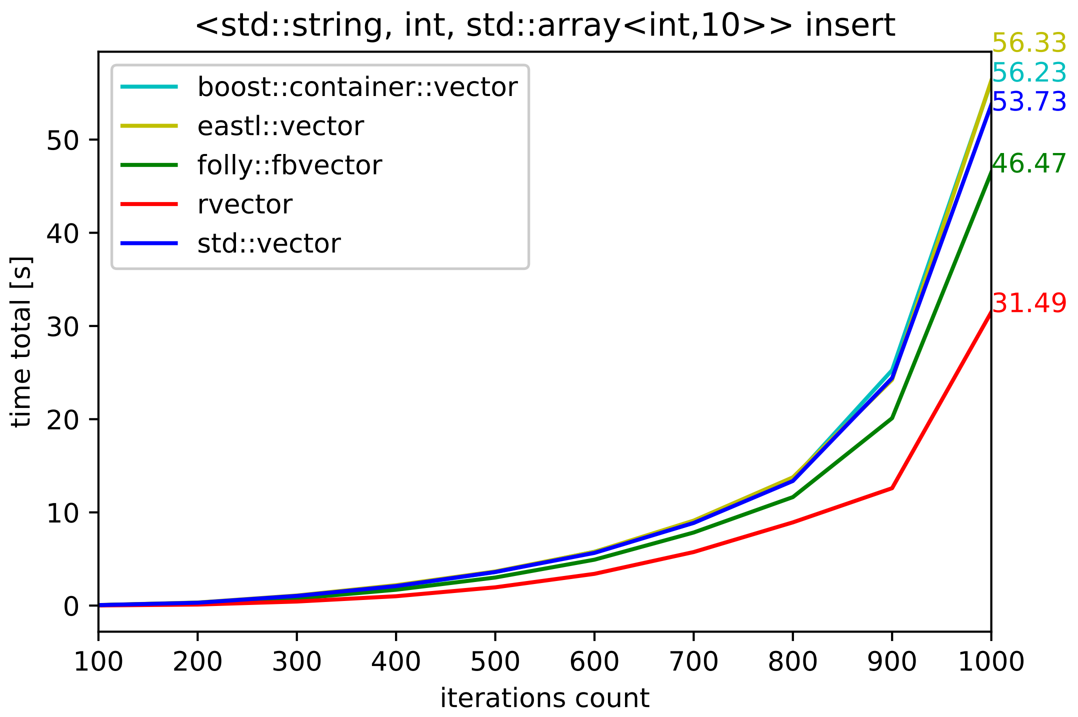

Figure 4.28: Insert action time for element types `std::string`, `int`, `std::array`.

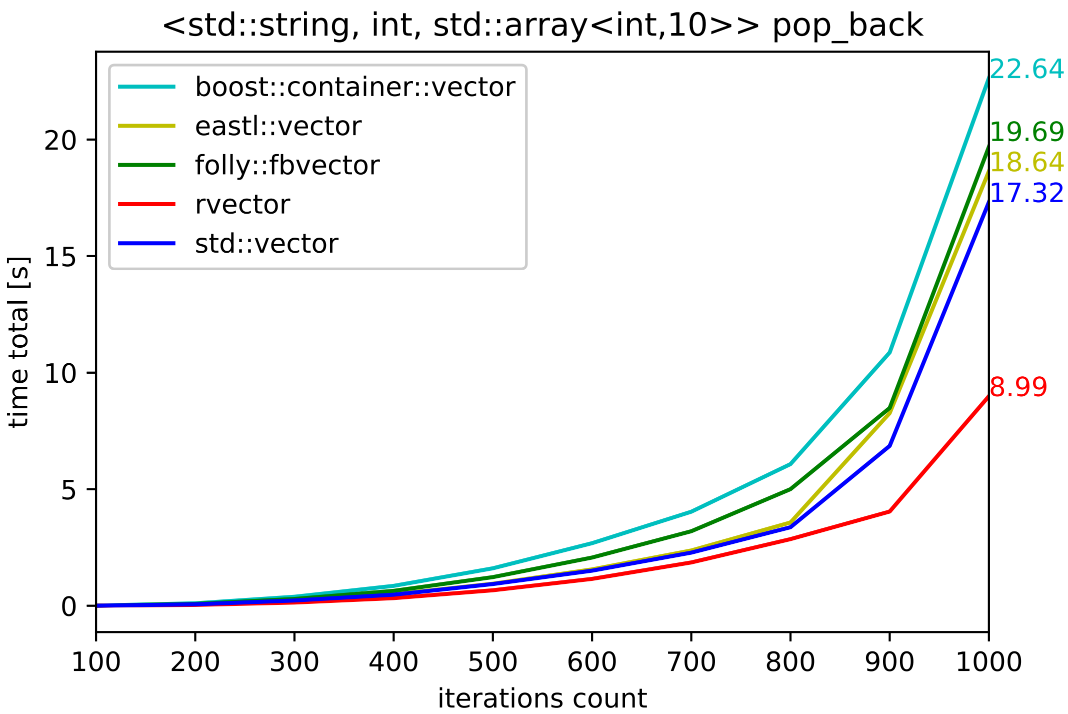

Figure 4.29: Pop_back action time for element types `std::string`, `int`, `std::array`.

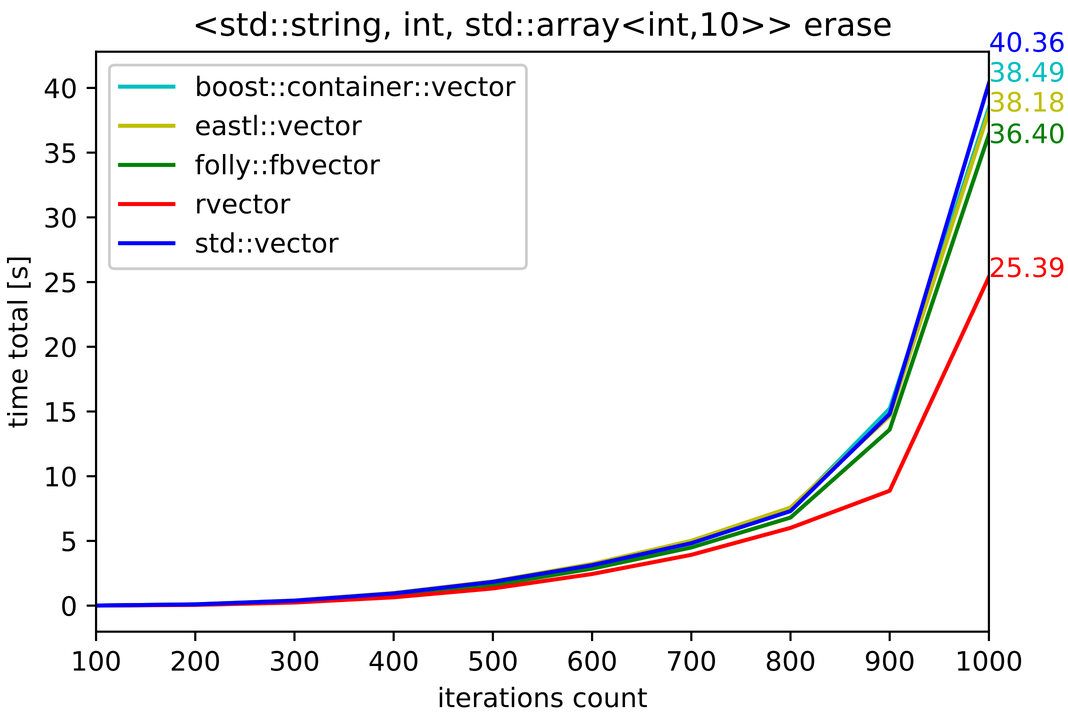

Figure 4.30: Erase action time for element types `std::string`, `int`, `std::array`.

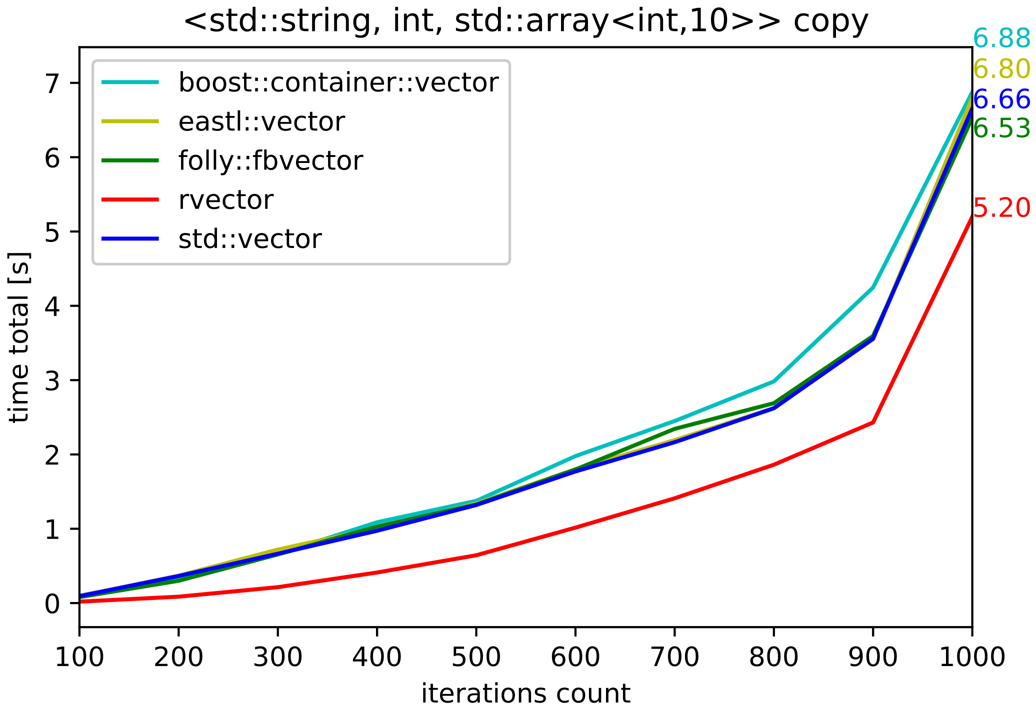

Figure 4.31: Copy action time for element types `std::string`, `int`, `std::array`.

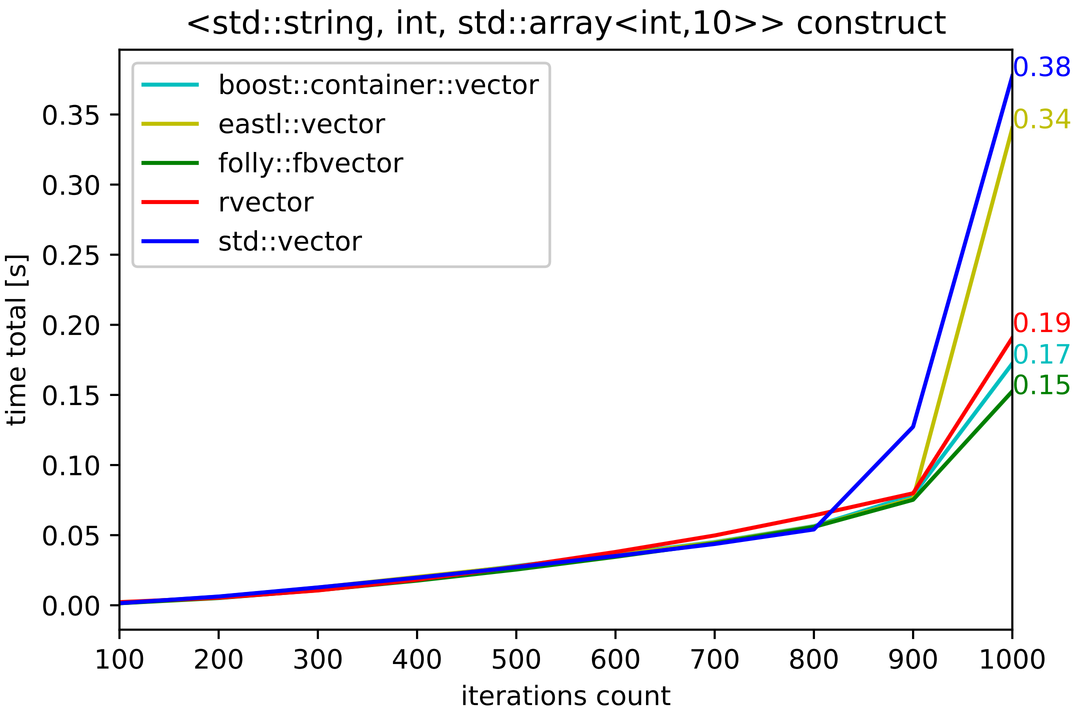

Figure 4.32: Construct action time for element types `std::string`, `int`, `std::array`.

4.8 Project test

Finally, we tested the vectors in a real complex project. For this, an implementation of an algorithm for computing the length of the *shortest reset words* was used [14]. The program was run for random binary automata with a different number of states. The solved problem is computationally hard, and both time and space complexity of the algorithm grows exponentially in the number of states in an average case. In the implementation, there are vectors for different element types, including custom ones. The most heavily used vectors are for the `unsigned int` element type, which are used to store large dynamic data structures.

In the tests, all the vectors occurring in the code were simply replaced with a particular tested vector implementation. The results are shown in Fig. 4.33.

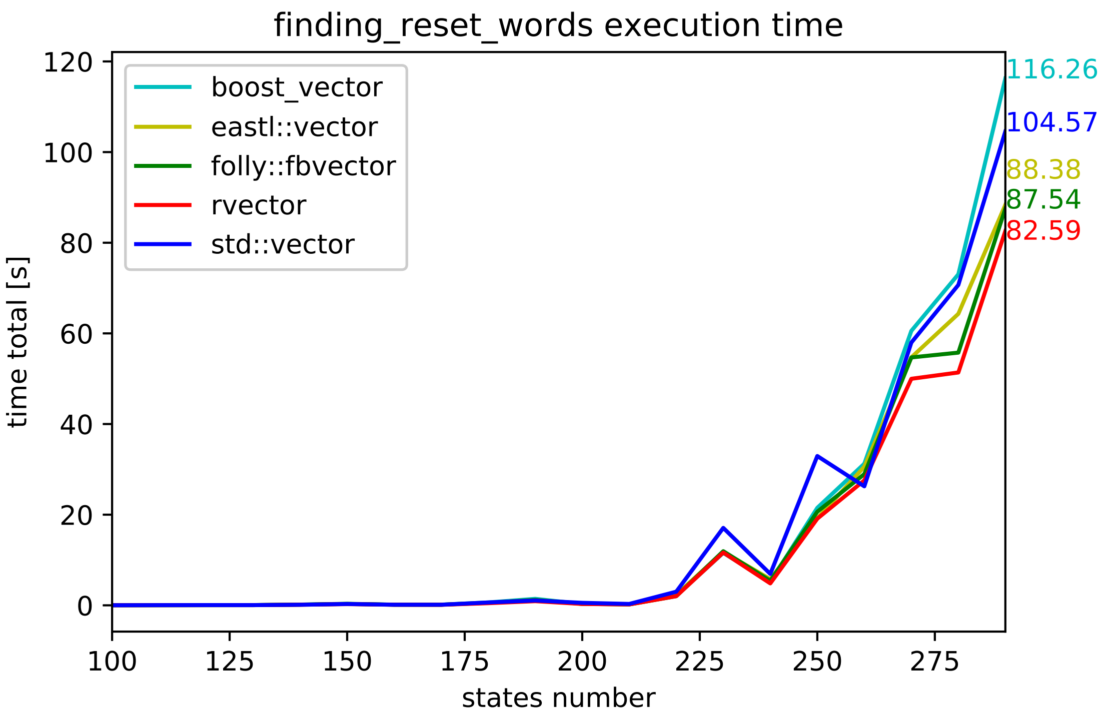

Figure 4.33: The efficiency of the vectors in computing the length of the shortest reset words.

For the largest input, `rvector` is about 21% faster than `std::vector` and about 6% faster than `folly::fbvector`. Thus, in a real project, the speed up gained just from optimizing vectors can be significant.

Chapter 5

Conclusions

From the results on small and large *Plain Old Data* (*POD*) types, one can conclude that the gain from using `mremap` as a reallocation is more significant with a greater memory block to be reallocated. It is also the case for non-trivial types, yet the advantage might be lesser. This is because an in-place expansion may not occur often, since the process heap is fragmented due to many allocations performed by a non-trivial type.

Summarizing, `rvector` provides a significant speed up over the existing vector implementations in most typical cases. The detailed features and advantages of `rvector` are as follows:

- It performs a much faster reallocation when the element type is POD. This is very helpful for heavy usage of `push_back` or `insert`.
- For non-POD types, `mremap` reallocation in place still gives an advantage over the other implementations, yet it is influenced by memory fragmentation.
- `Rvector` behaves much better during throttling (i.e. when the memory resources are running low).
- Only the standard allocator (`malloc` for small sizes) is used and the memory is taken directly from the system. This means that there is no extra memory reserved, in contrast with usual custom memory allocators.
- There are no dependencies, which allows easy integration with any project as a replacement of the standard implementation.
- Compared to the others, it has a simple implementation, which yet uses features of C++17.
- It is compliant with C++17 `std::vector` interface, including deduction guides for constructors.

However, it also has a disadvantage:

- It works only in Linux-based systems because it relies on their memory management interface.

Possible further improvements are as follows:

- Make `rvector` give C++17 standard vector exceptions guarantees.
- Make `rvector` cross-platform, utilizing OS-specific memory management interface.
- Add parameters customizing the vector, such as `growth_factor` or `map_threshold` parameter, which sets the maximal capacity for allocations with `malloc`.
- Add a template parameter indicating whether the element type should be considered as trivial, overriding the automatic policies.

Bibliography

[1] mremap syscall manual http://man7.org/linux/man-pages/man2/mremap.2.html

[2] std::vector libstdc++ implementation https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/include/bits/stl_vector.h

[3] folly::fbvector implementation https://github.com/facebook/folly/blob/master/folly/FBVector.h

[4] folly::fbvector description https://github.com/facebook/folly/blob/master/folly/docs/FBVector.md

[5] boost::container::vector documentation https://www.boost.org/doc/libs/1_68_0/doc/html/boost/container/vector.html

[6] https://www.boost.org/doc/libs/1_68_0/doc/html/container/cpp_conformance.html#container.cpp_conformance.vector_exception_guarantees

[7] dlmalloc description ftp://g.oswego.edu/pub/misc/malloc.c

[8] eastl::vector implementation https://github.com/electronicarts/EASTL/blob/master/include/EASTL/vector.h

[9] EASTL FAQ https://rawgit.com/electronicarts/EASTL/master/doc/EASTL%20FAQ.html

[10] rvector implementation https://github.com/Bixkog/rvector/blob/master/rvector.h

[11] Itanium ABI for exception handling https://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html

[12] rvector tests https://github.com/Bixkog/rvector/blob/master/test.cpp

[13] trivial type definition http://www.cplusplus.com/reference/type_traits/is_trivial/

[14] A. Kisielewicz, J. Kowalski, M. Szykuła. Computing the shortest reset words of synchronizing automata. *Journal of Combinatorial Optimization* 29(1):88–124, 2015.

List of Figures

Listings