Benchmark Results Report

This report summarizes the benchmark analysis of VectorTree
by comparing its performance with:

1. std::vector
2. a simple persistent vector

VectorTree is created to obtain a persistent version of std::vector for multithreaded usage. In other words, the target of VectorTree is to achieve the performance of std::vector while keeping the persistency to deal with the concurrency issues. Hence, the 1st container to compare with is std::vector. The 2nd one is a very simple definition for a persistent vector. The flowchart of any operation on the simple persistent vector is as follows:

1. create a copy of the original vector,
2. apply the request on the copy,
3. return the modified copy.

Four fundamental operations are inspected in this benchmark:

1. emplace_back
2. pop_back
3. pop_front
4. traversal

The 3rd operation is actually not a property of the vector data structure such that std::vector interface does not include it. It requires one-step move operation on all elements which is linear, O(N). Its the same, even more problematic, in case of a tree of vectors. VectorTree solves this problem using swap-and-pop idiom while loosing the order of elements. Hence, the pop_front graphs showing a better performance for VectorTree, actually shows the performance support coming from the swap-and-pop idiom. In other words, consider the graphs for the pop_front operation as the comparison between:

- pop_front: move<T> with O(N)
- swap-and-pop: front() = back(); pop_back(): copy<T> with O(2*BufferSize)

Summary of the test parameters:

1. Inspected operations: emplace_back, pop_back, pop_front and traversal
2. Inspected objects: Two types are inspected: type_small and type_large
3. Original container size:
   BUFFER_SIZE_1 (32)
   BUFFER_SIZE_2 (1024)
   BUFFER_SIZE_3 (32768)

The two object types are defined simply as follows:

1. type_small: an object with one field of int
2. type_large: an object with an array of 256 elements of int

The number of the combinations of the test parameters are:

1. # of the inspected operations: 4
2. # of the inspected objects: 2
3. # of the inspected original container sizes: 3

Hence, the combination of the test parameters yield:

Test count = 4 * 2 * 3 = 24

Note that the tests are performed with a VectorTree of buffer size of 32.

The tests follow a simple algorithm:

1. Initialize a container with a predefined size, N
2. Aplly the operation iteratively N times on this original container
3. Measure timing

A templated wrapper class is created to simulate a uniform interface for the inspected four operations which helps to simplify the google benchmark macros. A base template (VectorTree) and two specializations (std::vector and persistent_vector) simulate the three containers to be compared.

DEFINE_BENCHMARK macro defines a shortcut to the google benchmark macros.

The tests are quite simple and the corresponding graphs are self-explanatory.
Hence, i will not go through the results in detail.
In summary:

- For containers of small size, although VectorTree is the worst one,
  it performs efficiently (~600ms):
  -> See graphs with BUFFER_SIZE_1
- For containers of large size, VectorTree performs way better than
  the simple persistent vector:
  -> See graphs with BUFFER_SIZE_3
- VectorTree approaches to std::vector
  in case of the three fundamental operations:
  -> See graphs with BUFFER_SIZE_3 and especially with type_large
- The swap-and-pop idiom provides
  an efficient solution to the pop_front operation:
  -> See graphs with pop_front

- The most important problem of VectorTree is the traversal performance.
  A traversal with a VectorTree of size=32768 takes around ~0.01 seconds
  which may be unacceptable in some cases:
    -> See Figures 21 and 24

  The iterator must step to the next leaf node
  when the end of the current leaf node buffer is reached
  which requires a DFS algorithm within the tree structure.
  The worst traversal performance is a result of the DFS algorithm
  abstracted/hidden by the path_to_leaf_node__current variable.
  Hence, the iterator of VectorTree must be improved.
  I will study two approaches to either remove or improve the DFS:
    1. Memoization (removes DFS):
       VectorTree stores a linear container of pointers
       to the leaf node buffers,
    2. Asynchronous approach (improves DFS):
       A thread determines the next leaf node buffer asynchronously.
  Both have pros and cons which must be studied in detail.
  For example the memoization adds more memory load:
    array of pointers of N/32 size where 32 is the buffer size.
  Asynhcronous approach, on the other hand,
  terminates the default copy capability of the iterator
  which reduces the copy construction performance.
  The increment operator (i.e. operator++(int)) and the derivatives
  becomes ineffective as a result.

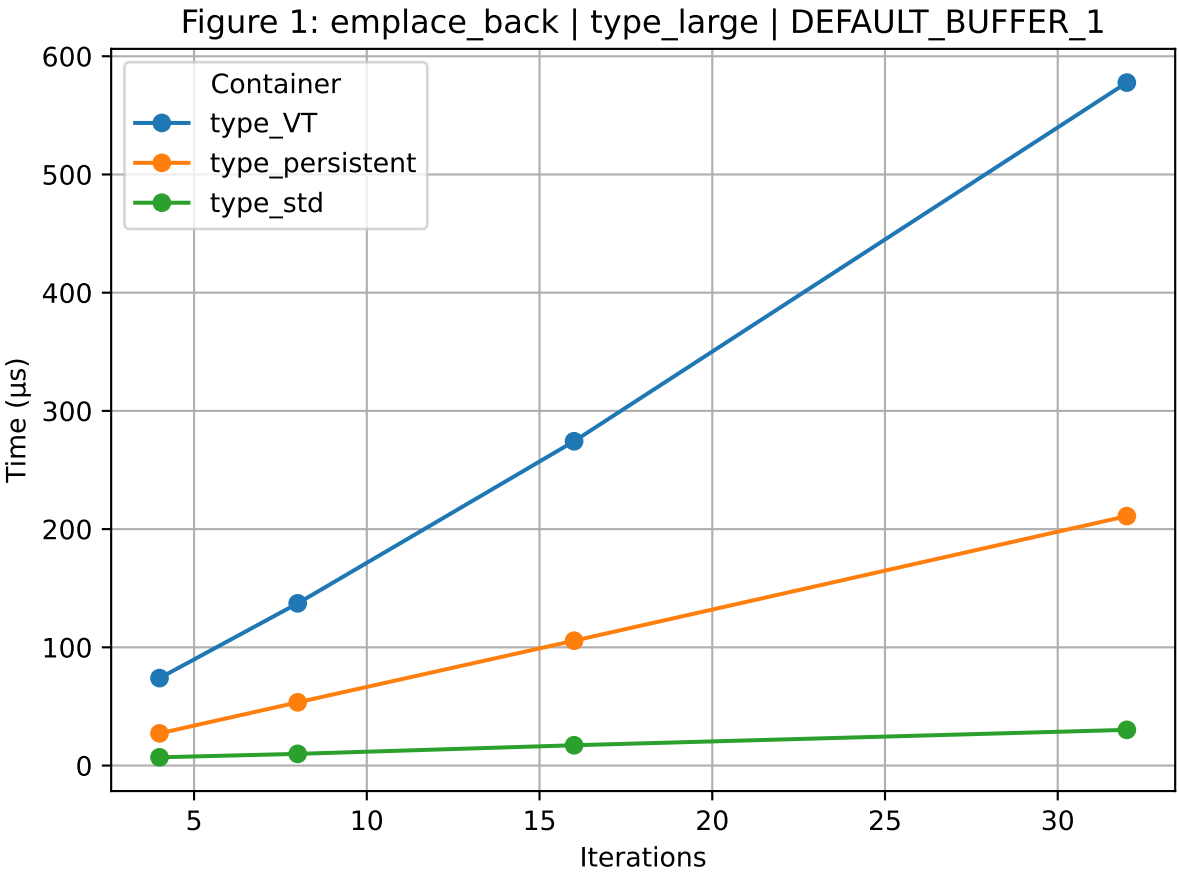
Figure 1: emplace_back | type_large | DEFAULT_BUFFER_1
Container
type_VT
type_persistent
type_std
Time (µs)
Iterations
0
100
200
300
400
500
600
5
10
15
20
25
30

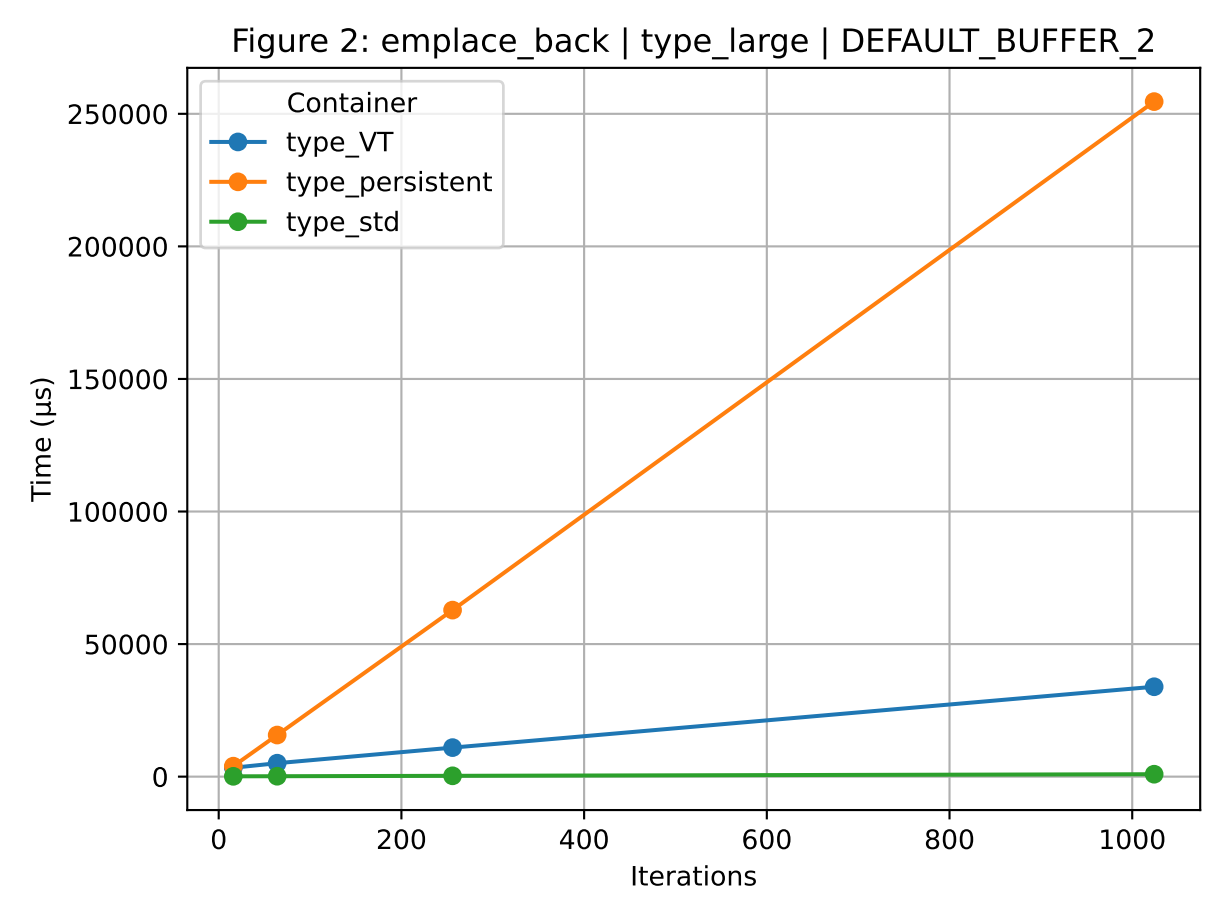
Figure 2: emplace_back | type_large | DEFAULT_BUFFER_2
Container
type_VT
type_persistent
type_std
Time (µs)
Iterations
0
50000
100000
150000
200000
250000
0
200
400
600
800
1000

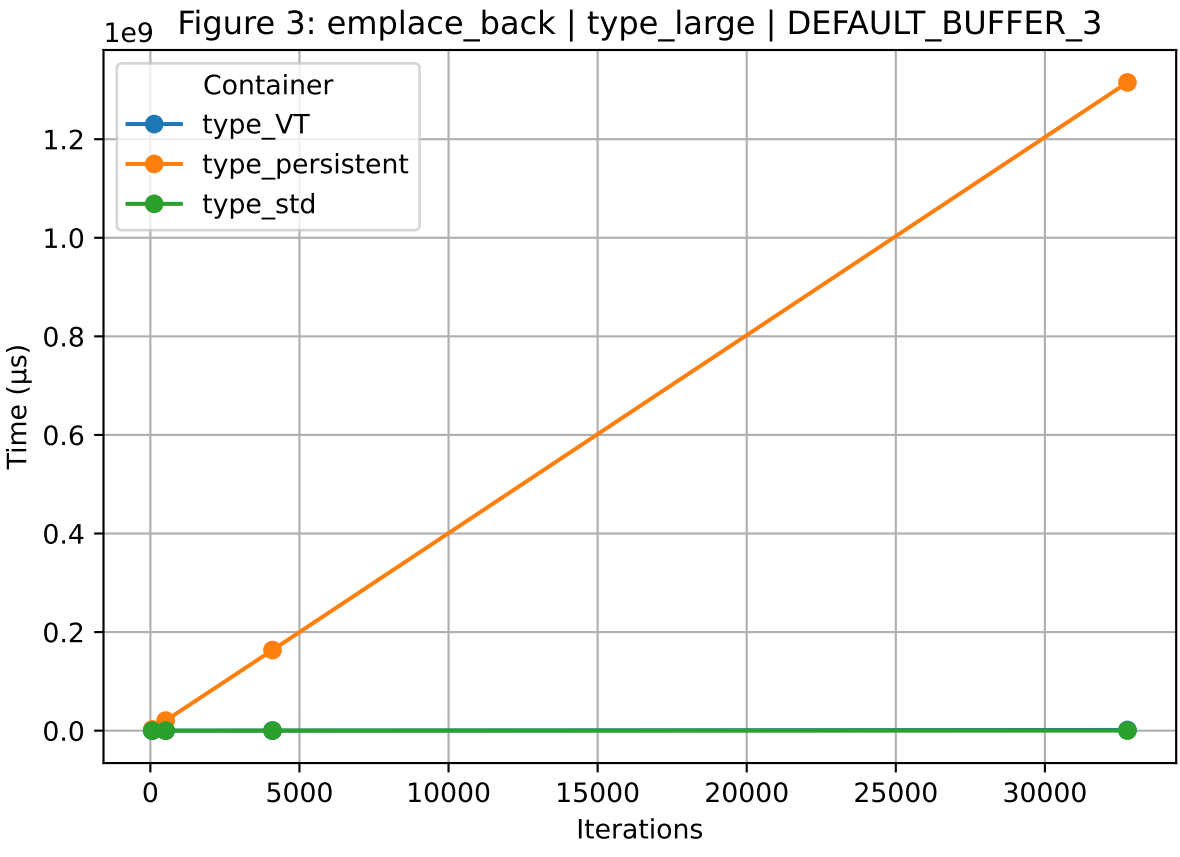
Figure 3: emplace_back | type_large | DEFAULT_BUFFER_3
1e9
Container
type_VT
type_persistent
type_std
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Time (µs)
0
5000
10000
15000
20000
25000
30000
Iterations

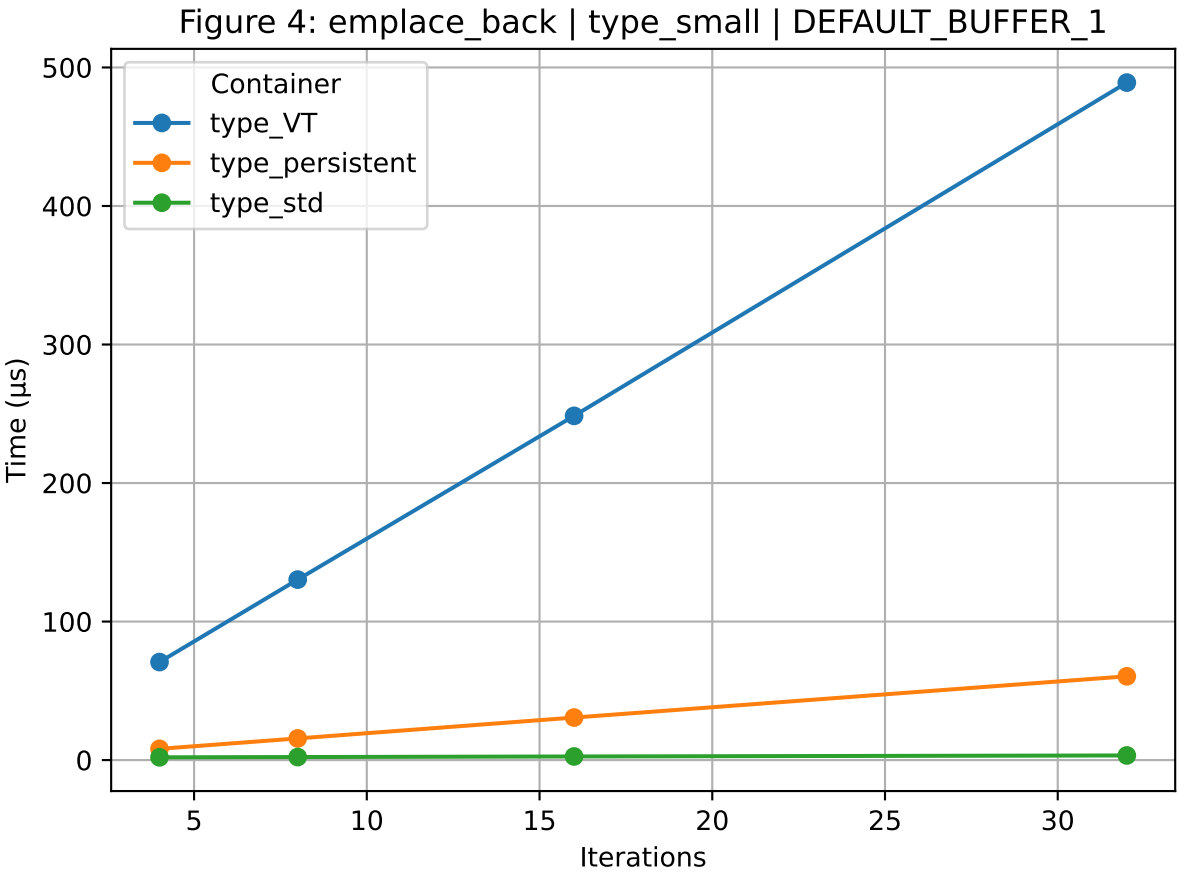
Figure 4: emplace_back | type_small | DEFAULT_BUFFER_1
Container
type_VT
type_persistent
type_std
Time (µs)
Iterations
0
100
200
300
400
500
5
10
15
20
25
30

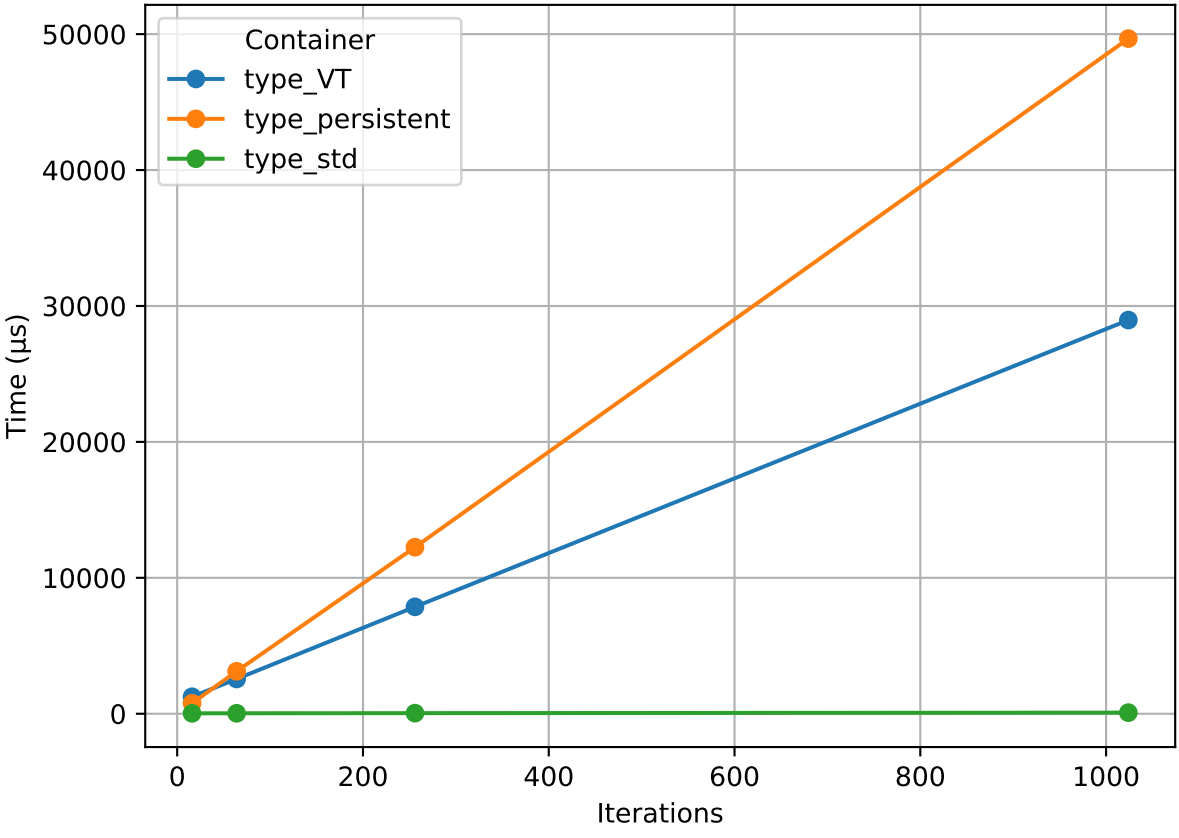
Figure 5: emplace_back | type_small | DEFAULT_BUFFER_2
Container
type_VT
type_persistent
type_std
50000
40000
30000
20000
10000
0
Time (µs)
0
200
400
600
800
1000
Iterations

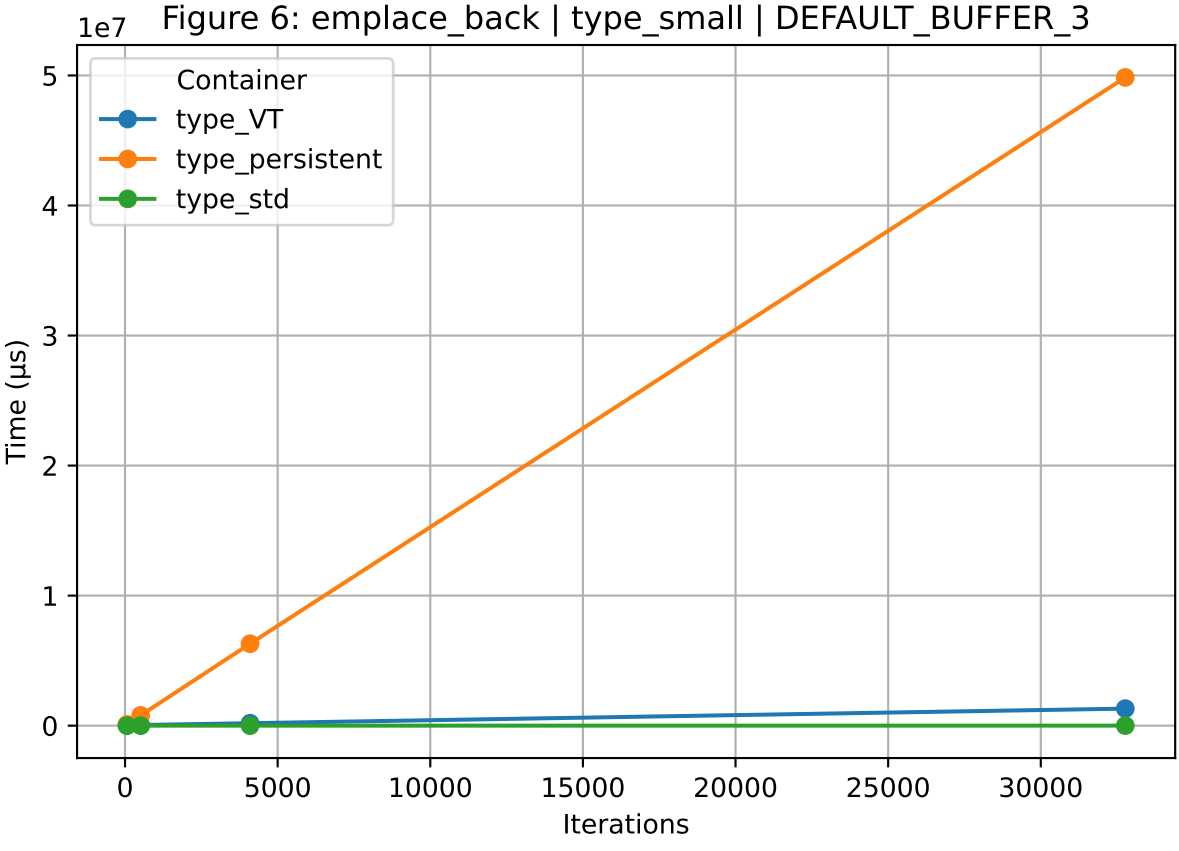
Figure 6: emplace_back | type_small | DEFAULT_BUFFER_3
1e7
Container
type_VT
type_persistent
type_std
Time (µs)
Iterations
0
1
2
3
4
5
0
5000
10000
15000
20000
25000
30000

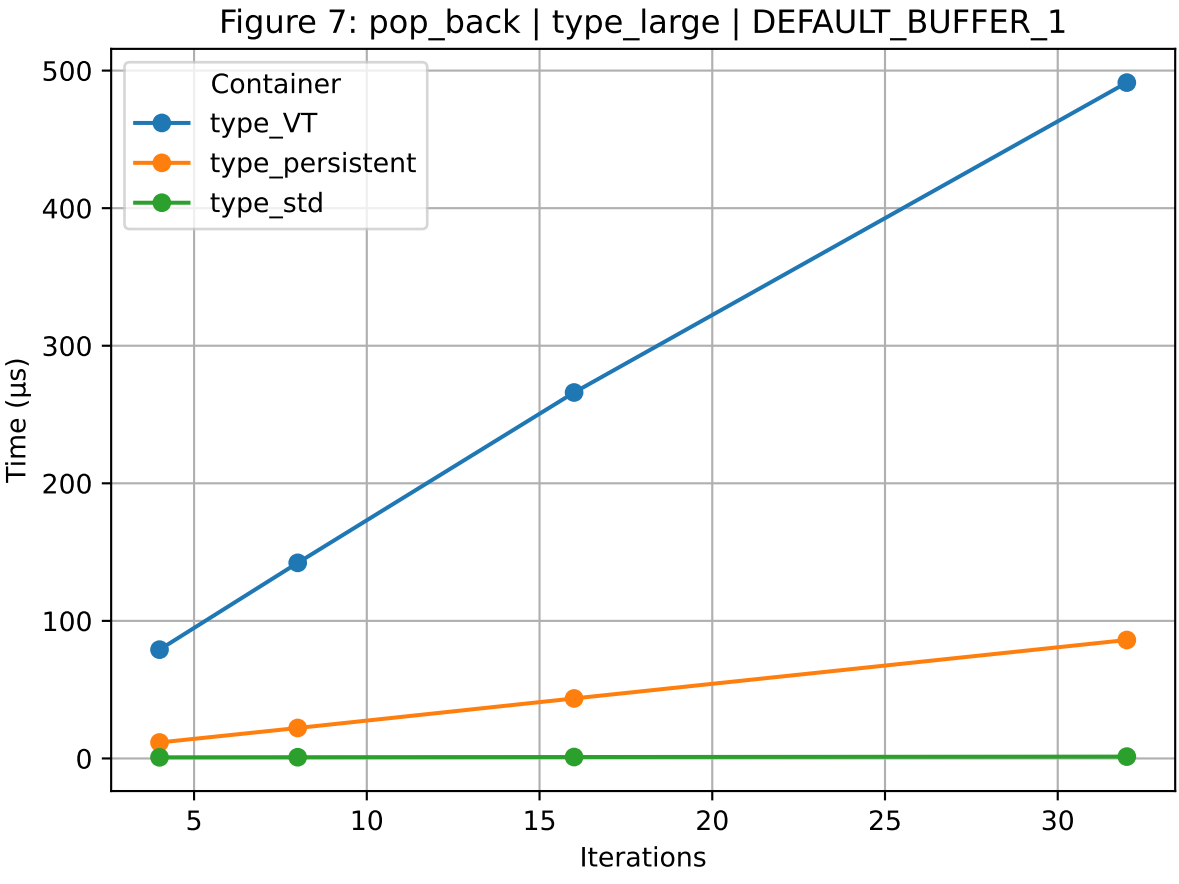
Figure 7: pop_back | type_large | DEFAULT_BUFFER_1
Container
type_VT
type_persistent
type_std
Time (µs)
0
100
200
300
400
500
5
10
15
20
25
30
Iterations

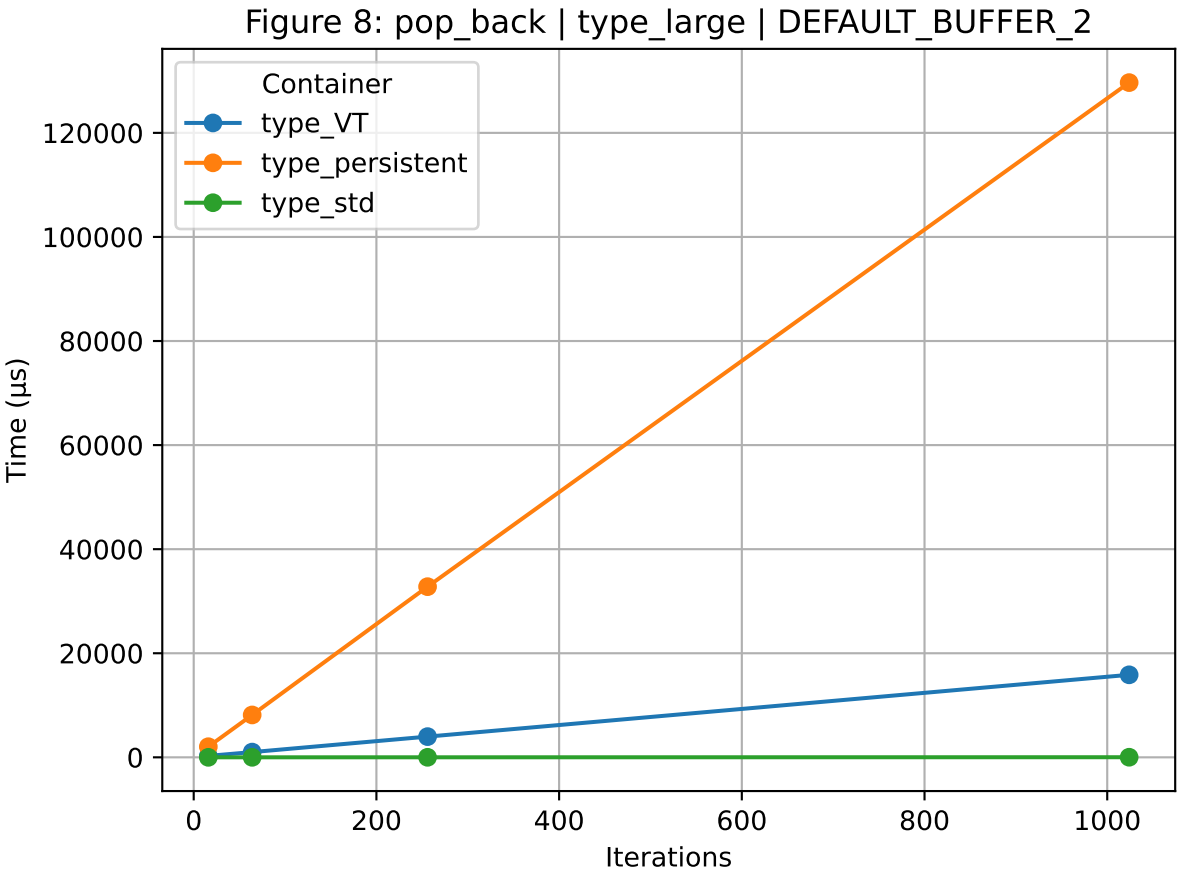
Figure 8: pop_back | type_large | DEFAULT_BUFFER_2
Container
type_VT
type_persistent
type_std
120000
100000
80000
60000
40000
20000
0
Time (µs)
0
200
400
600
800
1000
Iterations

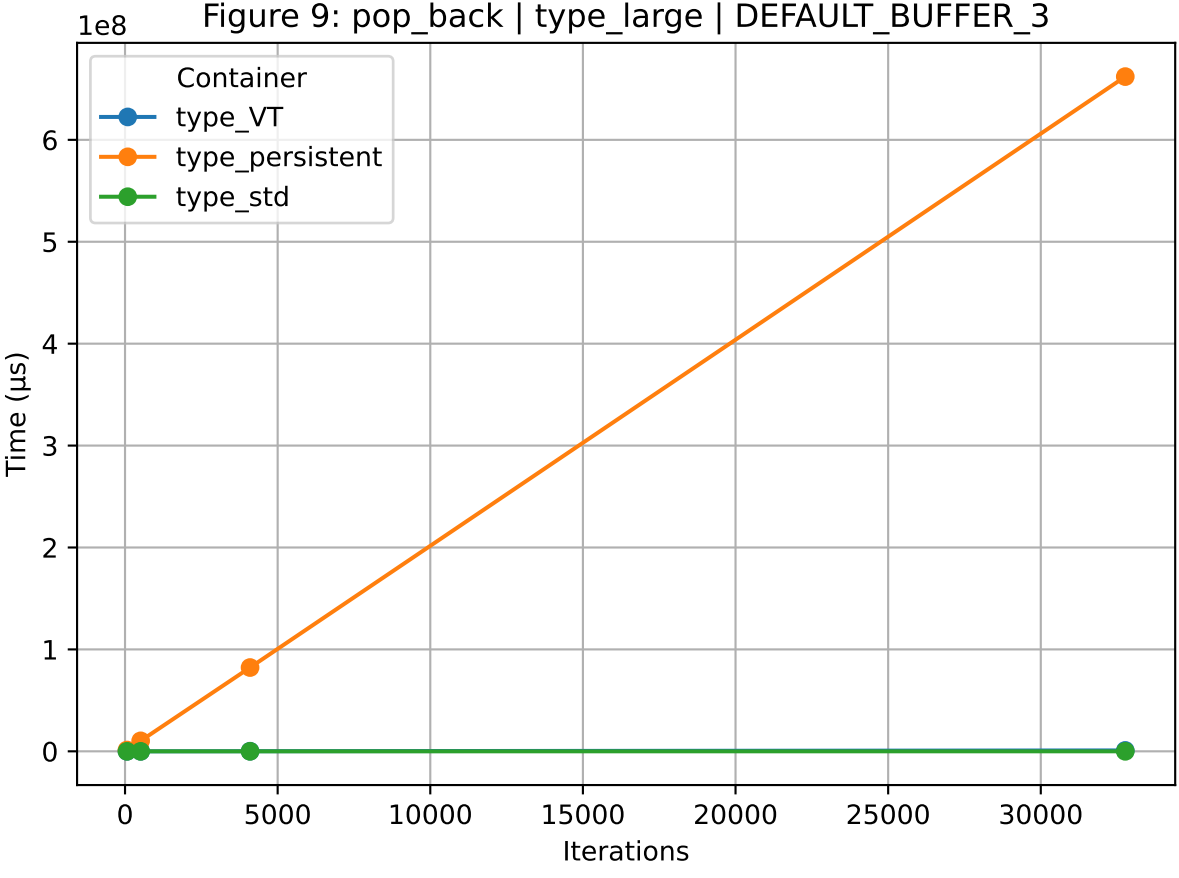
Figure 9: pop_back | type_large | DEFAULT_BUFFER_3
1e8
Container
type_VT
type_persistent
type_std
Time (µs)
Iterations
0
1
2
3
4
5
6
0
5000
10000
15000
20000
25000
30000

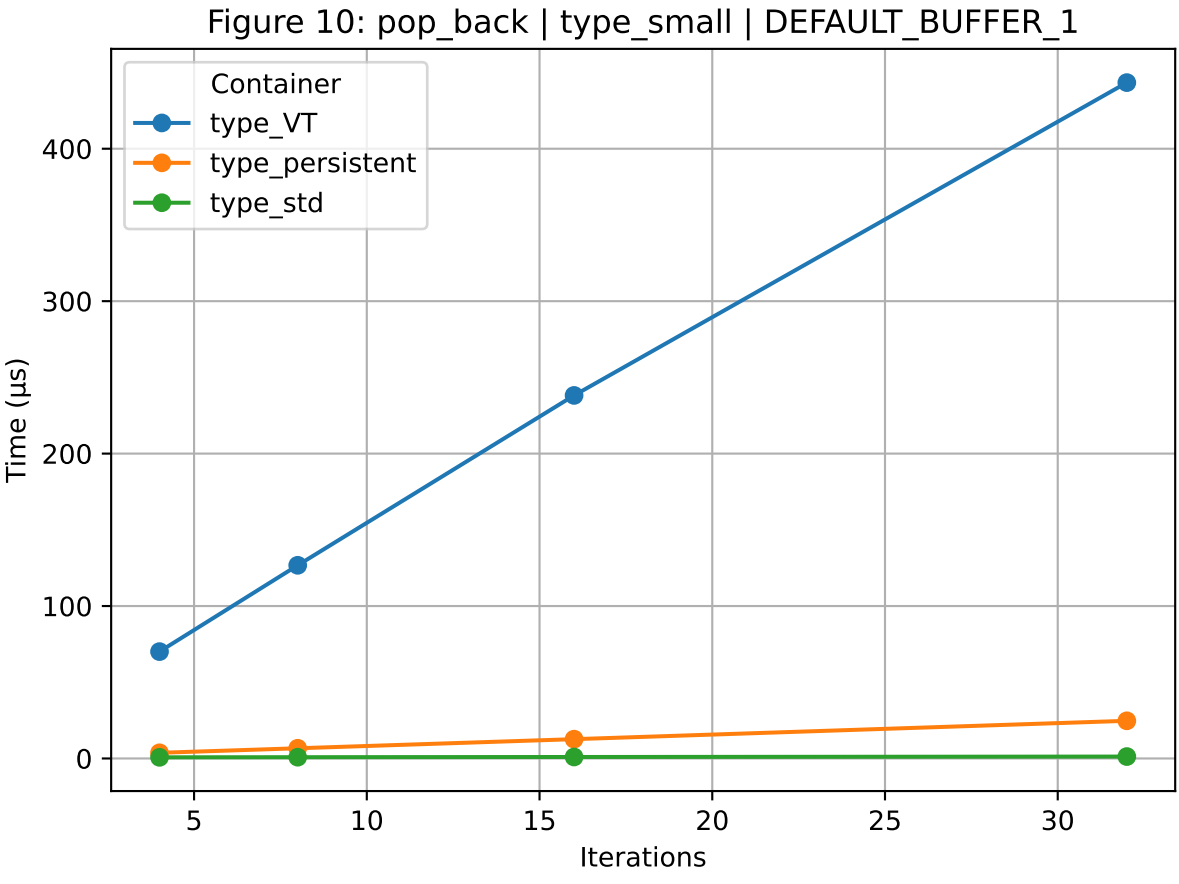
Figure 10: pop_back | type_small | DEFAULT_BUFFER_1
Container
type_VT
type_persistent
type_std
Time (µs)
0
100
200
300
400
5
10
15
20
25
30
Iterations

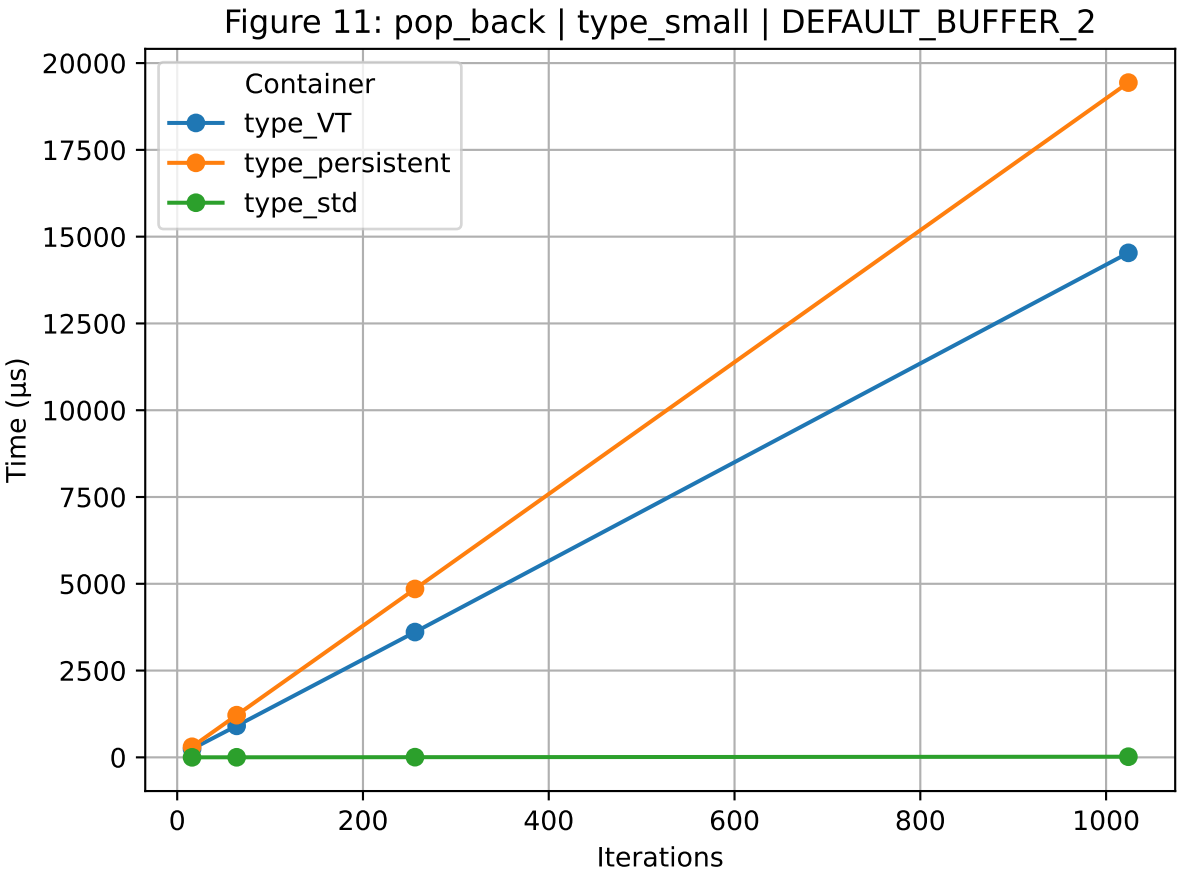
Figure 11: pop_back | type_small | DEFAULT_BUFFER_2
Container
type_VT
type_persistent
type_std
Time (µs)
Iterations
20000
17500
15000
12500
10000
7500
5000
2500
0
0
200
400
600
800
1000

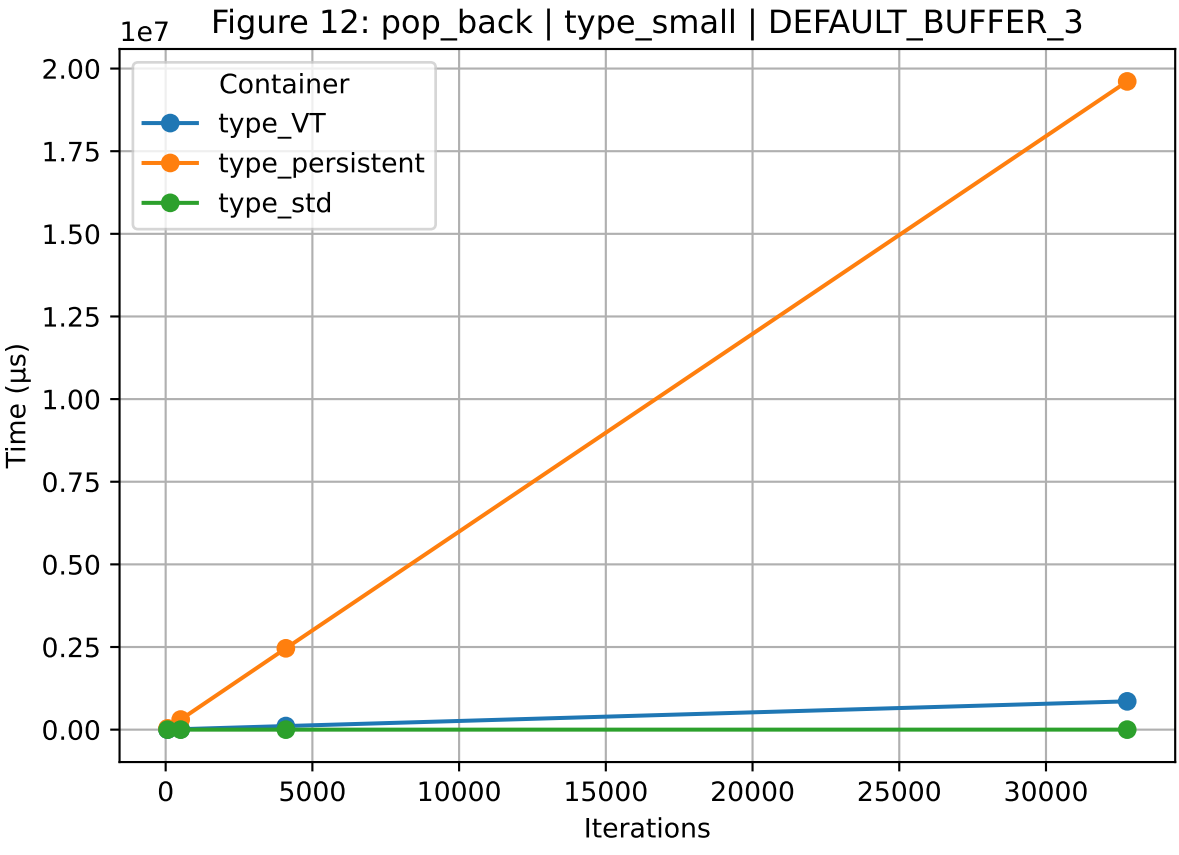
Figure 12: pop_back | type_small | DEFAULT_BUFFER_3
1e7
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
Time (µs)
0
5000
10000
15000
20000
25000
30000
Iterations
Container
type_VT
type_persistent
type_std

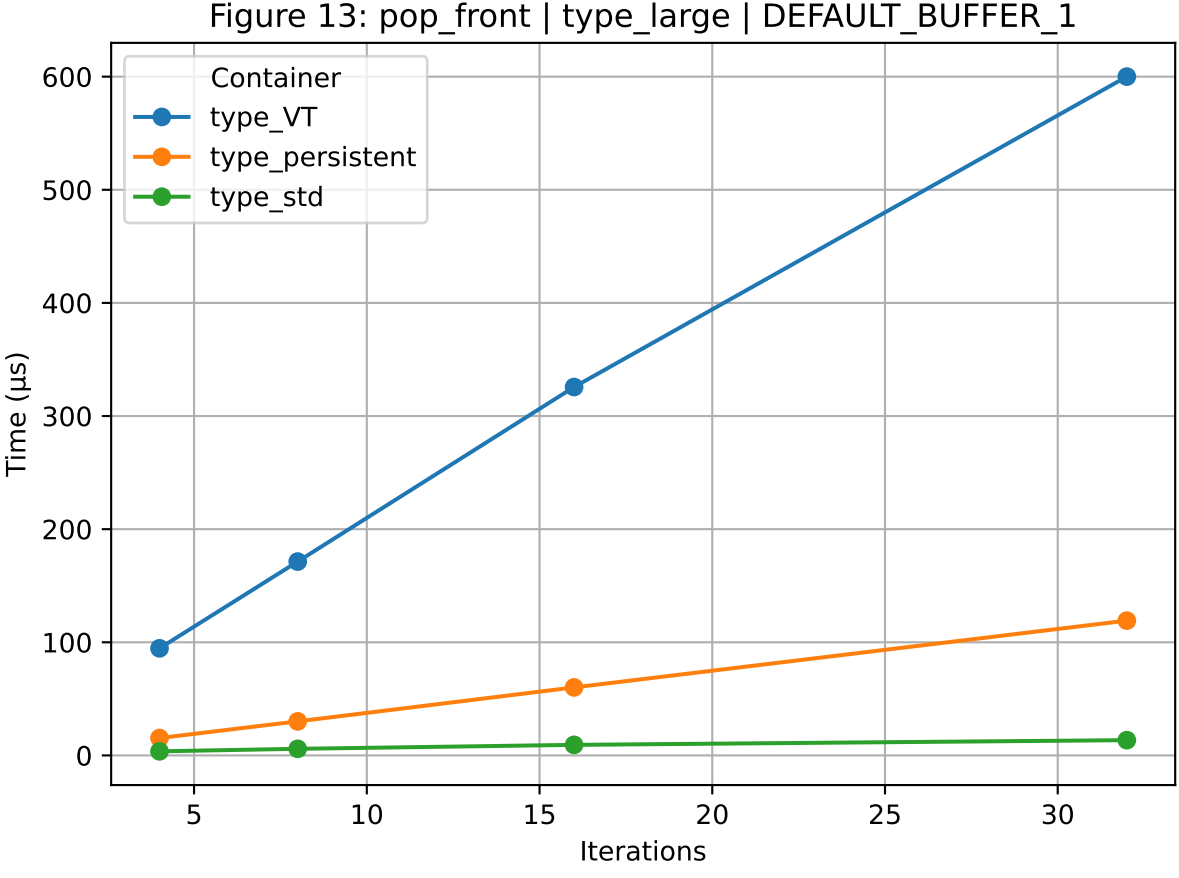
Figure 13: pop_front | type_large | DEFAULT_BUFFER_1
Container
type_VT
type_persistent
type_std
Time (µs)
Iterations
0
100
200
300
400
500
600
5
10
15
20
25
30

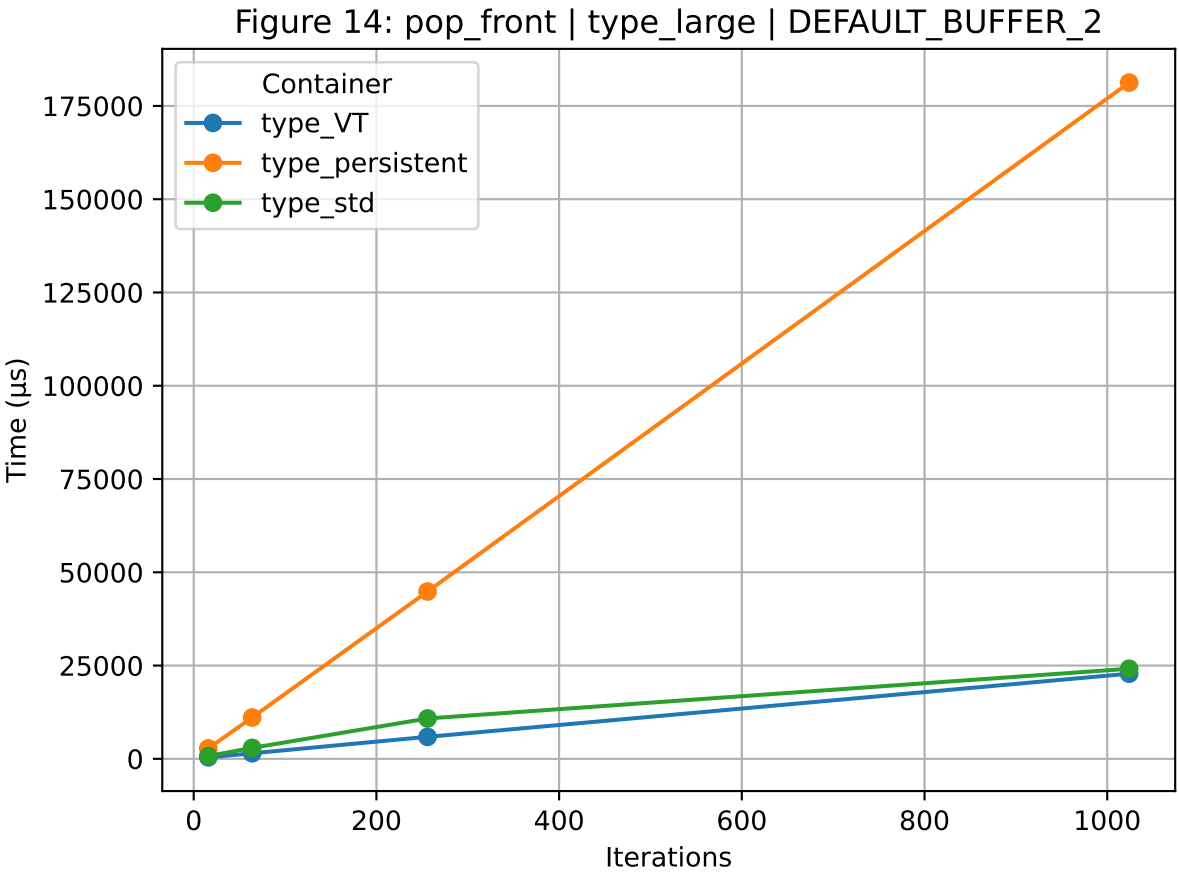
Figure 14: pop_front | type_large | DEFAULT_BUFFER_2
Container
type_VT
type_persistent
type_std
175000
150000
125000
100000
75000
50000
25000
0
Time (µs)
0
200
400
600
800
1000
Iterations

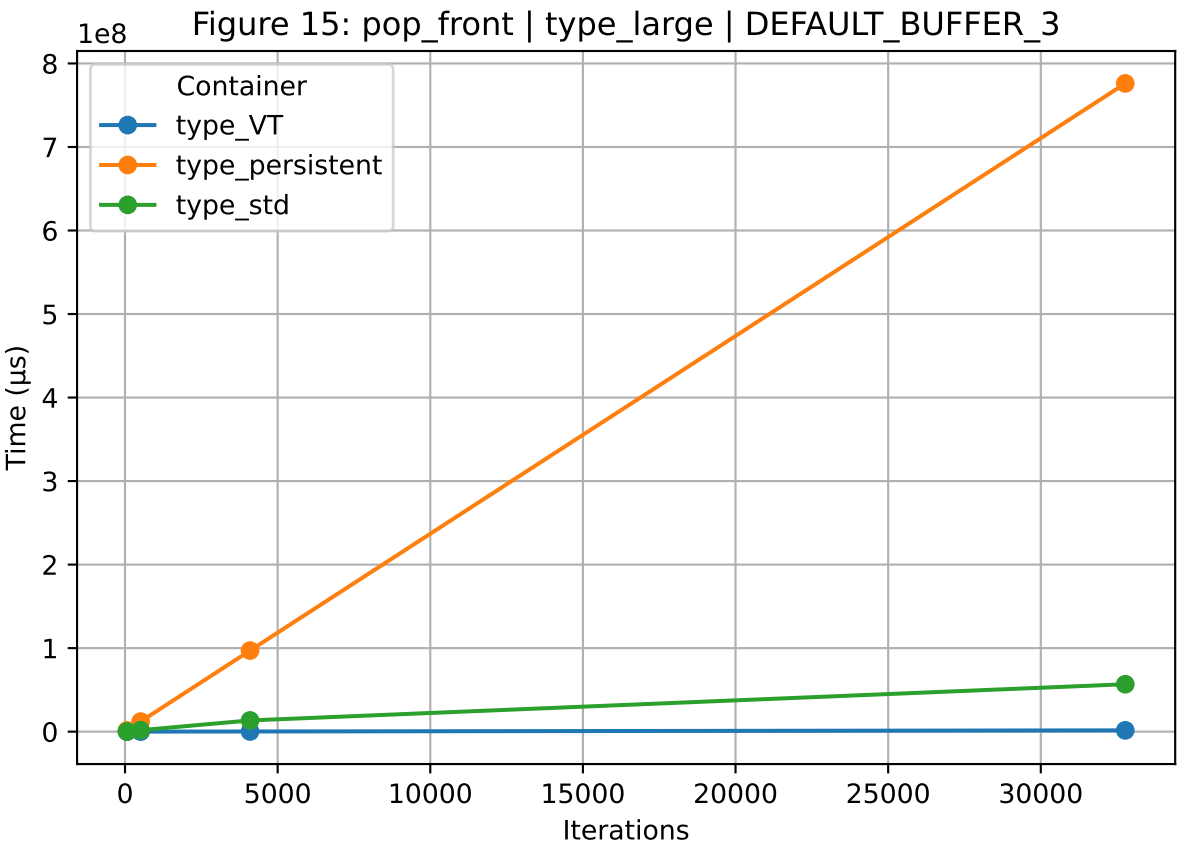
Figure 15: pop_front | type_large | DEFAULT_BUFFER_3
1e8
Container
type_VT
type_persistent
type_std
Time (µs)
Iterations
0
1
2
3
4
5
6
7
8
0
5000
10000
15000
20000
25000
30000

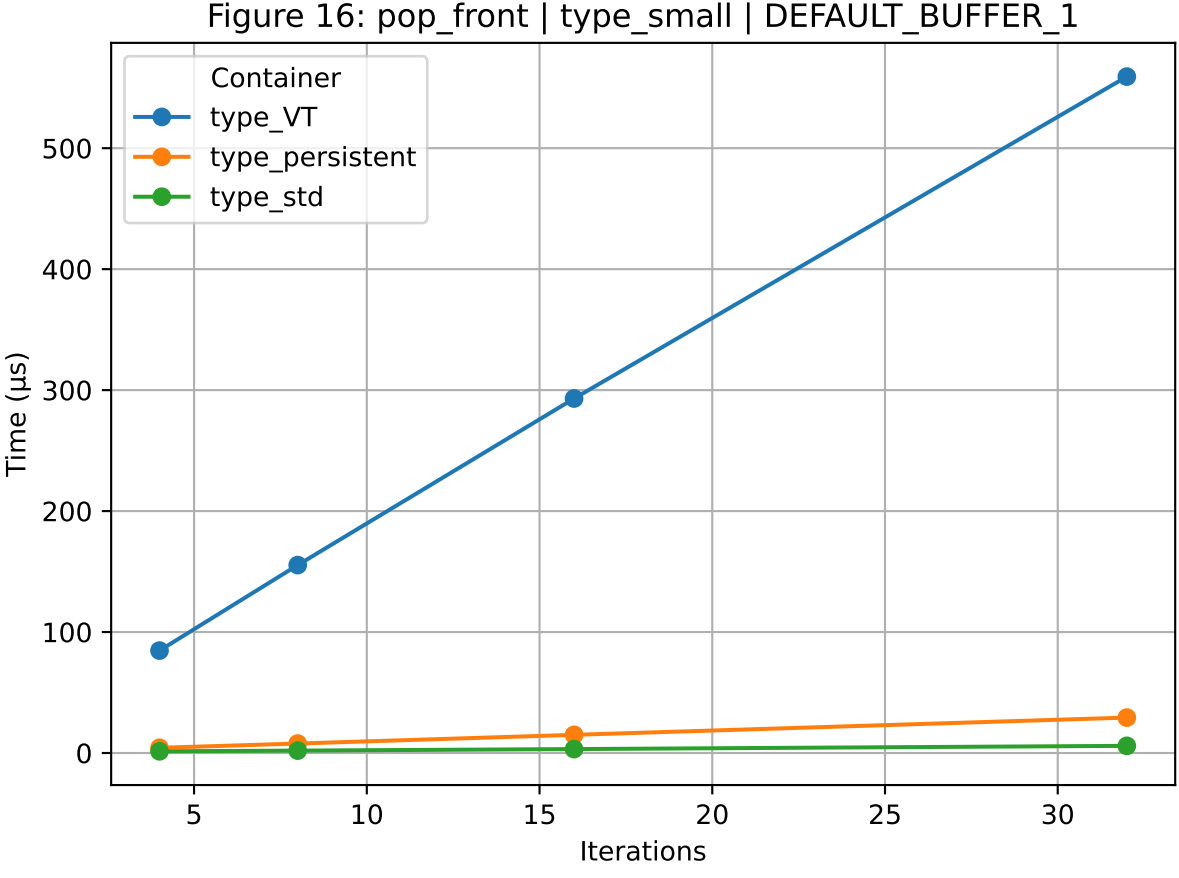
Figure 16: pop_front | type_small | DEFAULT_BUFFER_1
Container
type_VT
type_persistent
type_std
Time (µs)
0
100
200
300
400
500
5
10
15
20
25
30
Iterations

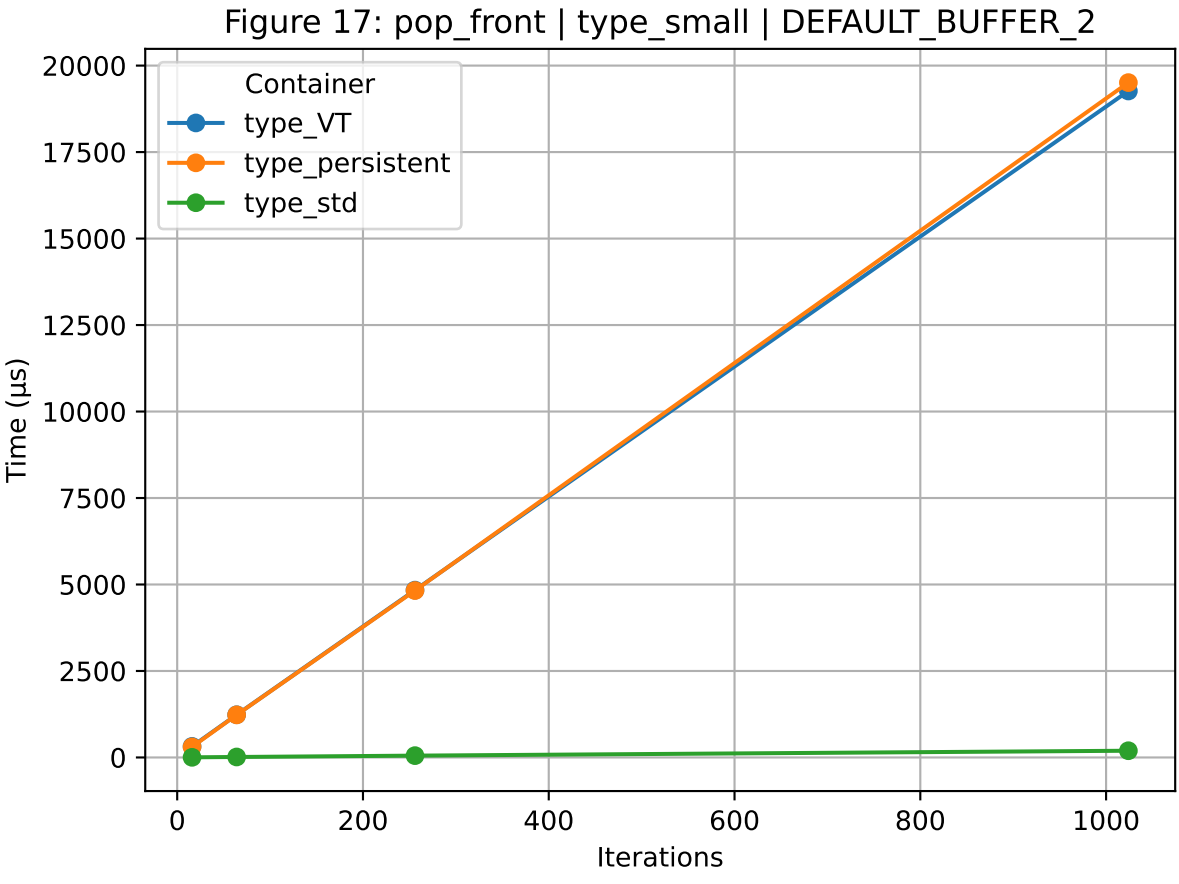
Figure 17: pop_front | type_small | DEFAULT_BUFFER_2
Container
type_VT
type_persistent
type_std
Time (µs)
Iterations
20000
17500
15000
12500
10000
7500
5000
2500
0
0
200
400
600
800
1000

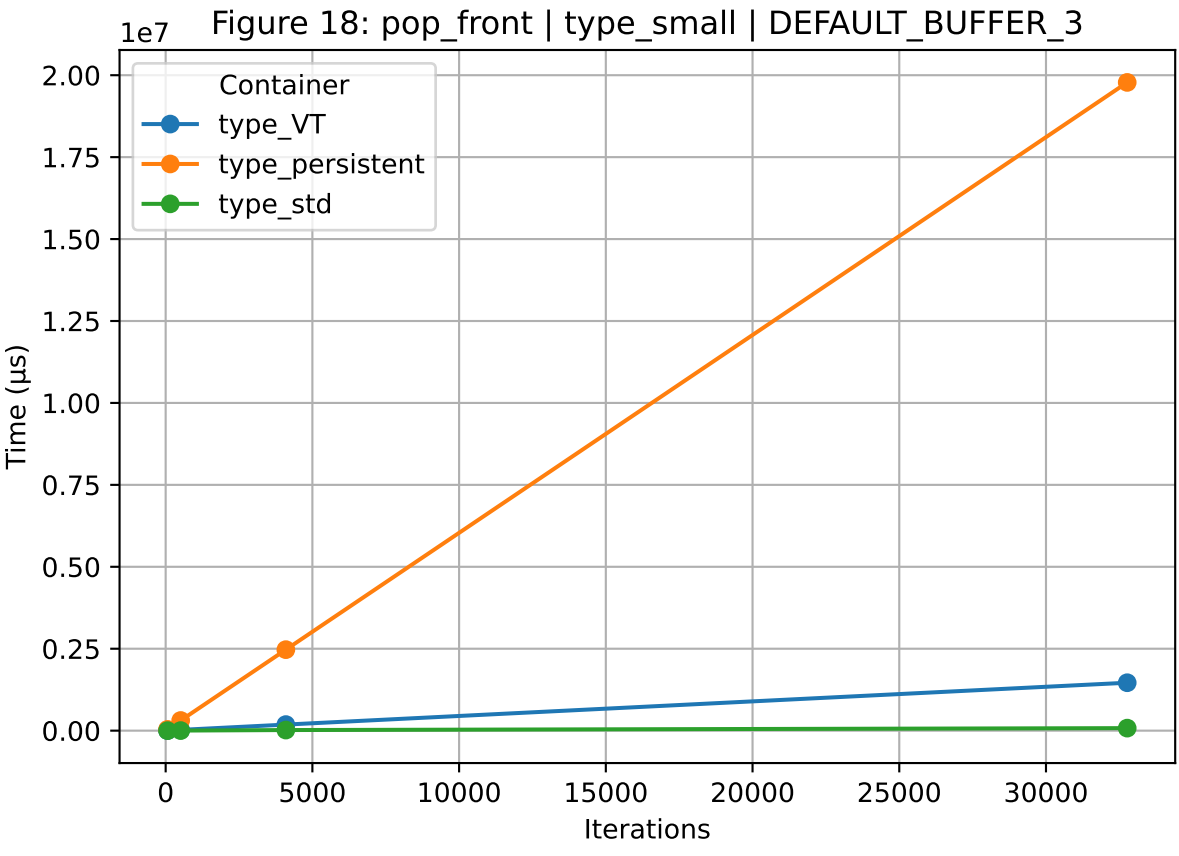
Figure 18: pop_front | type_small | DEFAULT_BUFFER_3
1e7
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
Time (µs)
0
5000
10000
15000
20000
25000
30000
Iterations
Container
type_VT
type_persistent
type_std

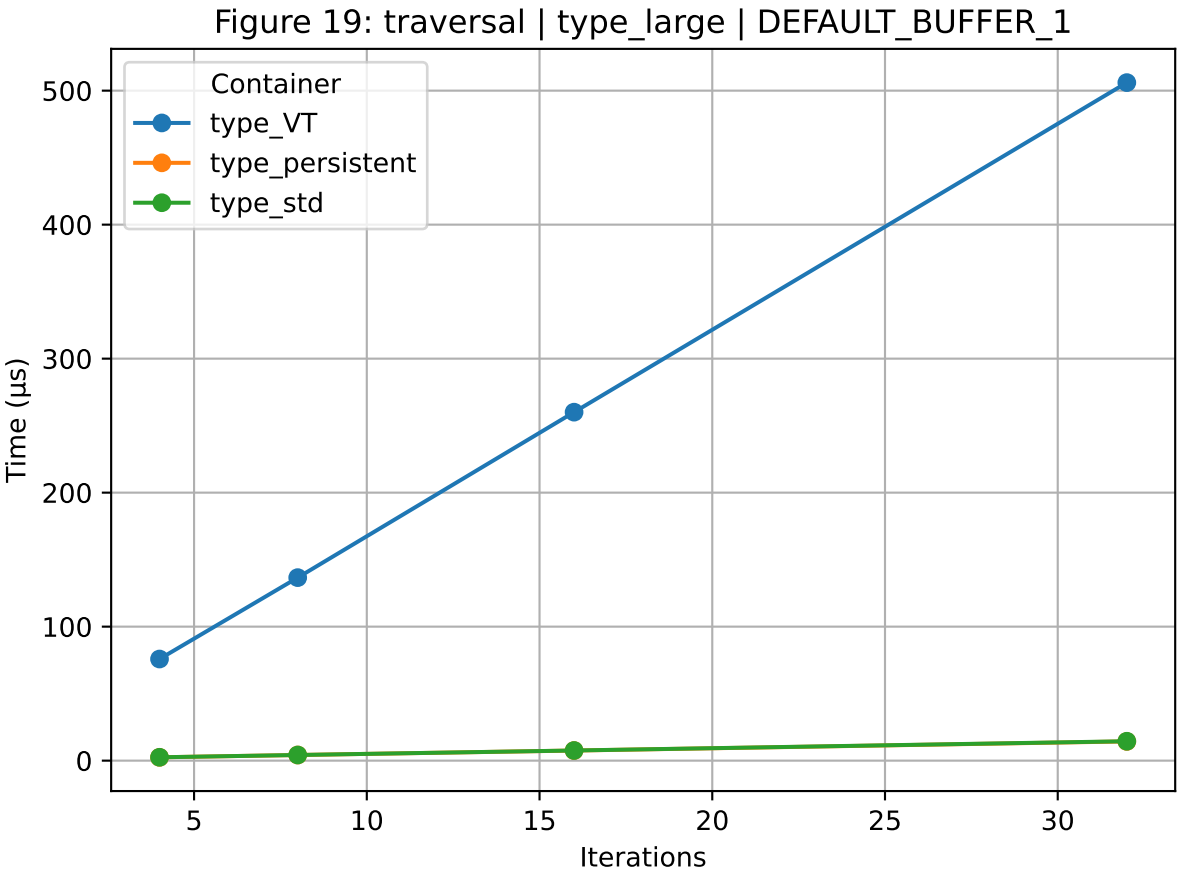
Figure 19: traversal | type_large | DEFAULT_BUFFER_1
Container
type_VT
type_persistent
type_std
Time (µs)
0
100
200
300
400
500
5
10
15
20
25
30
Iterations

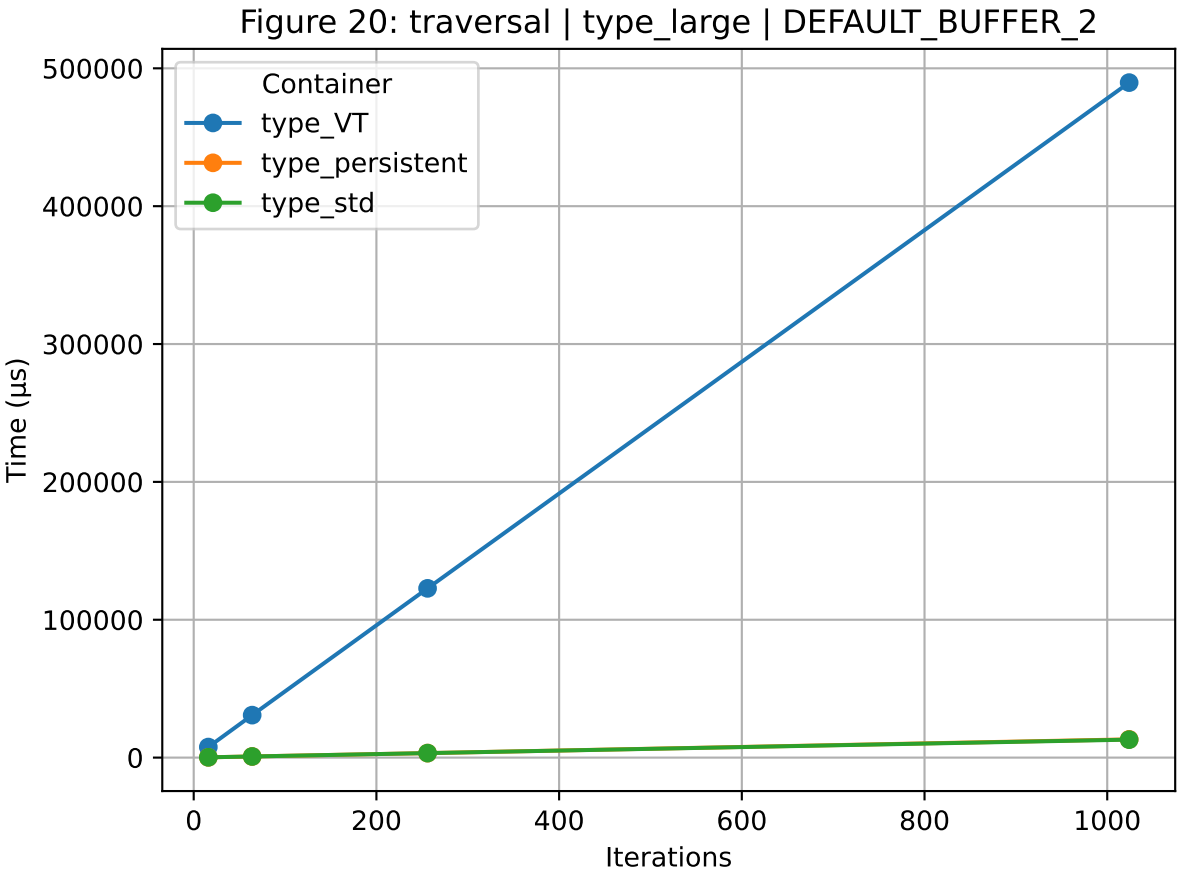
Figure 20: traversal | type_large | DEFAULT_BUFFER_2
Container
type_VT
type_persistent
type_std
Time (µs)
500000
400000
300000
200000
100000
0
0
200
400
600
800
1000
Iterations

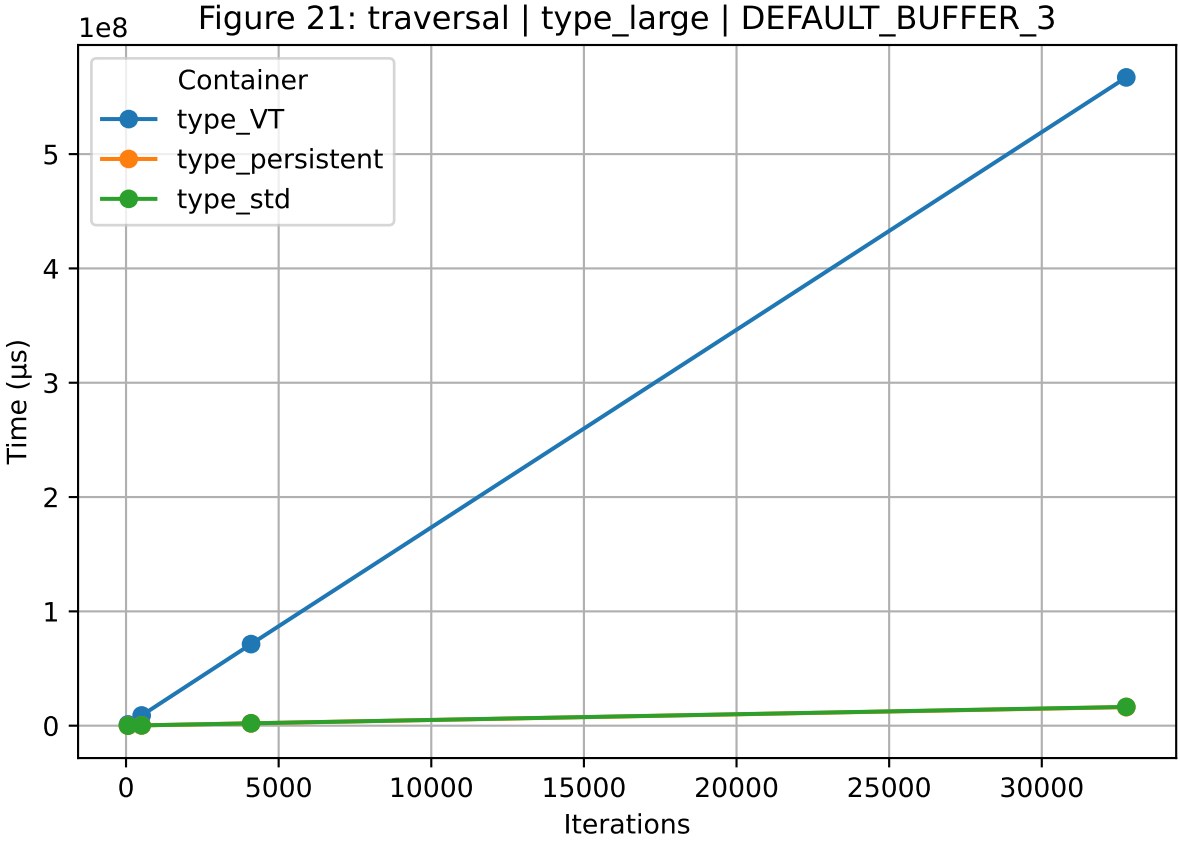
Figure 21: traversal | type_large | DEFAULT_BUFFER_3
1e8
Container
type_VT
type_persistent
type_std
Time (µs)
Iterations
0
1
2
3
4
5
0
5000
10000
15000
20000
25000
30000

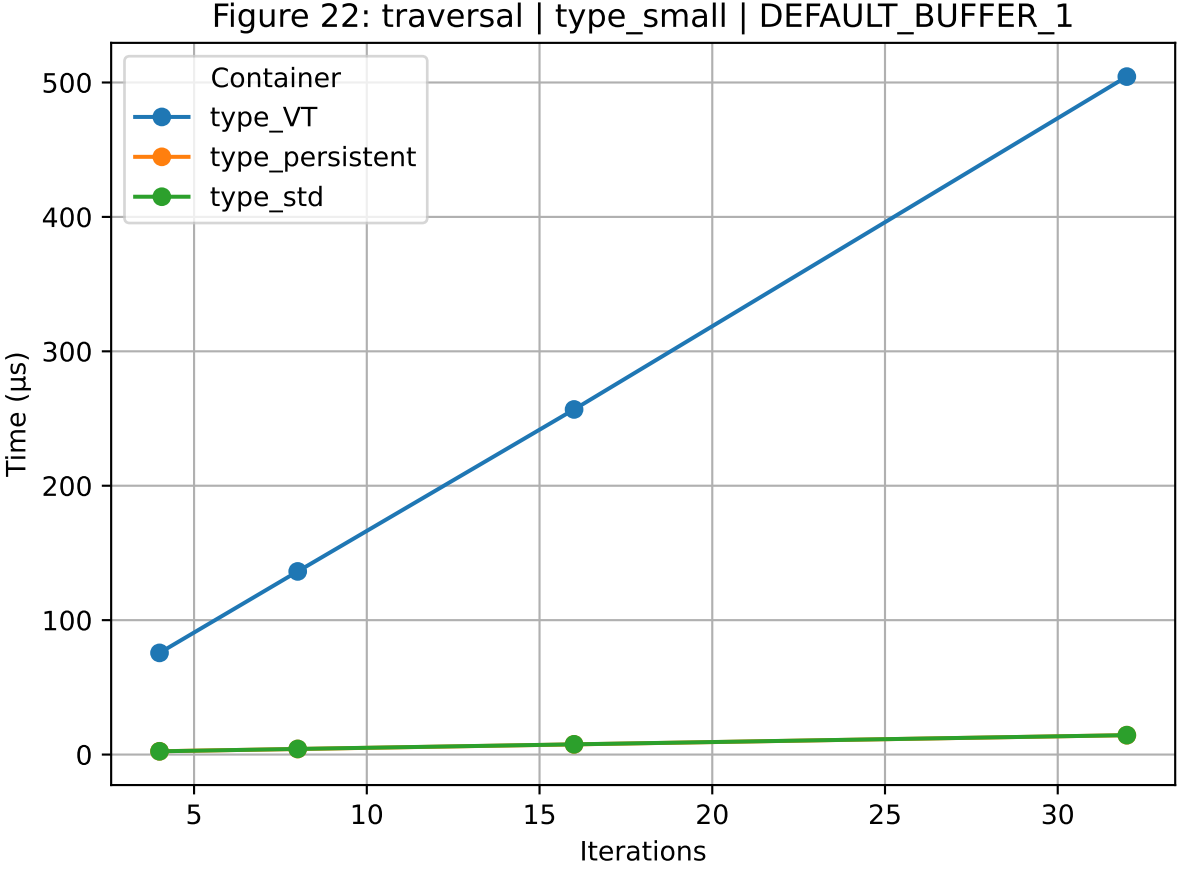
Figure 22: traversal | type_small | DEFAULT_BUFFER_1
Container
type_VT
type_persistent
type_std
Time (µs)
0
100
200
300
400
500
5
10
15
20
25
30
Iterations

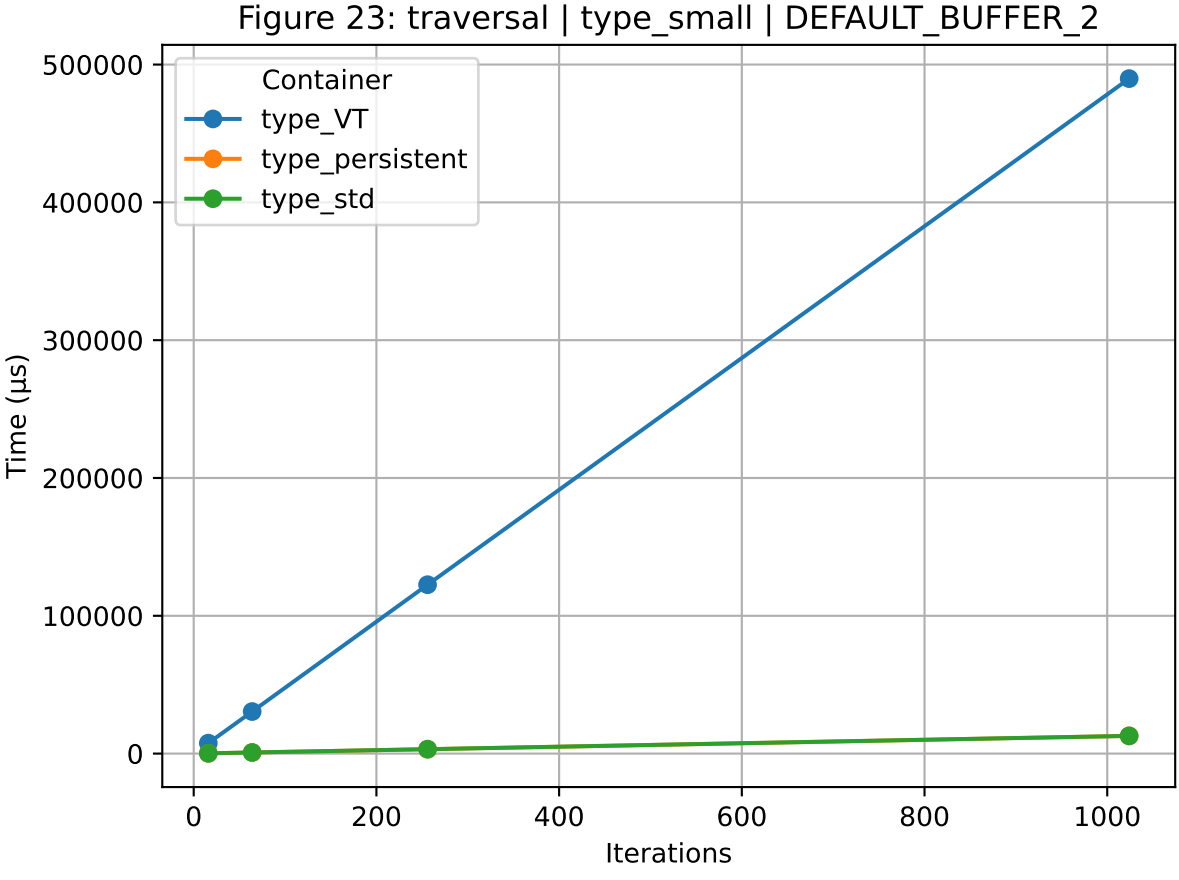
Figure 23: traversal | type_small | DEFAULT_BUFFER_2
Container
type_VT
type_persistent
type_std
500000
400000
300000
200000
100000
0
Time (µs)
0
200
400
600
800
1000
Iterations

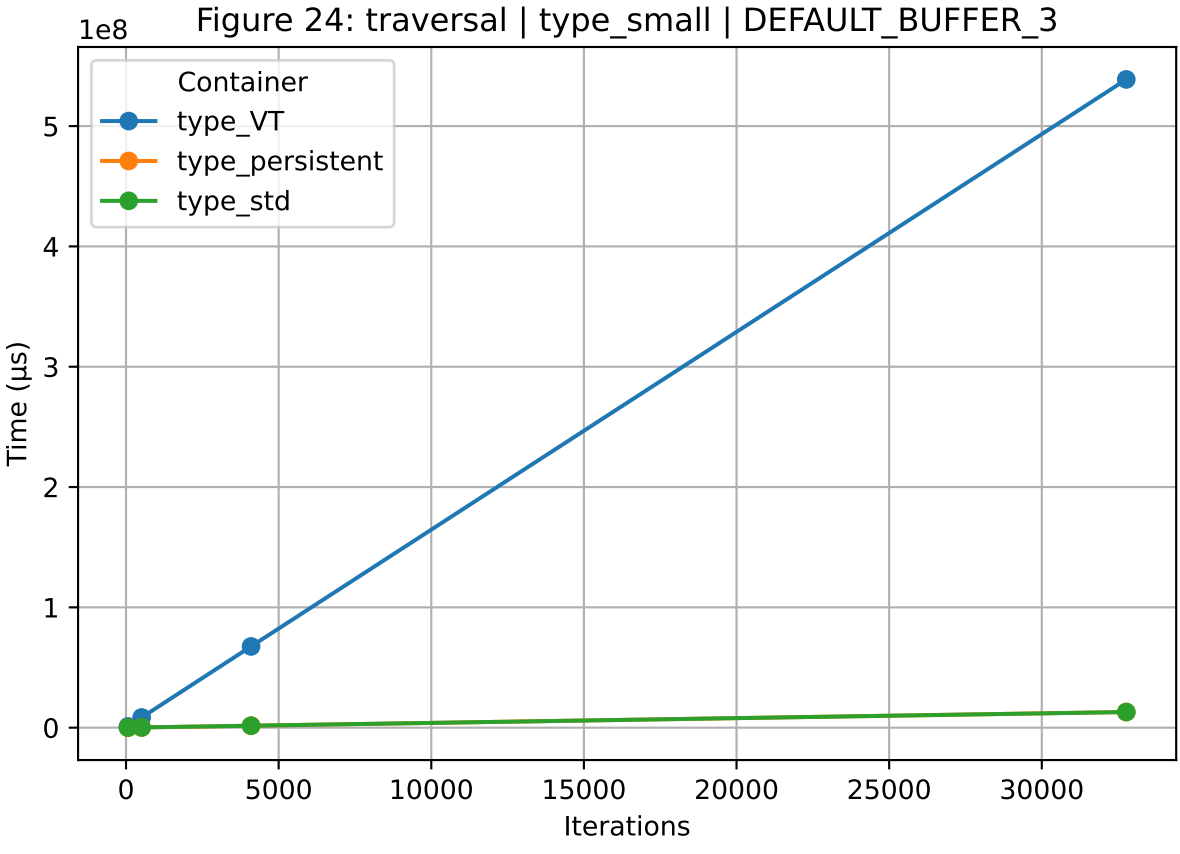
Figure 24: traversal | type_small | DEFAULT_BUFFER_3
1e8
Container
type_VT
type_persistent
type_std
Time (µs)
Iterations
0
1
2
3
4
5
0
5000
10000
15000
20000
25000
30000