Performance Analysis and Optimization of C++ Standard Libraries

Sebastian Pop, Aditya Kumar

SARC: Samsung Austin R&D Center

October 26, 2017

Agenda

- C++ standard template libraries
- software performance analysis
- improvements to libc++ and libstdc++ performance

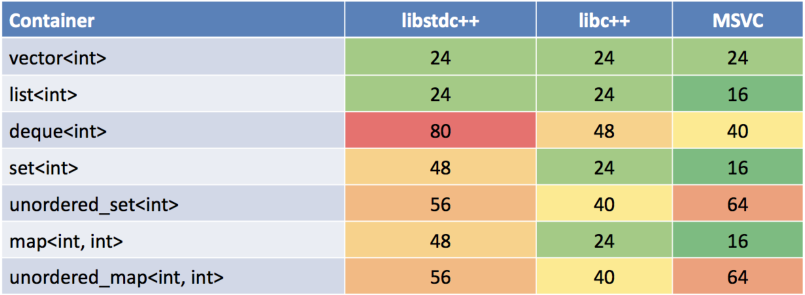
C++ Standard Template Libraries

- STL is easy to use
    - standard interface: portable
    - easy to change data types: list, vector, deque, map, etc.
    - easy to change algorithms: iterators
- complexity of operations specified by standard
- performance left to implementation

Performance of STL implementations

- performance and memory usage depend on
  - implementation: libc++ vs. libstdc++ vs. MSVC, etc.
  - container type
  - inefficiencies in implementations
- always analyze software performance to validate your choice

Figure: Size in bytes of empty containers on x86_64



Container	libstdc++	libc++	MSVC
vector<int>	24	24	24
list<int>	24	24	16
deque<int>	80	48	40
set<int>	48	24	16
unordered_set<int>	56	40	64
map<int, int>	48	24	16
unordered_map<int, int>	56	40	64

Software Performance Analysis

- identify hot functions from execution profiles
- inspect hot path: unit-benchmarking
- identify resource utilization on hot path

Profiling: identify hot path

- linux-perf, oprofile: cycles, instructions, HW counters
- valgrind: cachegrind (R/W/Instrs), callgrind

Unit-benchmarking: inspect hot path

unit-benchmarking is unit-testing for performance

- set-up data structures in memory
- time hot function
- execute hot function until performance measures stabilize [1]

check performance of a single hot operation: less noise, keep focus

[1] https://github.com/google/benchmark

Example: performance analysis of std::string.find()

```
$ git clone https://github.com/hiraditya/std-benchmark.git && cd std-benchmark
$ git submodule update --recursive --remote
$ mkdir build && cd build && cmake .. && make -j8
$ cat ../cxx/string.bench.cpp
[...]
  while (state.KeepRunning()) {
    pos = s1.find(s2);
    benchmark::DoNotOptimize(pos);
  }
[...]
$ perf record ./cxx/string.bench.cpp.out --benchmark_filter=BM_find/16384
$ perf report
```

```
  79.71% libstdc++.so.6.0.21   [.] std::__cxx11::basic_string<char, [...] >::find
```

```
 14.08 |78:    add    $0x1,%rbx
 13.82 |       add    $0x1,%rbp
 12.74 |       cmp    %r15,%rax
       |       ja     b8
 18.21 |85:    cmp    -0x1(%rbp),%r14b
 13.16 |       lea    -0x1(%rbx),%r13
 13.28 |       mov    %rbx,%rax
 14.62 |       jne    78
```

Analyze resource utilization on hot path

Inspect:

- source code, compiler IR, assembly code
- CPU usage, instructions used and their latencies
- memory bus and caches: loads/stores, spills, cache misses

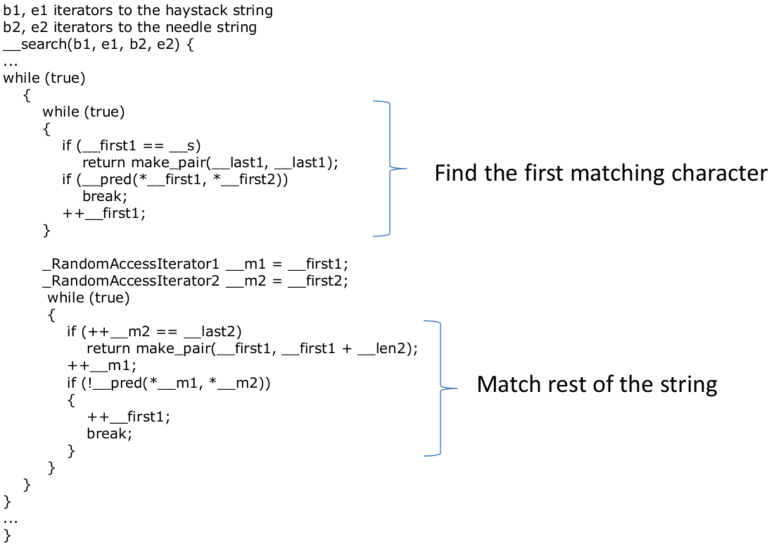
Improve Software Performance

- eliminate unnecessary work
  - call functionality from libc or libc++
  - reduce bus traffic: vectorize loads and stores
  - help compiler remove redundancies: attributes and inline
- analyze performance of different implementations
  - change data structures
  - change algorithms
  - change STL implementations
- analyze trade-offs of caching previous results
  - use more memory vs. less computation (and vice versa)

Our contributions to libc++ and libstdc++

- std::string.find(): find string within string
- xsgetn in libc++: string to int value parsing
- inline several ctors and dtors
- add attribute noreturn to non-returning functions
- fixed quadratic behavior of std::sort of libc++

Issues with std::string.find (libc++ and libstdc++)



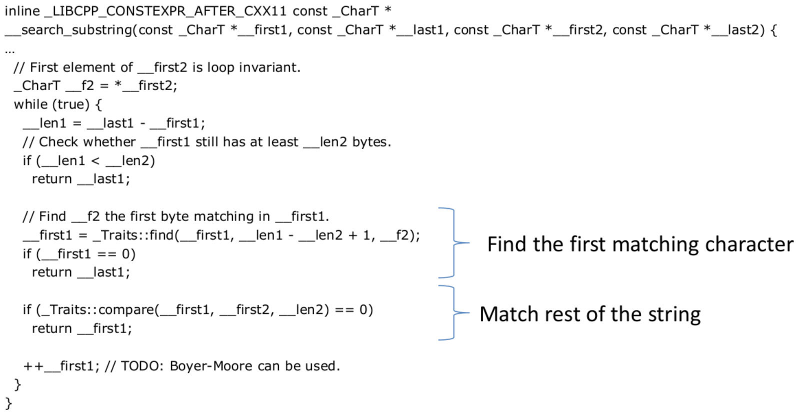
```
b1, e1 iterators to the haystack string
b2, e2 iterators to the needle string
__search(b1, e1, b2, e2) {
...
while (true)
   {
      while (true)
      {
         if (__first1 == __s)
            return make_pair(__last1, __last1);
         if (__pred(*__first1, *__first2))
            break;
         ++__first1;
      }

      _RandomAccessIterator1 __m1 = __first1;
      _RandomAccessIterator2 __m2 = __first2;
       while (true)
       {
          if (++__m2 == __last2)
             return make_pair(__first1, __first1 + __len2);
          ++__m1;
          if (!__pred(*__m1, *__m2))
          {
             ++__first1;
             break;
          }
       }
   }
}
...
}
```

Find the first matching character

Match rest of the string

Improved std::string.find (libc++ and libstdc++)



```
inline _LIBCPP_CONSTEXPR_AFTER_CXX11 const _CharT *
__search_substring(const _CharT *__first1, const _CharT *__last1, const _CharT *__first2, const _CharT *__last2) {
...
 // First element of __first2 is loop invariant.
 _CharT __f2 = *__first2;
 while (true) {
  __len1 = __last1 - __first1;
  // Check whether __first1 still has at least __len2 bytes.
  if (__len1 < __len2)
   return __last1;

  // Find __f2 the first byte matching in __first1.
  __first1 = _Traits::find(__first1, __len1 - __len2 + 1, __f2);
  if (__first1 == 0)
   return __last1;

  if (_Traits::compare(__first1, __first2, __len2) == 0)
   return __first1;

  ++__first1; // TODO: Boyer-Moore can be used.
 }
}
```

Find the first matching character

Match rest of the string

- replace byte by byte compare with call to memchr + memcmp
- 12x speedup on std-benchmark [2]

[2] https://github.com/hiraditya/std-benchmark

Our contributions to libc++

- string to int value parsing: xsgetn in libc++
  - replace byte by byte copy with call to libc memcpy
  - important speedup on proprietary benchmark
- inline ctor/dtor
  - shared_ptr
  - basic_string
- add attribute noreturn to non-returning functions
  - __locale, vector, deque, future, regex, system_error, etc.
  - important for compiler optimizations
  - remove false positives in static analysis tools

Issues with std::sort (libc++)

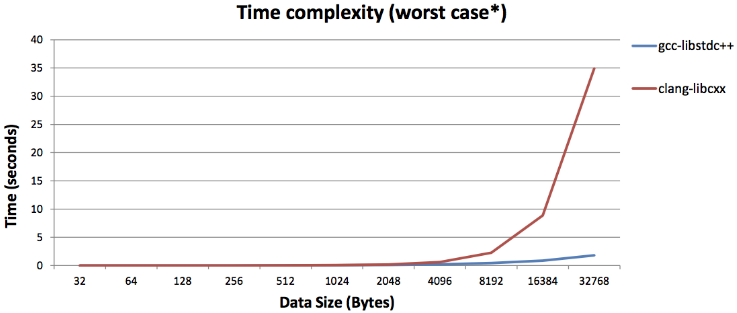


Figure: Quadratic behavior of libc++ compared to libstdc++

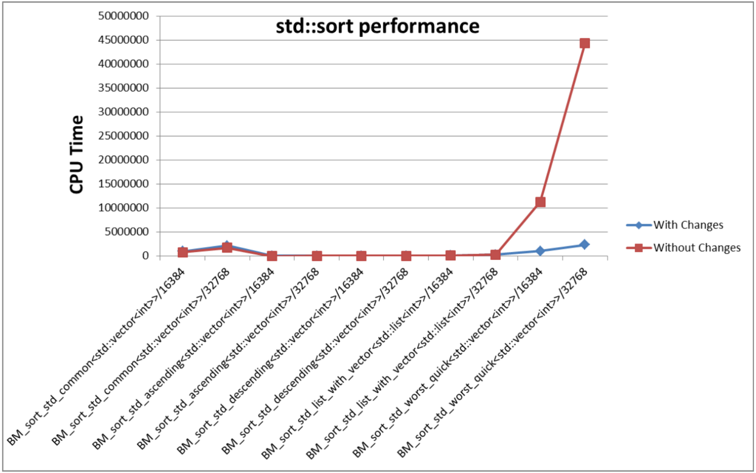
* `https://bugs.llvm.org/show_bug.cgi?id=20837`

std::sort (libc++)

- Convert to introsort
- Sorting technique, which begins with quicksort and switches to heapsort after recursion reaches a threshold
- Worst case complexity of O(NlogN)
- Eliminate recursion
- Replaced memory intensive recursive calls with stack std::stack uses std::deque, which uses std::algorithm
- Improved worst case time complexity by a factor of 10 `https://reviews.llvm.org/D36423`
- quicksort with tail recursion elimination: quadratic worst case
- reimplemented as introsort: begin with quicksort, switch to heapsort when recursion depth goes beyond a threshold
- 16x speedup in the worst case (std-benchmark [3])

[3] `https://github.com/hiraditya/std-benchmark`

Sorting Results Plot (With std-benchmark)



Questions