Performance analysis and optimization of C++ standard libraries

Aditya Kumar

Samsung Austin R&D Center

Suboptimal implementation of basic_streambuf::xsgetn

```
template <class _CharT, class _Traits>
streamsize
basic_streambuf<_CharT, _Traits>::xsgetn(char_type* __s, streamsize __n)
{
    const int_type __eof = traits_type::eof();
    int_type __c;
    streamsize __i = 0;
    for (;__i < __n; ++__i, ++__s)
    {
        if (__ninp_ < __einp_)
            *__s = *__ninp_++;
        else if ((__c = uflow()) != __eof)
            *__s = traits_type::to_char_type(__c);
        else
            break;
    }
    return __i;
}
```

Suboptimal implementation of basic_streambuf::xsgetn

```
template <class _CharT, class _Traits>
streamsize basic_streambuf<_CharT, _Traits>::xsgetn(char_type* __s, streamsize __n) {
    const int_type __eof = traits_type::eof();
    int_type __c;
    streamsize __i = 0;
    while(__i < __n)
    {
        if (__ninp_ < __einp_)
        {
            const streamsize __len = _VSTD::min(__einp_ - __ninp_, __n - __i);
            traits_type::copy(__s, __ninp_, __len);
            __s +=  __len;
            __i +=  __len;
            this->gbump(__len);
        }
        else if ((__c = uflow()) != __eof)
        {
            *__s = traits_type::to_char_type(__c);
            ++__s;
            ++__i;
        }
        else
            break;
    }
    return __i;
}
```

Performance improvements

Valgrind profile of a synthetic test case which only exercises xsgetn.

```
struct test
    : public std::basic_streambuf<char> {

typedef std::basic_streambuf<char> base;
test() {}

void
setg(char* gbeg, char* gnext, char* gend) {
    base::setg(gbeg, gnext, gend);
}
};
```

```
int foo(char* input, char *output, int N) {
    test t;
    t.setg(input, input, input+N);
    char* pos = output;
    pos += t.sgetn(pos, N);
    return *pos;
}
```

	Base compiler without patch	Base compiler with patch
Total no of instructions (valgrind)	1,378,842	1,359,235
basic_streambuf::xsgetn (char*, long)	20,015	0

Improvements to string::find algorithm

- Used to call the (suboptimal) generic std::find function
- Solution:
  - Separately implement string::find
  - The new algorithm gets converted to optimized versions of memchr and memcmp

string::find original implementation

```
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

string::find new algorithm

```
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;
  }
}
```

Find the first matching character

Match rest of the string

Experimental results

Benchmark	Without patch	With patch	Gain
BM_StringFindMatch1/32768	28157 ns	2203 ns	12.8x
BM_StringFindMatch2/32768	28161 ns	2204 ns	12.8x

```
// Match somewhere towards the end
static void
BM_StringFindMatch1(benchmark::State &state)
{
  std::string s1(MAX_STRING_LEN / 2, '*');
  s1 += std::string(state.range(0), '-');

  std::string s2(state.range(0), '-');

  while (state.KeepRunning())
    benchmark::DoNotOptimize(s1.find(s2));
}
```

```
// Match somewhere from middle to the end.
static void
BM_StringFindMatch2(benchmark::State &state)
{
  std::string s1(MAX_STRING_LEN / 2, '*');
  s1 += std::string(state.range(0), '-');
  s1 += std::string(state.range(0), '*');

  std::string s2(state.range(0), '-');

  while (state.KeepRunning())
    benchmark::DoNotOptimize(s1.find(s2));
}
```

Missing inlining opportunities in basic_string

- Important functions not inlined.
  - basic_string::__init(const value_type* __s, size_type __sz)
  - basic_string::~basic_string()
- Clang front end does not emit the definition of these functions (extern templates) in the IR
- Solution
  - Mark functions as inline

Missing function attributes

- Missing __attribute__((__noreturn__)) in important functions.
  - Prevents important compiler optimizations
  - Results in false positives in static analysis results
- __throw.* functions in __locale, deque, future, regex, system_error, vector

Example:

```
class __vector_base_common
{
protected:
  _LIBCPP_ALWAYS_INLINE __vector_base_common() {}
  void __throw_length_error() const _LIBCPP_NORETURN_ON_EXCEPTIONS;
  void __throw_out_of_range() const _LIBCPP_NORETURN_ON_EXCEPTIONS;
};
```

Issues with number parsing in locale

- Uses std::string to store the parsed numbers
  - Results in multiple (unnecessary) calls to memset
- Uses suboptimal ‘find’ function to search for a character in a string (can be converted to traits_type::find)
- Possible characters for all kinds of numbers (octal, hex, decimal) are stored in one string
  - __atoms = “0123456789abcdefABCDEFxX+-pPiInN”
- Makes unnecessary copies of ‘__atoms’ string which are never modified

Issues with number parsing in locale

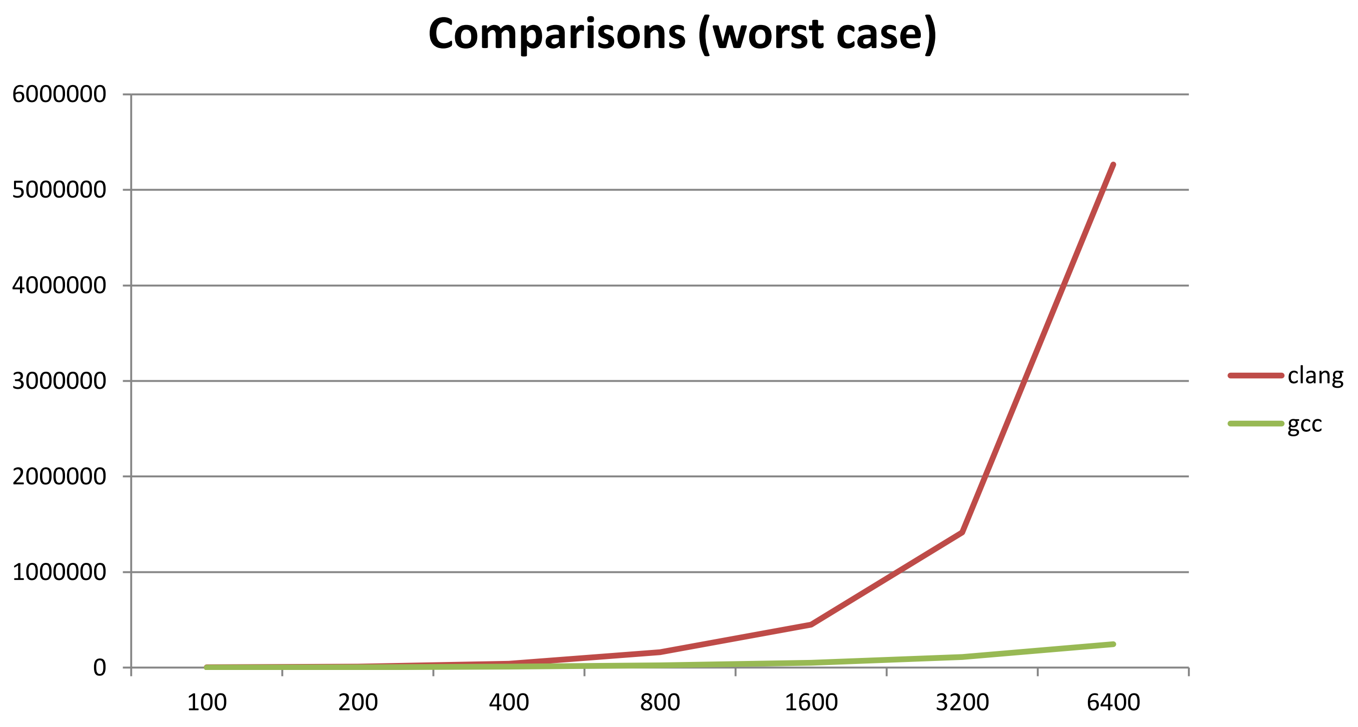
- Avoiding copy of __atoms is hard because of ABI incompatibilities.
- Current workaround is to version the change with a macro

Benchmark	Without patch	With patch	Gain
BM_Istream_numbers/32	8336 ns	7472 ns	11%

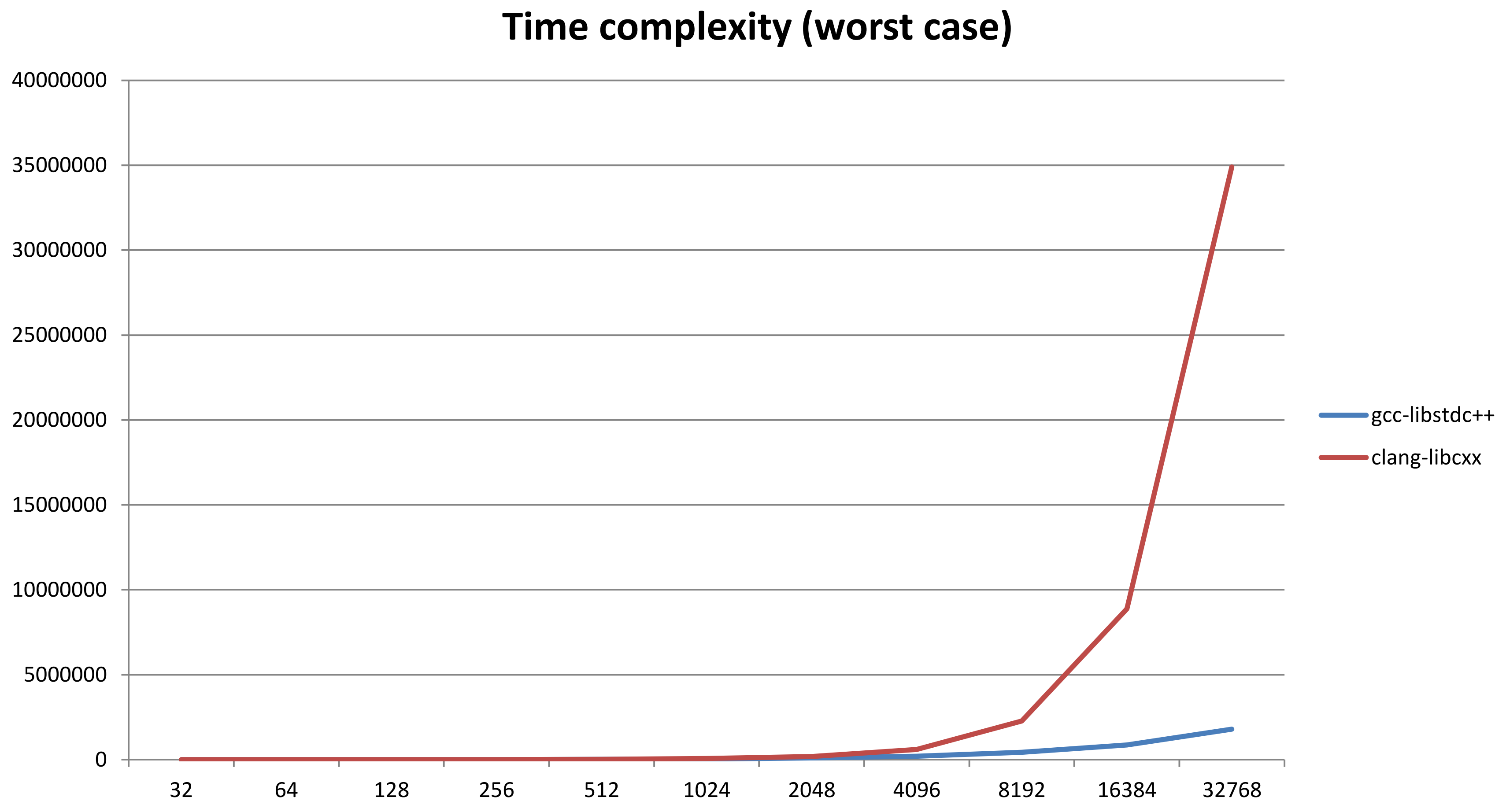
- Benchmark source:
  - std-benchmark/cxx/stringstream.bench.cpp
  - https://reviews.llvm.org/D30268

Issues with std::sort

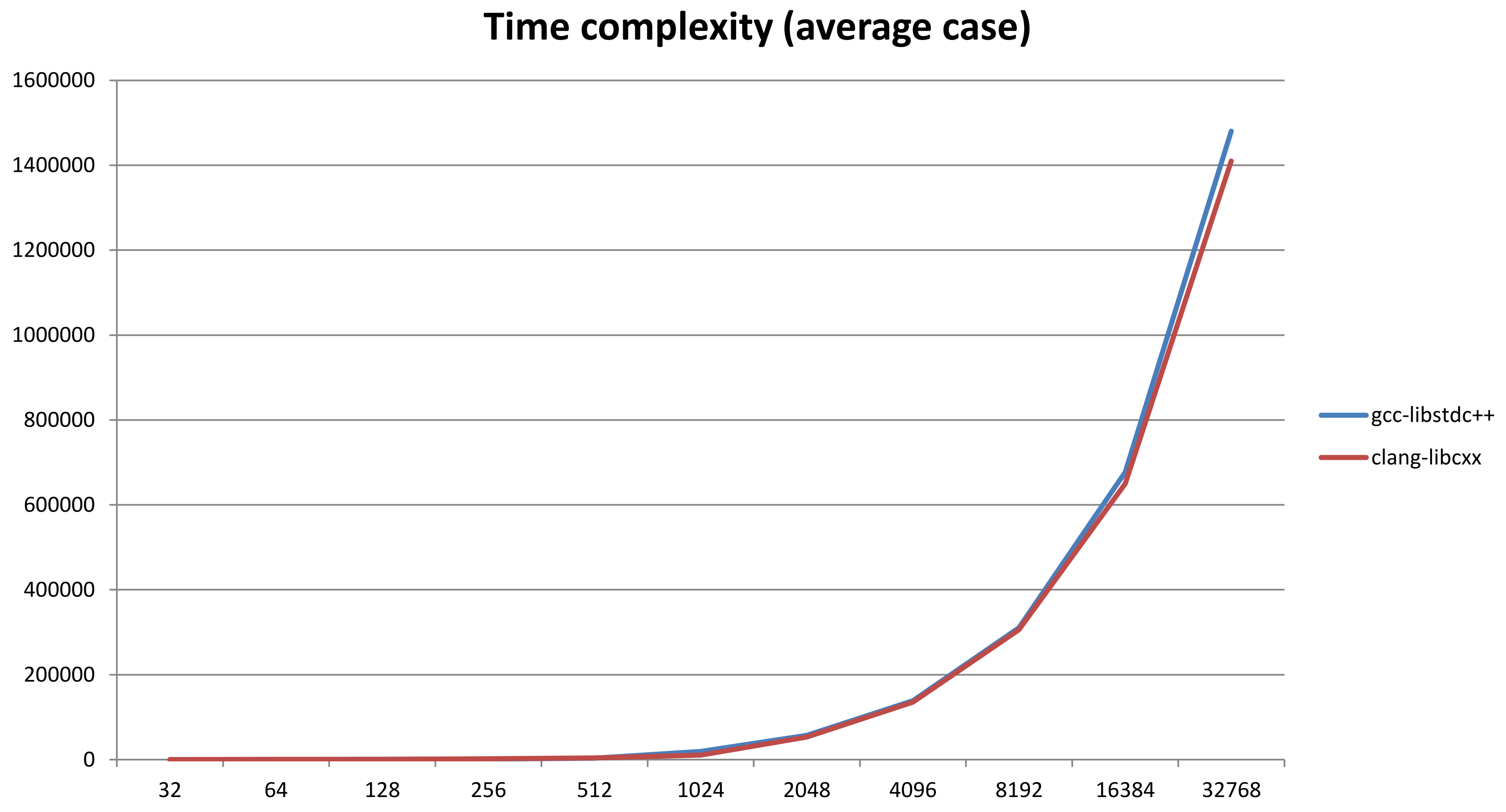
- Worst case O(N^2) comparisons against gcc-libstdc++ O(NlgN)
  - PR20837



Issues with std::sort



Issues with std::sort



std-benchmark

- https://github.com/hiraditya/std-benchmark
  - WIP
  - Builds on Linux, Windows, Mac (Thanks to cmake)
  - Performance numbers are very stable (Thanks to google-benchmark)

Lessons learned

- vector::push_back without reserve will cause a lot of allocations (~2N)
- vector::resize, string::resize initializes the memory
  - May not be what you want
- std::find may not always be the right choice
  - traits_type::find may be very efficient for string
- Rotate but not std::rotate on linked lists
- The destructor of basic_string is difficult to optimize away

Sequence containers

MSVC Data: 32KB	vector	list	deque
push_back	252,136	1,414,562	491,372
push_back_resize	253,664	1,338,775	402,114
push_back_reserve	252,729		

Libstdc++ Data: 32KB	vector	List	deque
push_back	60,567	403,278	47,859
push_back_resize	60,246	405,581	47,867
push_back_reserve	60,480		

Libc++ Data: 32KB	Vector	list	deque
push_back	89,629	537,019	65,522
push_back_resize	88,341	395,488	65,594
push_back_reserve	88,024		

std::string

Data: 32KB	no_match	all_match	match1	match2	prefix
MSVC	7673	1688	9208	8998	
libc++	474	827	966	971	
libstdc++	474	849	968	971	
c string (time) /cxx string (time)					
MSVC	0.18	1.48	0.32	0.30	0.07
libc++	2.15	1332.5	73.8	69.2	0.02
libstdc++	2.14	1292.9	80.15	75.4	0.02

No_match: no match in substring
All_match: matches in the beginning of the string
Match1: Matches at the end
Match2: Matches at the middle
Prefix: multiple prefix matches

Associative vs Hashed Associative (Inserting Random elements)

Data:32KB	Set	Map	Unordered_set	Unordered_map
MSVC	139	183	14	75
Libc++	102	100	21	21
Libstdc++	118	116	19	46

compiler vs. programmer

Data:32KB	programmer	compiler	hand-optimized (memcpy)
MSVC	11,736ns	11,808ns	1,124ns
clang++	1083ns	1082ns	1478ns
g++	1084ns	1448ns	1460ns

```
const char*
assign(const char *beg,
        const char *end, char *dest) {
 while (beg != end)
   *dest++ = *beg++;
 return beg;
}
```

```
const char*
assign_res(const char * __restrict beg,
             const char * __restrict end,
             char *__restrict dest) {
 while (beg != end)
   *dest++ = *beg++;
 return beg;
}
```

References

- https://gcc.gnu.org/onlinedocs/libstdc++/index.html
- http://clang-analyzer.llvm.org/annotations.html#attr_noreturn
- https://reviews.llvm.org/D21103
- https://reviews.llvm.org/D22782
- https://reviews.llvm.org/D22834
- https://reviews.llvm.org/D21232
- https://reviews.llvm.org/D27068
- https://github.com/google/benchmark
- https://github.com/hiraditya/std-benchmark

Caution while using std::vector

- push_back:
  - Invalidates iterators
  - Causes reallocation when enough space is not available (~2N space for N elements)
- inserting element(s) anywhere except the end will result in reallocation

Alternatives to std::vector

- If the size is known at compile time std::array may be a better choice
- If reads and writes are of the same order, std::deque is a better choice (Find the ratio of read/write to switch the container)

Caution with std::string

- Calls memset when resized
- The destructor of basic_string is difficult to optimize away
- String::find does not get inlined for g++

```
#include<string>

int main() {
  std::string s("a");
  s+='a';
  return 0;
}
```

```
g++ -O3 t.cpp -S –fno-exceptions –std=c++11 -o - | grep _ZdlPv

clang++ -O3 t.cpp -S –fno-exceptions –std=c++11 -o - | grep _ZdlPv
        call    _ZdlPv
```

TODO: MSVC

```
#include<string>
void foo();

int main() {
  std::string s("a");
  foo()
  return 0;
}
```

```
g++ -O3 t.cpp -S –fno-exceptions –std=c++11 -o - | grep _ZdlPv
        call    _ZdlPv

clang++ -O3 t.cpp -S –fno-exceptions –std=c++11 -o - | grep _ZdlPv
```

Alternatives to std::string

```
#include<string>

int main() {
  std::string s("a");
  s+='a';
  return 0;
}
```

- Alternative:
  - std::array<char> when size known at compile time

std::stable_sort

- https://bugs.llvm.org//show_bug.cgi?id=26886
  - libstdc++: greater performs half as many comparisons than less for a sorted array
  - Windows: greater performs same comparisons than less for a sorted array
  - libc++: greater performs 10 times more comparisons than less for a sorted array

Worst case time complexity vs. real world performance

time-complexity	libstdc++	libcxx
std::sort	O(nlogn)	O(n^2)
std::find	O(n)	O(n)

performance	libstdc++	libcxx
std::sort(random)	0.58NlgN	0.43NlgN
std::find		

Algorithms

- std::find may not always be the right choice
  - traits_type::find may be very efficient for string
- Rotate but not std::rotate on linked lists

Size of containers

Container	gcc	clang	MSVC
std::vector<int>()	24	24	24
std::list<int>()	24	24	16
std::deque<int>()	80	48	40
std::set<int>()	48	24	16
std::unordered_set<int>()	56	40	64
std::map<int, int>()	48	24	16
std::unordered_map<int, int>()	56	40	64

Optimize for latency

Memory	Latency (cycles)
L1	4
L2	12
L3	36
RAM	36+57ns

Intel i7-4770 3.4GHz (Turbo Boost off) 22 nm. RAM: 32 GB (PC3-12800 cl11 cr2).

Source: http://www.7-cpu.com/cpu/Haswell.html