Performance analysis of libcxx

Aditya Kumar
Sebastian Pop
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	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
  - Highly optimized string utilities are already available in glibc and can be leveraged
- 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;         // no need to check range on __m1 because __s guarantees we have enough source
          if (!__pred(*__m1, *__m2))
          {
             ++__first1;
             break;
          }
       }
   }
}
...
}
```

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;
  }
}
```

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()
- Compiler front end does not emit the definition of these functions (extern templates) to the IR
- Solutions?
  - attribute((__always_inline__))
  - Fix the compiler front end

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:

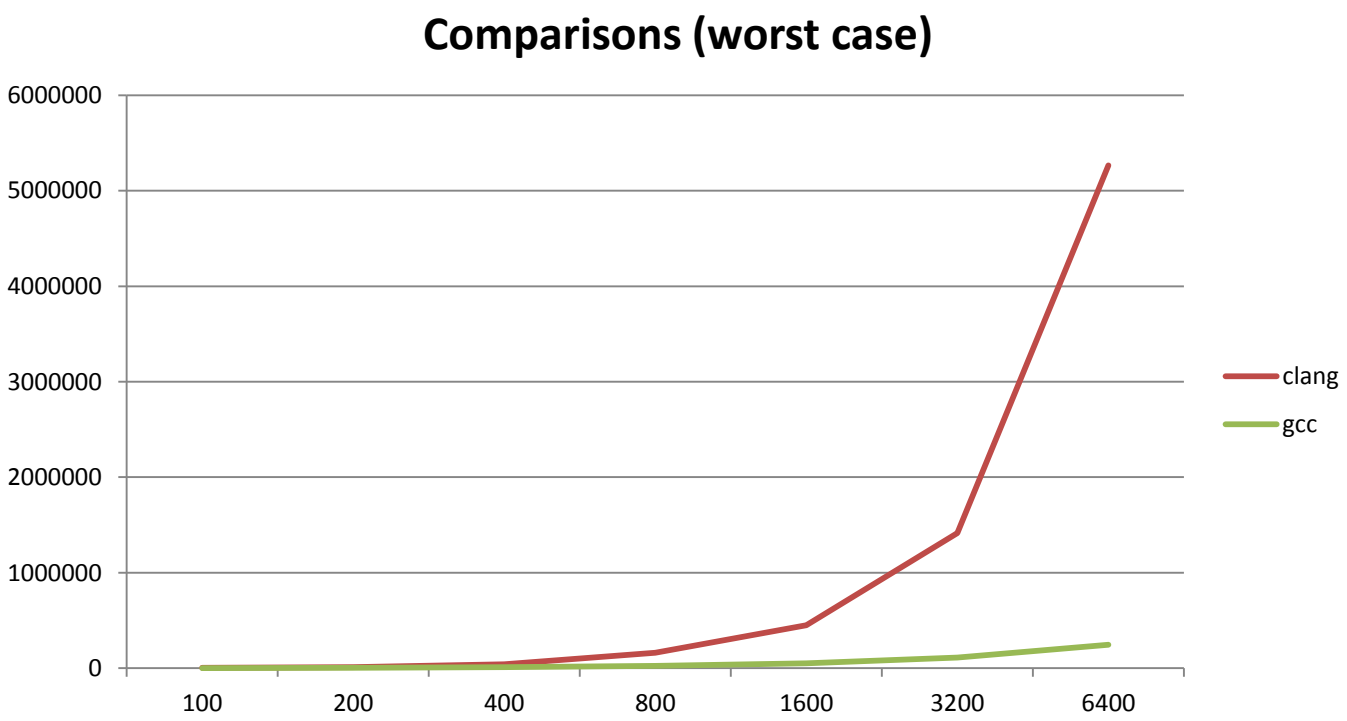
```
class __vector_base_common
{
protected:
  _LIBCPP_ALWAYS_INLINE __vector_base_common() {}
  void __throw_length_error() const _LIBCPP_NORETURN_ON_EXCEPTIONS;
  void __throw_out_of_range() const _LIBCPP_NORETURN_ON_EXCEPTIONS;
};
```

Issues with stringstream [WIP]

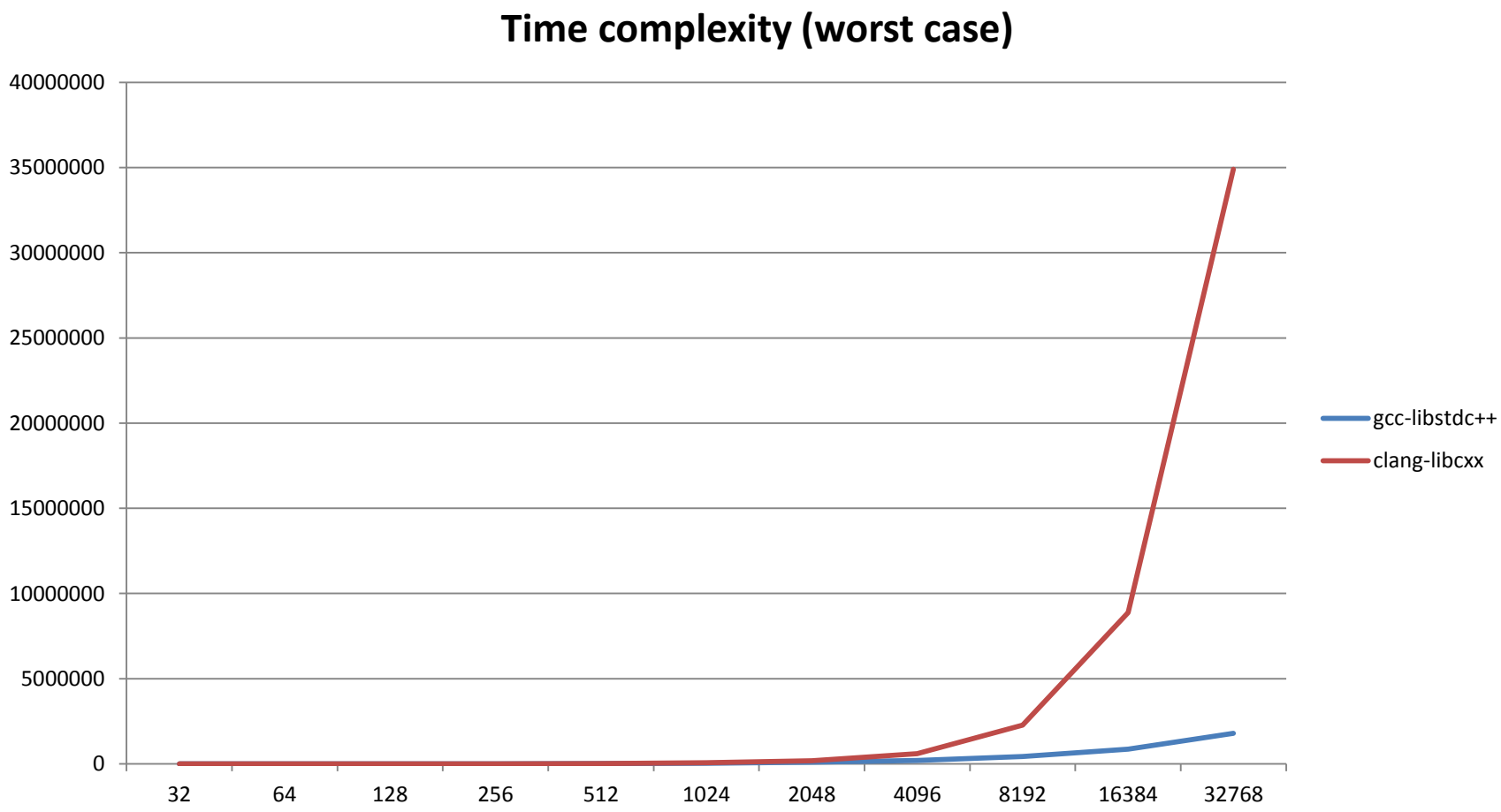
- Uses std::string to store the parsed numbers
  - Results in multiple (unnecessary) calls to memset
- Uses suboptimal ‘find’ function to search for a character in a string (can be converted to traits_type::find)
- Possible characters for all kinds of numbers are stored in one string
  - __atoms = “0123456789abcdefABCDEFxX+-pPiInN”
- Makes unnecessary copies of ‘__atoms’ string

Issues with std::sort

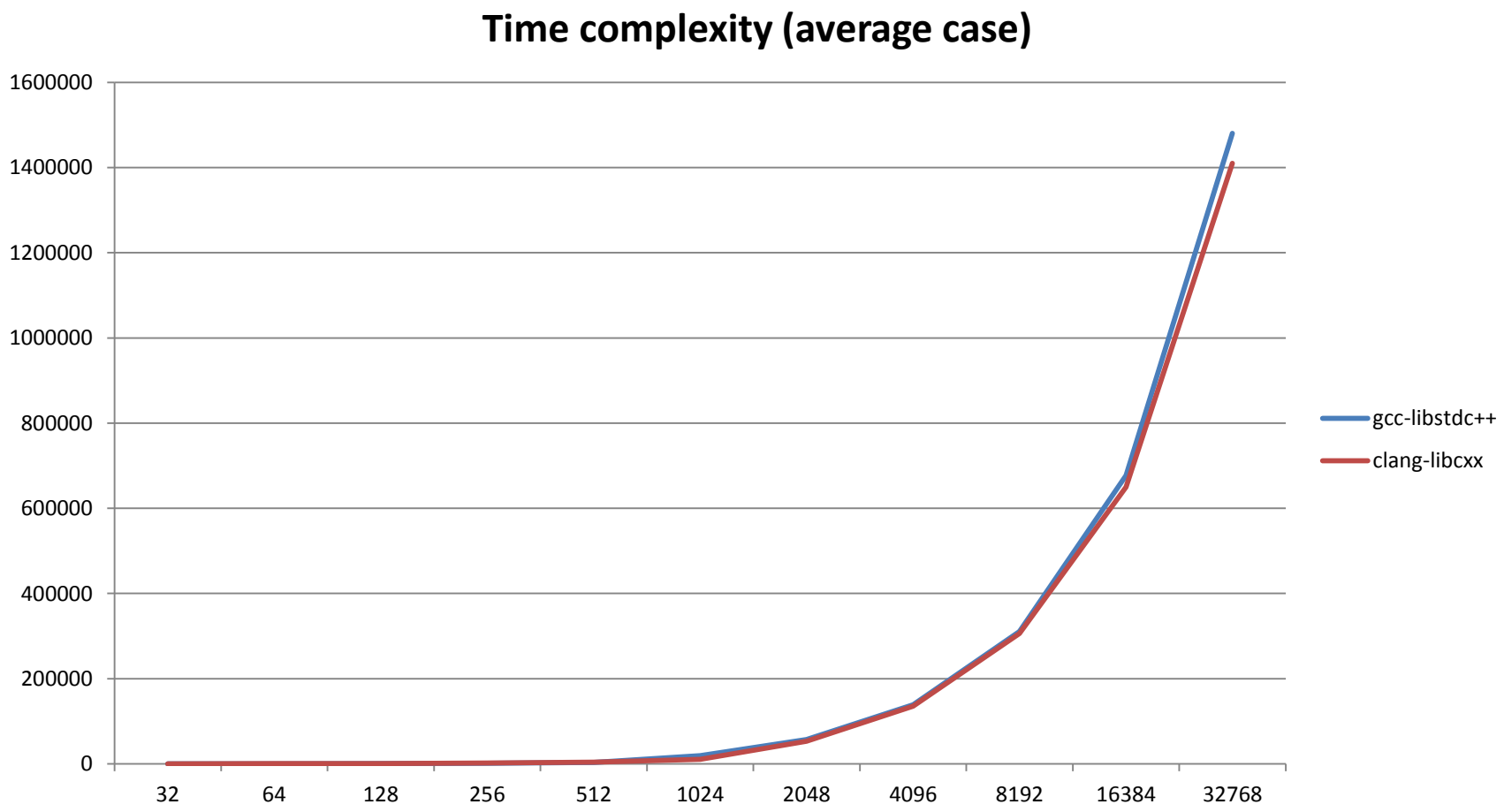
- Worst case O(N^2) comparisons against gcc-libstdc++ O(NlgN)
  - PR20837



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

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