Project 2

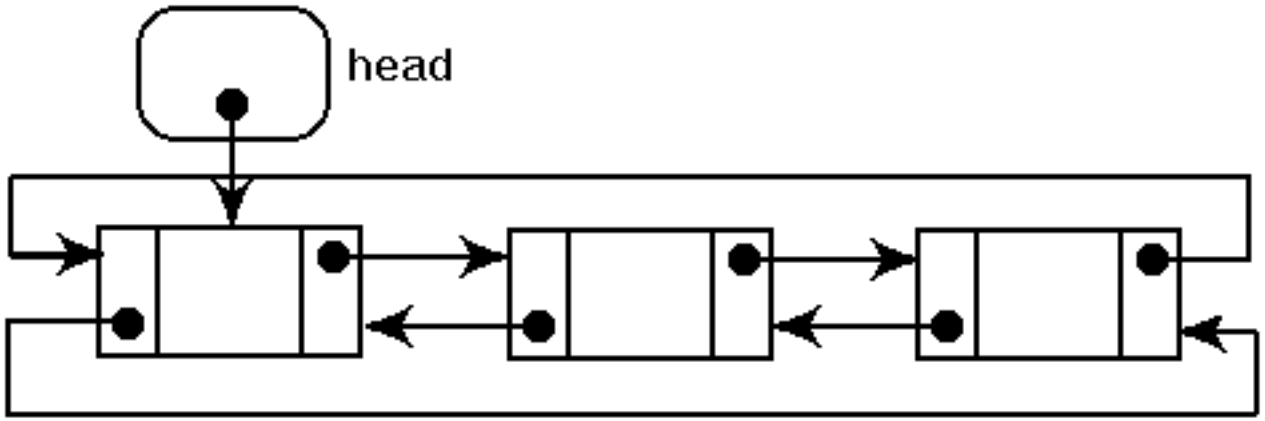
104021219 鄭余玄

Vector

- 一般 Class
  - new [], delete[]
  - placement new
- 判斷 POD
  - std::is_trivial<T>::value
  - memcpy, memmove
- 實作細節
  - begin() 位置為 data_[0]
  - end() 位置為 data_[size_]

List

- Doubly Linked Circular List
  - 快速找到 front(), back()
- 實作細節
  - begin() 位置為 head_->next
  - end() 位置為 head_



Doubly Linked Circular list

Set

- Binary Search Tree
- 實作細節
  - begin() 位置為第一個元素
  - end() 位置為額外設計的無限大（小）元素

Experiment

Experiment Environment

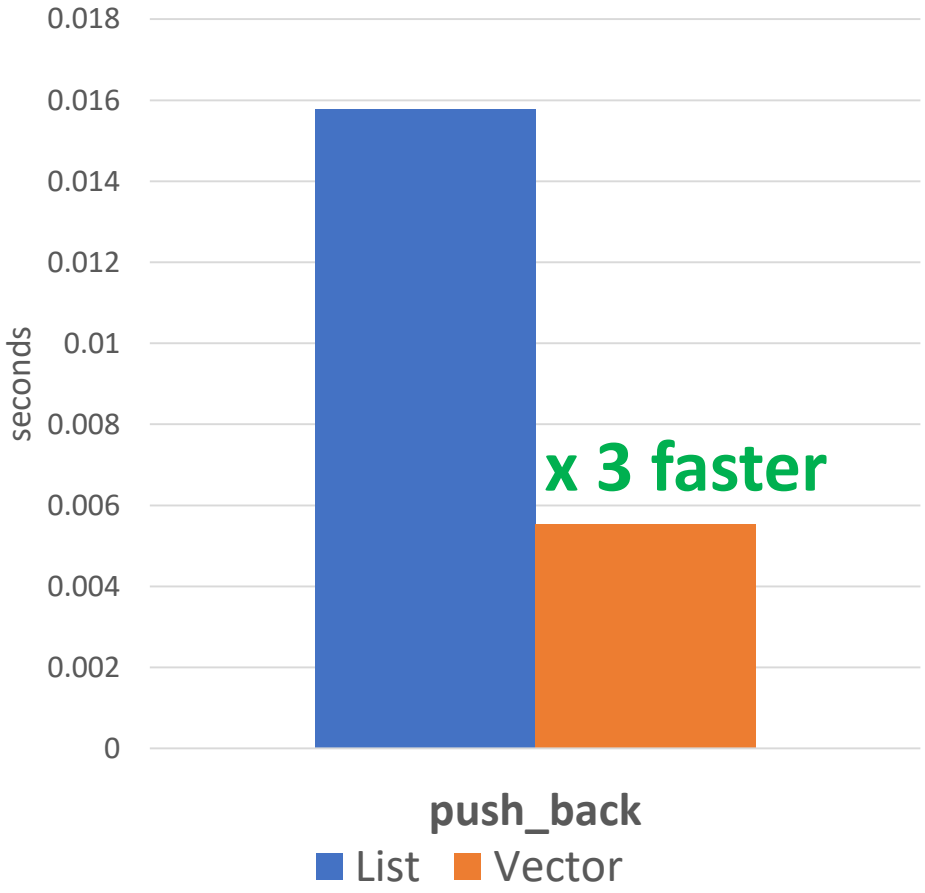
- Intel Xeon E5-2683 v3 @ 2.00GHz
- 350 GB DDR4 @ 2133MHz

- Arch Linux 4.13
- g++ 7.2.0
- taskset
- std::chrono::high_resolution_clock
- Valgrind (massif, memory leak check)

testcase

- N = 10,000
- random_shuffle 1 ~ N
- value_type = int64_t

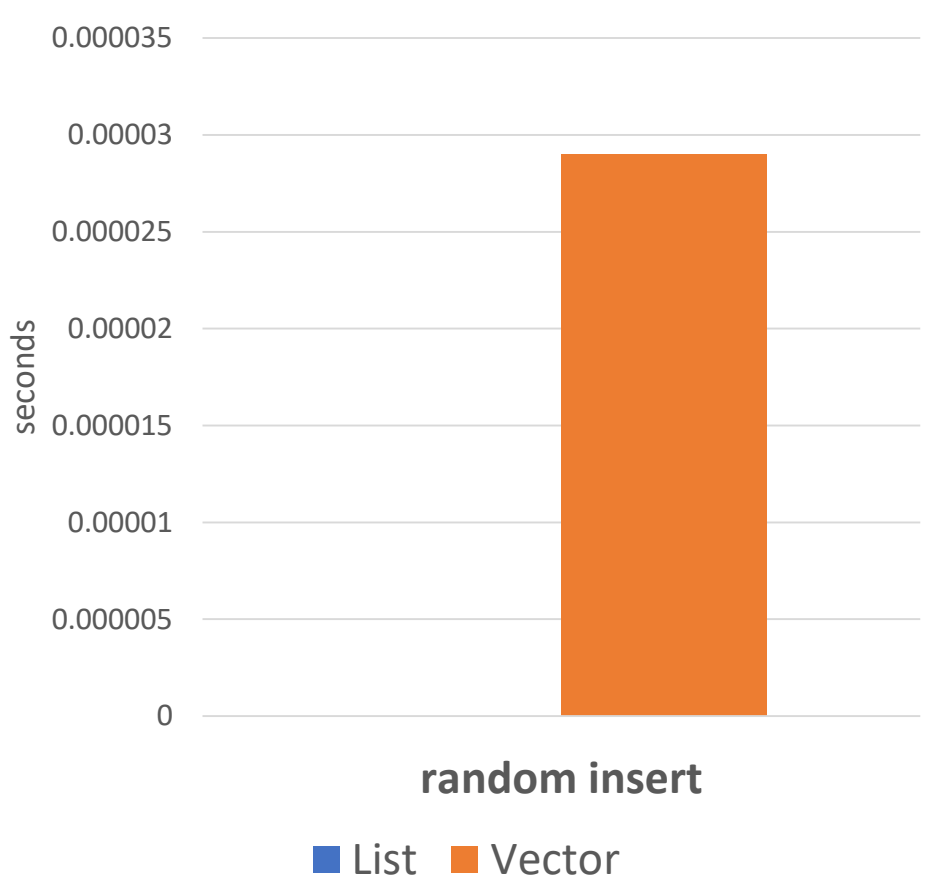
push front / end

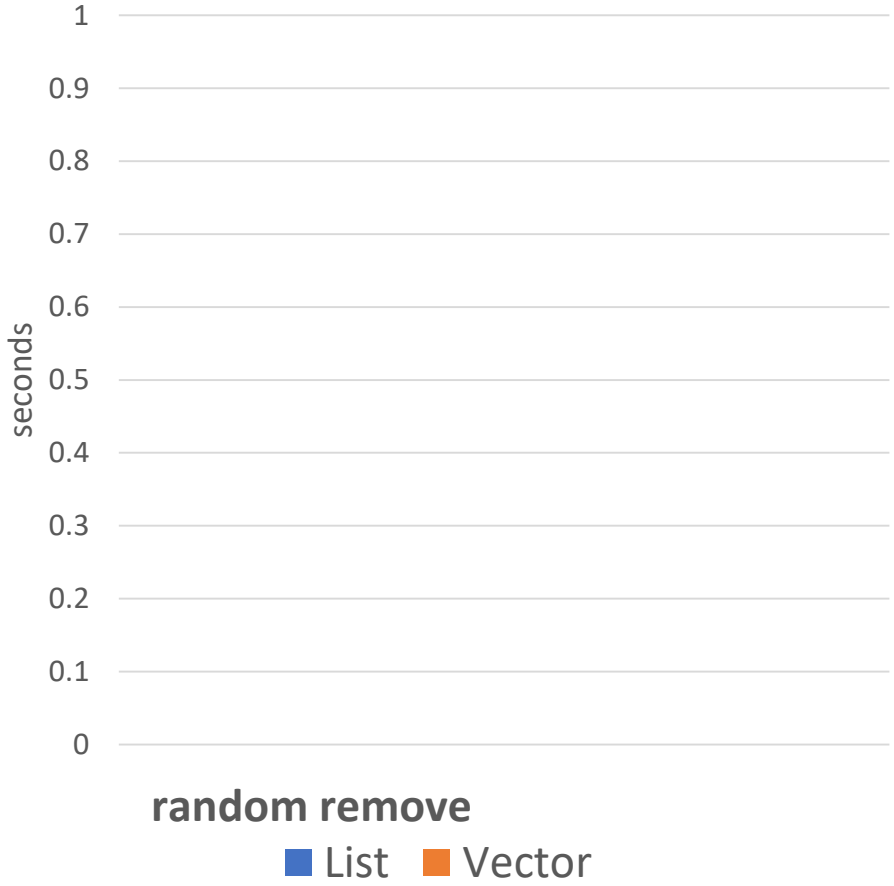


Comparison

- push_front
  - List 可以在開頭 O(1) 插入資料
  - Vector 需要進行 O(n) 資料搬移後，再插入資料
- push_back
  - 注意上頁圖表 y 軸執行時間數量級
  - List 可以在結尾 O(1) 插入資料
  - Vector 因為會預先 reserve 較大的記憶體空間，因此插入資料時會快許多
- 整體而言，List 最適合在頭尾插入元素
  - 因為不需要額外的記憶體搬移

random insert / erase

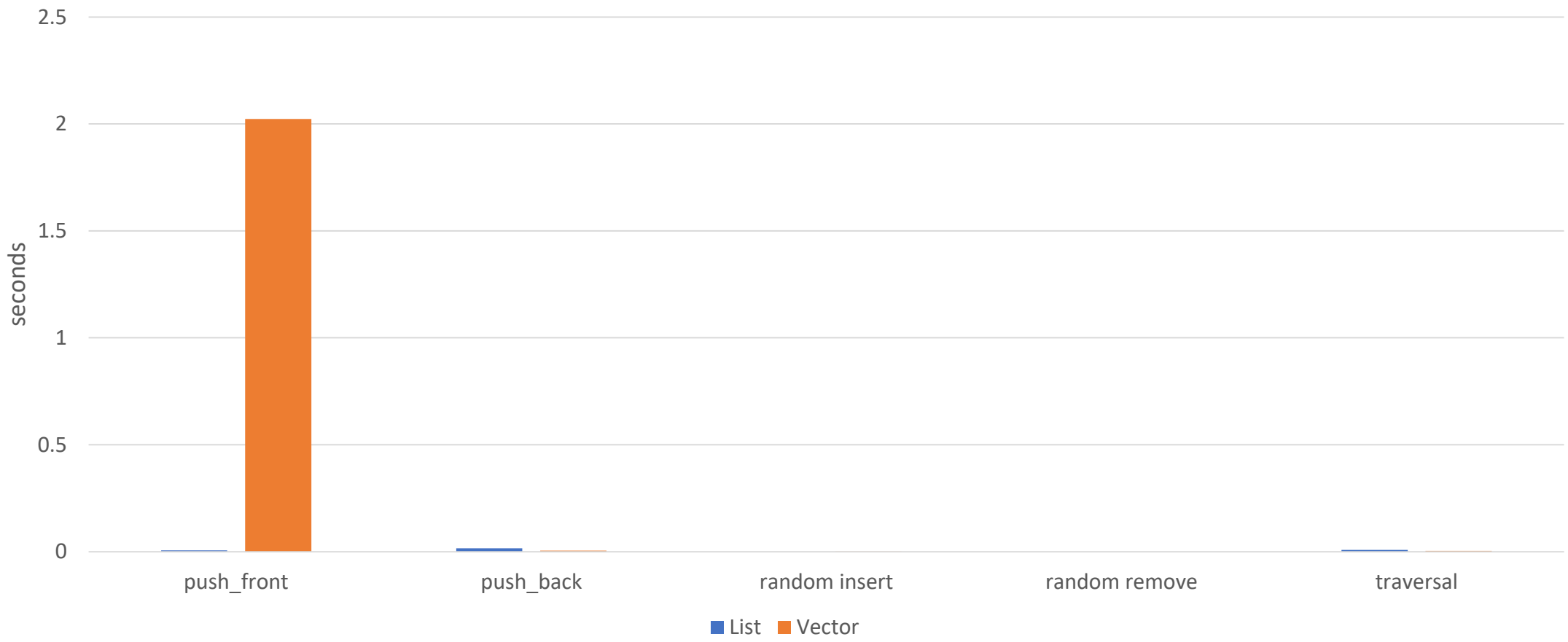




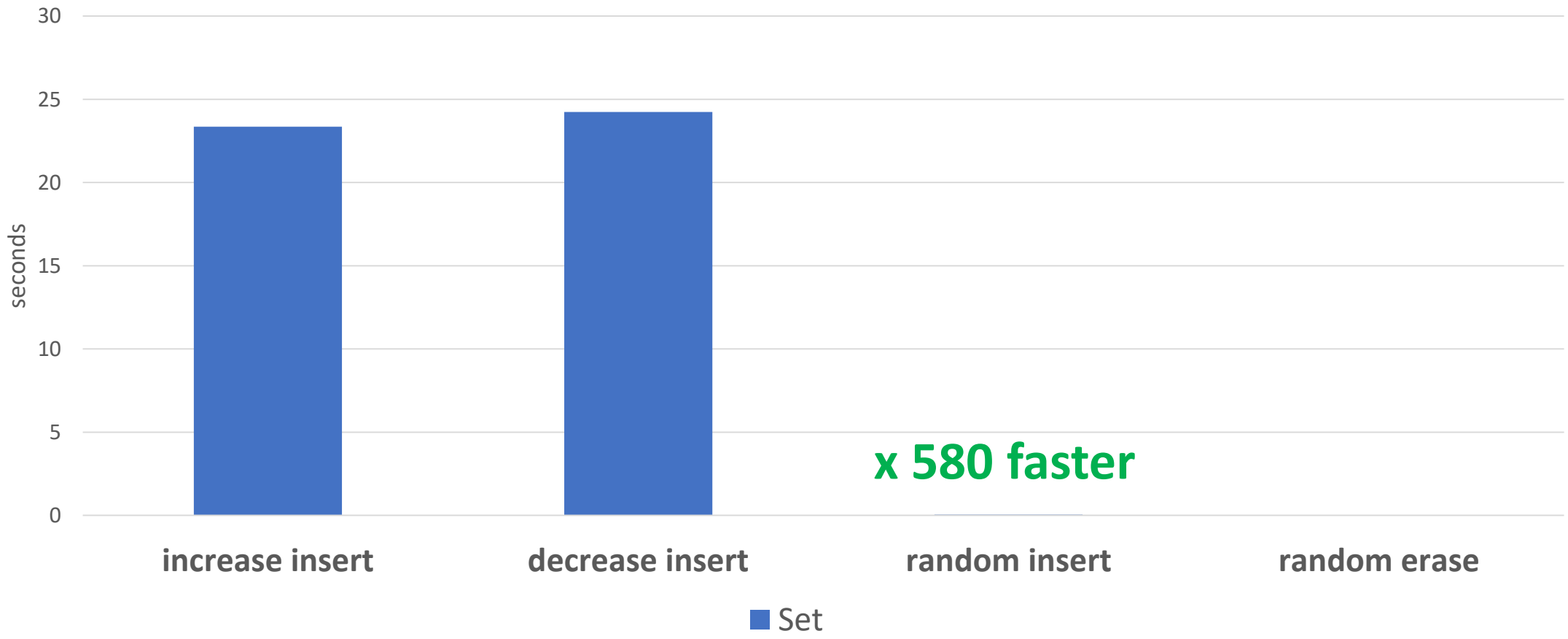
Comparison

- 隨機插入相較插入頭尾，兩者都十分快
  - List 插入只需要 O(1) 操作，因此在效能測試中時間幾乎是零
  - Vector 插入因為有可能需要搬移記憶體，因此略慢
- 隨機刪除兩者測量時間幾乎是零
- 整體而言，List 適合頻繁的隨機插入或刪除

List & Vector


2.5
2
1.5
1
0.5
0
seconds
push_front
push_back
random insert
random remove
traversal
List
Vector

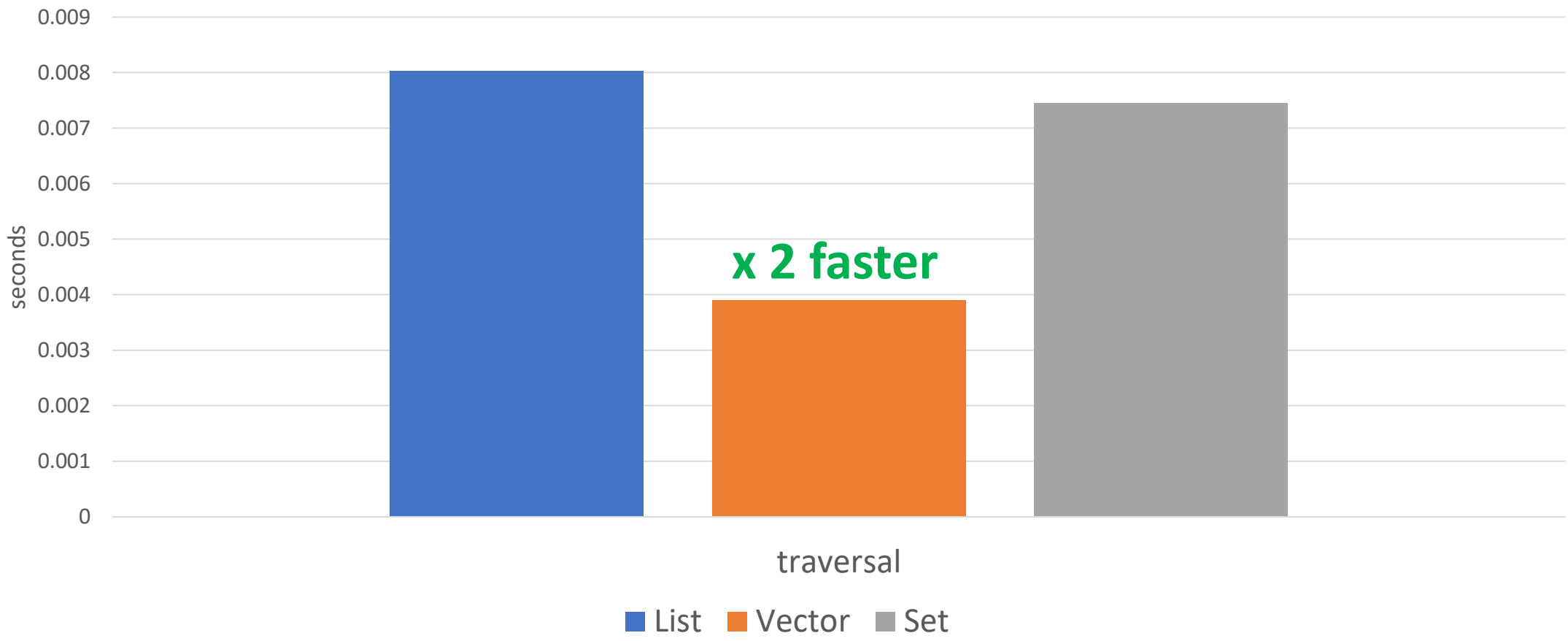
Set insert / erase



Comparison

- 插入隨機元素相較連續遞增、遞減的元素有顯著好的效能
  - 因為 Set 實作是使用 Binary Search Tree，對於前者會期望產生一顆平衡樹，而後者等同於左偏或右偏的 Linked List
  - 在底層樹的實作上，前者深度期望是 O(lg n)，而後者會是 O(n)
  - 因此 input 插入順序**會**影響效能
- 隨機刪除測量時間幾乎為零，因為移除只需 O(1) 操作和 List 相似

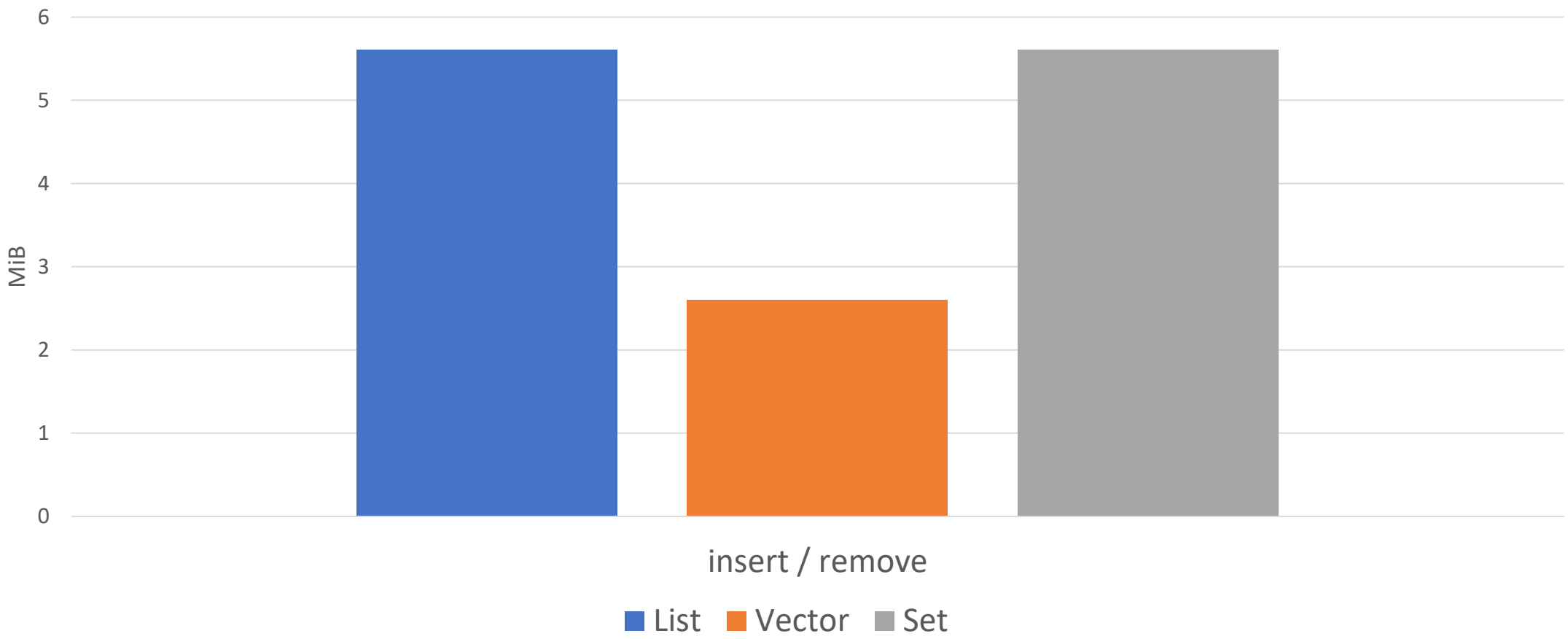
Traversal


0.009
0.008
0.007
0.006
0.005
0.004
0.003
0.002
0.001
0
seconds
x 2 faster
traversal
List
Vector
Set

Comparison

- Vector 遍歷所有元素有兩倍效能
  - 因為記憶體是連續的，下一個元素可以快速找到並 dereference
- List 和 Set 底層都是使用相同資料結構
  - 訪問下一個元素能需要遍歷到下一個記憶體位置
- 整體而言，Vector 適合頻繁遍歷所有元素

Memory Peak



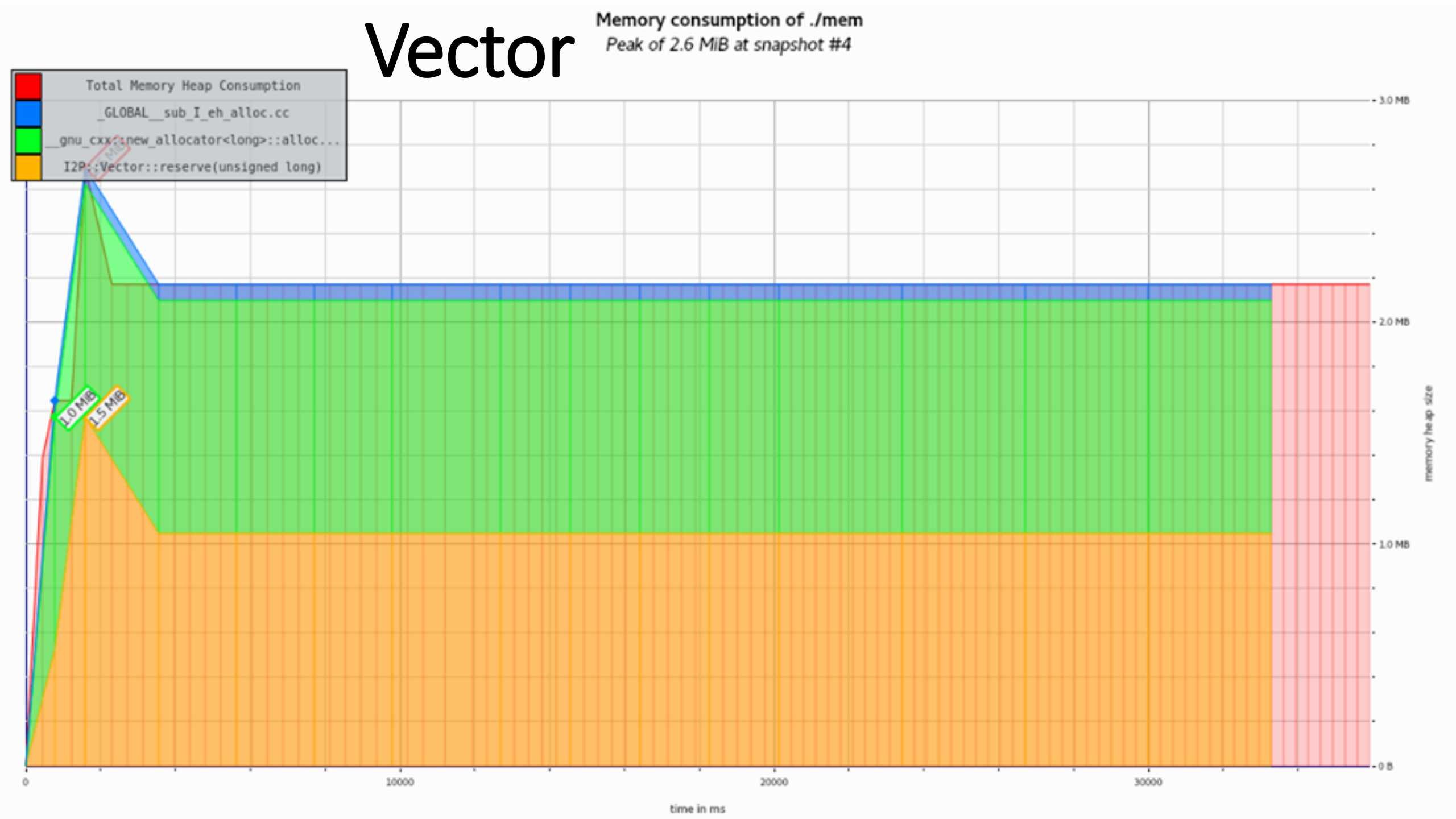
Comparison

- List 和 Vector 底層使用相同資料結構，因此最高記憶體用量相近
- Vector 只需要配置資料在連續記憶體，因此少了一倍的用量
- 整體而言，Vector 最有效運用記憶體

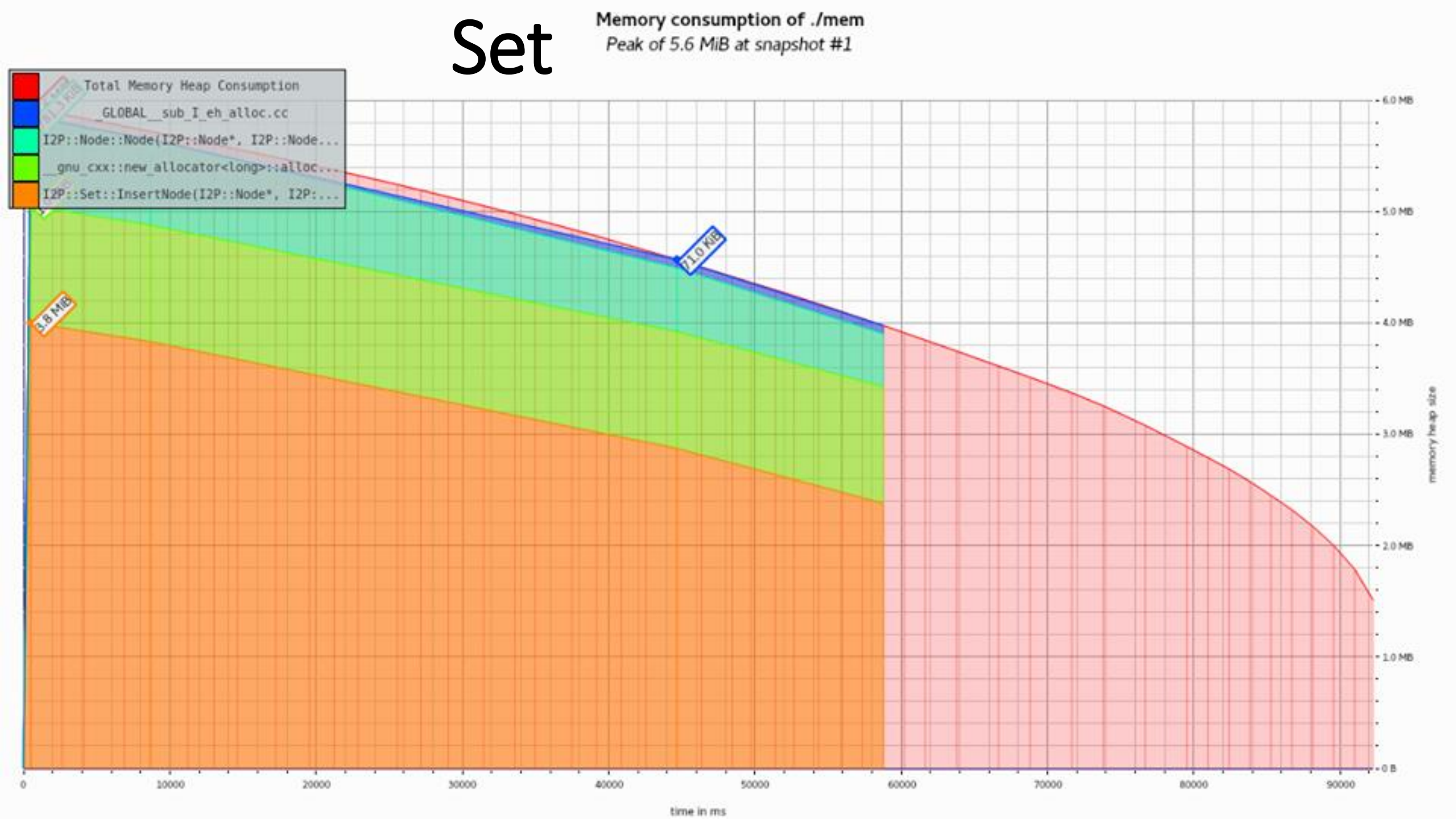
List


Memory consumption of ./mem
Peak of 5.6 MiB at snapshot #31
Total Memory Heap Consumption
_GLOBAL__sub_I_eh_alloc.cc
I2P::Node::Node(I2P::Node*, I2P::Node...
__gnu_cxx::new_allocator<long>::alloc...
I2P::List::insert(I2P::const_iterator...
773.2 KiB
3.8 MiB
1.0 MiB
6.0 MB
5.0 MB
4.0 MB
3.0 MB
2.0 MB
1.0 MB
0 B
memory heap size
0
10000
20000
30000
40000
50000
60000
70000
80000
time in ms

Vector


Memory consumption of ./mem
Peak of 2.6 MiB at snapshot #4
Total Memory Heap Consumption
_GLOBAL__sub_I_eh_alloc.cc
__gnu_cxx::new_allocator<long>::alloc...
I2P::Vector::reserve(unsigned long)
1.0 MB
1.5 MB
3.0 MB
2.0 MB
1.0 MB
0 B
memory heap size
0
10000
20000
30000
time in ms

Set


Memory consumption of ./mem
Peak of 5.6 MiB at snapshot #1
Total Memory Heap Consumption
_GLOBAL__sub_I_eh_alloc.cc
I2P::Node::Node(I2P::Node*, I2P::Node...
__gnu_cxx::new_allocator<long>::alloc...
I2P::Set::InsertNode(I2P::Node*, I2P:...
71.0 KiB
3.8 MiB
6.0 MB
5.0 MB
4.0 MB
3.0 MB
2.0 MB
1.0 MB
0 B
memory heap size
0
10000
20000
30000
40000
50000
60000
70000
80000
90000
time in ms

Previous Memory Leak

- 透過 massif 可以發現之前曾有嚴重的 Memory Leak