Performance work in the libstdc++-v3 project

Paolo Carlini
SUSE
pcarlini@suse.de

Abstract

The GNU Standard C++ Library v3 is a long term project aimed at implementing a fully conforming C++ runtime library, as mandated by the ISO 14882 Standard. Whereas during the first years the focus was mostly on features, recently, after my appointment as one of its official maintainers, much more attention is devoted to performance issues and contributions in the area are particularly encouraged and appreciated. In this paper the main approaches being followed are reviewed (e.g., hand-coding, exploitation of glibc extensions, caching), together with the tools used, and a number of satisfying results obtained so far, particularly, in the iostreams and locales chapters. Quantitative comparisons on x86-linux with the Icc/Dinkumware offer will be also presented, based on code snippets provided by the new performance testsuite and distilled from actual performance PRs. In the final section, a better integration with the compiler team is argued for and emphasized.

1 Introduction

Today, *circa* 2004, the libstdc++-v3 project delivers in a typical GCC distribution more than 420000 lines of code, including 1350 regression testcases and a growing performance testsuite. Many different architectures, both 32-bit and 64-bit, are fully supported, on many different OSes, from x86 to s390x and from Linux to Darwin.

The project was started in 1998 and release after release the "degree of conformance" to the ISO C++ Standard is becoming very high, with many features implemented satisfactorily and quickly stabilizing.

Indeed, an analysis of the 3.4.0 Release Notes reveals that major changes, that first blush may seem conformance related (e.g., UTF-8 support, generic character traits) in fact should be strictly speaking categorized as QoI improvements.

On the other hand, the users are becoming rather demanding as far as performance is concerned. Among the possible causes: the good speed in some areas (e.g., I/O) of the old, pre-standard, C++ runtime library; new offers, like Icc/Dinkumware, on the market and easily available on the widespread x86-linux platform. More generally, does not seem obvious anymore that library functions that have a C library counterpart must be necessarily slower: people want a complete "object oriented" application not renouncing to performance.

Also, some new facilities offered by the ISO Standard are recently gaining larger popularity (e.g., locale) and real world applications are able to emphasize weaknesses that went unnoticed to the implementors, naturally caring more about conformance, in the first place.

The main focus of the work is therefore slowly changing and the purpose of this paper is dis-

cussing how, using which methods, and exploiting which instruments. Of course, one of its main objectives is soliciting feedback and opening a discussion on such topics. The first part presents sort-of a chronology of the most relevant recent achievements[1], spanning the last year or so: it represents also a nice occasion to thank some of the most generous contributors. Then, three items will be discussed more throughly to convey a few specific, general points. In the last part, moving from a recent episode, a better integration with the compiler team will be wished.

2 A Chronology

Necessarily, there is a good amount of "fuzziness" in this type of historical reconstruction: many important contributions went in only after a long discussion, or piecewise, during a few months. Most of the changes presented below are only in 3.4 (and mainline, of course), but, also due to the above mentioned reasons, not *all* the performance related improvements in the current release branch will be exhaustively listed.

Output of integers For GCC 3.3 Jerry Quinn rewrote from scratch the code formatting integer types for output, avoiding going through `sprintf` for performance sake: probably for the first time, the implementation interpreted non-trivially one of those typical "as if" specifications present in the Standard.

Separate synched `filebuf` In this case, it could be said that a speed gain has been obtained as a (very welcome) by product: Pétur Runólfsson separate synched filebuf improved remarkably the conformance of the library in the interactions with C stdio (e.g., cin/stdin). Anyway, as a matter of fact, `cout.rdbuf()->sputc('a')` became for instance about three times faster.

`Numpunct` cache After some initial attempts during GCC 3.3 lifetime, finally GCC 3.4 exploits caching for formatted I/O: this important issue will be discussed in detail below. In any case, formatted output of integer types is now three time faster than in GCC 3.2.3 (Table 1).

GCC 3.2.3	14.590u 0.010s 0:14.67 99.5%
GCC 3.3.3	4.780u 0.010s 0:04.80 99.7%
GCC 3.4.0	4.160u 0.010s 0:04.19 99.5%
Icc8.0	10.430u 0.020s 0:10.48 99.7%

Table 1: Output of int`s` from 0 to 9999999 to `/dev/null`

Empty `string` speedup At the beginning of 2003, Nathan Myers, the original author of v3 `basic_string` class, noticed that the multi-processor bus contention can be reduced by comparing addresses first, and never touching the reference count of the empty object. The final patch has been committed in time for 3.4 and improves remarkably the performance on single-processor systems too: Table 2 presents satisfying timings for a simple snippet shown in Figure 1.

```
for (int i = 0; i < 2000; ++i)
  std::string a[100000];
```

Figure 1: Creating and immediately destroying lots of `string` objects

Non-unified `filebuf` According to the C++ Standard, a `seek` is needed in order to switch from read mode to write mode (and

[1] If not otherwise indicated, all the timings are relative to a P4-2400 machine, linux2.4, glibc-cvs, -O2, Icc8.0 Build 20040412Z.

GCC 3.3.3	20.890u 0.020s 0:21.01 99.5%
GCC 3.4.0	0.790u 0.000s 0:00.79 100.0%
Icc8.0	17.200u 0.030s 0:17.33 99.4%

Table 2: Execution times for the code displayed in Figure 1

vice versa) during I/O. This is a very reasonable requirement, by the way inherited from the C Standard. However, the old implementation, as a (very puzzling) QoI feature, had relaxed it: unfortunately, the upshot was that the get area and put area pointers had always to be updated in a lockstep way. Figure 2 compares GCC 3.3 and GCC 3.4 code for `sputc`: the former called `_M_out_cur_move` instead of simply bumping the put area pointer by way of `pbump`. Additionally, the `_M_out_buf_size` helper was also needed: as a result the function was not amenable to inlining anymore. The same happened of course for `sbumpc` and elsewhere. The performance suffered consequently as Table 3 demonstrates.

GCC 3.3.3	42.440u 0.290s 0:42.91 99.5%
GCC 3.4.0	4.080u 0.300s 0:04.39 99.7%
Icc8.0	11.080u 0.360s 0:11.45 99.9%
'C' (unlocked)	6.590u 0.280s 0:06.90 99.5%

Table 3: Char-by-char copy of 1 GB from `/dev/zero` to `/dev/null`

Fixing this required consistent, invasive changes to `streambuf`, and `stringbuf` but eventually enabled a much simpler maintenance and paved the way to the UTF-8 support.

Input of integers The code parsing integers could be improved rather easily, thanks to the `numpunct` caching mechanism already in place and functioning well. Interestingly, though, in this area the library sports some design choices not shared by other implementations (whereas consistent with the letter of the standard!), to be discussed below.

Table-based `ctype` In order to obtain fast `time_get` and `time_put` facets (not suited for caching, due to their special requirements), and also for free standing use, `ctype` functions, such as `narrow`, `widen`, and `is`, are now table-based. Thanks to a sophisticated solution devised by Jerry and refined on the discussion list, for `char` type it is even avoided the virtual function call cost. The improvement is more visible for `wchar_t`, however: once more, close to an order of magnitude with respect to the previous generation.

`Codecvt` rewrite During GCC 3.4 Stage 1 Pétur rewrote the `codecvt` facet, obtaining a very good support of encoding-zero (e.g., UTF-8) locales too. In the process, he provided a rather complete set of testcases. Finally, as will be discussed in the second part, performance has been also improved, thus delivering for the first time both correct and efficient support for a wide set of locales.

Other `string` improvements In Item 29 of his latest book, *Effective STL*, Scott Meyers proposes an elegant idiom for copying a text file into a `string` object (Figure 3).

```
string Data(istreambuf_iterator<char>(File),
            istreambuf_iterator<char>());
```

Figure 3: Istreambuf_iterators usage

In order for this proposal to be effective, the constructor from a pair of `input_iterator`*s* must be efficient: a satisfactorily fix involved redesigning the latter to exploit a centralized growth fa-

cility, previously not available at construction time. Only in 3.4.1-pre is present another unrelated improvement, very simple but appreciable in almost every use of the `string` class[2]. It consists in special-casing single char changes to avoid the general `traits::copy` and `traits::assign`, which end up calling C library functions (Table 4).

GCC 3.3.3	1.150u 0.040s 0:01.19 100.0%
GCC 3.4.0	0.670u 0.070s 0:00.74 100.0%
GCC 3.4.1 pre	0.220u 0.060s 0:00.28 100.0%
V2	0.710u 0.030s 0:00.74 100.0%

Table 4: Ten millions of `str.append(1, 'x')`

Monetary facets Extending the `numpunct` caching work to `moneypunct` turned out to be easy. However, in the process, a few bugs and other opportunities for performance surfaced. Some are certainly straightforward (e.g., reordering operations on `string` objects to avoid reallocations), but, nevertheless, the overall effect is quite noticeable. For instance, Table 5 shows the time it takes to read one million of times a big monetary amount, i.e., 100,000,000,000.00, from an `istringstream` into a long double.

GCC 3.3.3	10.610u 0.020s 0:10.69 99.4%
GCC 3.4.0	4.110u 0.000s 0:04.12 99.7%
GCC 3.4.1 pre	2.910u 0.010s 0:02.93 99.6%
Icc8.0	3.280u 0.000s 0:03.29 99.6%

Table 5: A simple `money_get` benchmark

The difference between GCC 3.4.0 and 3.4.1-pre is entirely due to the just-mentioned simple tweak to the `string` class: a similar effect can be measured in the formatted input of floating point types, much more used today.

`Locale` functions Probably, a large number of applications doesn't have these functions as a performance bottleneck. On the other hand, the way names were processed used to be rather dumb, due to the encoding adopted for "simple" named locales—that is, roughly, having all the categories named the same, say `de_DE`. As pointed out by library-friend Martin Sebor, most probably the sections of the standard having to do with combining named locales (22.1.1.2, 22.1.1.3) will be amended: therefore the real challenge was designing a new encoding ready for the most likely future changes. Table 6 shows the time needed to compare ten millions of times via `operator==` two "simple" locales.

GCC 3.3.3	13.410u 0.000s 0:13.45 99.7%
GCC 3.4.0	11.640u 0.000s 0:11.67 99.7%
GCC 3.5.0 exp	0.220u 0.000s 0:00.22 99.9%
Icc8.0	0.850u 0.000s 0:00.85 99.9%

Table 6: A simple `locale::operator==` benchmark

`Getline` speedups A wide ranging debate ensued to the submission of PR 15002, with the participation of Matt Austern, among others. Both the `getline`s had to be improved: the member taking a `char_type*` and a `streamsize` and the function taking an `istream` and a `string`. An elegant solution, devised by Pétur, could be adopted only for the former, since it exploits *protected* `streambuf` members. It became clear that, ideally, we should have two different versions of those functions: the fast version, which takes advantage of friendship

[2] Internally to the library too, as will be quantified in the next item.

and only works for `char` and `wchar_t`, and a slow version that goes through the public interface. For the moment, profiling revealed that a large speedup could be achieved by appending to the `string` object a chunk of each line at a time (say, 128 chars), instead of one char at a time. Table 7 shows timings for reading 600000 lines, each 200 characters, from file, via `getline`.

char_type*	
GCC 3.3.3	1.700u 0.090s 0:01.80 99.6%
GCC 3.4.0	1.230u 0.070s 0:01.30 100.0%
GCC 3.4.1 pre	0.180u 0.130s 0:00.30 103.3%
Icc8.0	1.410u 0.090s 0:01.50 100.0%
`string`	
GCC 3.3.3	15.560u 0.070s 0:15.69 99.6%
GCC 3.4.0	9.030u 0.160s 0:09.22 99.6%
GCC 3.4.1 pre	1.090u 0.110s 0:01.21 99.1%
Icc8.0	1.910u 0.120s 0:02.04 99.5%

Table 7: `Getline` benchmarks

3 Telling Stories

3.1 Parsing of integer types and caching

Back in February, in the occasion of some changes to the monetary facets that were supposed to be completely uncontroversial, a long exchange started on the discussion list about the correct way to parse monetary (and numeric) quantities.

In particular, it became evident to everyone that libstdc++-v3 is probably *alone* in closely following the letter of 22.2.2.1.2. Most, if not all, the other implementations are not using `widen` and are not matching characters as prescribed in p8: instead, in order to compute the value of each specific digit `d` something equivalent to `c = narrow(*beg, '*')` is first computed, then `d` is given by `c - '0'`.

Indeed, this approach has its own virtues: there is no need for caching (and the related complexities[3]) and an efficient table-based `narrow` is sufficient alone to obtain good performance; moreover, this approach solves elegantly an issue in the Standard with the "mysterious" `find` function mentioned in p8.

In fact, if the function is interpreted (rather naturally) as `traits::find`, the serious problem ensues that any `charT`, other than plain `char` and `wchar_t`, needs an appropriate `traits<charT>::find` to be available: the Standard nowhere requires this, still clearly mandates in Table 52 to make it possible to instantiate `num_get` on *any* `charT` type[4].

Interestingly, those issues are of course well known to the LWG members, but often do *not* correspond to detailed and well debated DRs.[5]

Anyway, GCC 3.4 provides for the first time a generic `traits` class, which includes indeed a generic `traits<charT>::find`: this leads to a complete solution characterized by an excellent performance/conformance balance. Table 8 below compares the timings for reading from file ten millions of integers, from 0 to 9999999.

GCC 3.3.3	41.180u 0.020s 0:41.37 99.5%
GCC 3.4.0	5.740u 0.030s 0:05.79 99.6%
Icc8.0	14.220u 0.120s 0:14.41 99.5%
V2	5.930u 0.060s 0:06.00 99.8%
Hammer	18.660u 0.040s 0:18.78 99.5%

Table 8: A simple `num_get` benchmark

For 3.4, integer types parsing has been rewrit-

[3] Only 3.4 finally managed to have it reliably working, fast, and... not leaking memory!

[4] A POD type.

[5] Only DR 303 [WP] and DR 427 [Open] are relevant and both the resolution of the former and the comment added in Kona to the latter are clearly *against* preferring `narrow` to `widen`.

ten, avoiding `strtol`, `strtoll`, and the other C library functions previously used, instead directly accumulating the result during the parsing. Therefore, no `string` objects are involved. The hammer-branch entry is also present in the Table in order to quantify what could be otherwise achieved within the constraints of the 3.3 ABI, basically, by improving the use of the `string`s.

On the current code base, `gprof` reports that about 58% of the total time is spent in the parsing loop itself: not much can be done about this, except, perhaps, avoiding an integer division, in principle not necessary. `Memchr`, called by `traits<char>::find`, is the second topmost entry, with about 26%: in the future, a small ABI change could make possible detecting in advance the occurrence of trivial `widen`s, very common indeed, then simply using `d = *beg - widen('0')` in such cases: `traits::find` would not be necessary at all and the QoI would be further improved. All the other entries are below 5% and `__use_cache::operator()` is below 1%, a reassuring check.

In any case, barring unexpected strong requests from the users, much more effort is planned in the area of parsing and formatting of *floating point* types, which probably could be made about two times faster, but this is another story...

3.2 `Codecvt` rewrite

As already mentioned, a few months ago became evident that the performance of the most important `codecvt` functions, such as `in`, `out`, and `length`, was not satisfying: that represented a major roadblock in the way of efficient encoded I/O, otherwise made finally possible by the redesigned `filebuf` virtuals.

By the way, this problem was unfortunately noticed only *months* after the `codecvt` rewrite, a sad episode less likely to happen nowadays thanks to the new performance testsuite[6], which currently includes 30 testcases and is quickly growing: most of the tests are distilled from performance PRs.

Luckily, after some preliminary attempts, only partially successful, the real fix became obvious: it involved exploiting `mbsnrtowcs` and `wcsnrtombs`, two glibc extensions that take an extra parameter with respect to the standard `mbsrtowcs` and `wcsrtombs`. Indeed, admittedly, in GCC 3.3 `codecvt` was almost broken but already fast, thanks to the use of the latter functions. Table 9 is relative to the conversion of 400000 buffers, 1024 characters each, in the C locale.

GCC 3.3.3	1.520u 0.000s 0:01.52 100.0%
GCC 3.4.0	1.650u 0.000s 0:01.65 100.0%
Icc8.0	41.670u 0.010s 0:41.85 99.5%

Table 9: `Codecvt::in` benchmark

The small difference between GCC 3.3.3 and GCC 3.4.0 is entirely due to the additional call of `memchr` (or `wmemchr`), which is used for splitting the input (the output, respectively) in chunks, ending in '\0' (or L'\0', respectively): each one is then processed by `mbsnrtowcs` (`wcsnrtombs`, respectively).

The numbers obtained with Icc8.0 are typical of an implementation using for correctness the *single* char C library functions `wcrtomb` and `mbrtowc`: this is still happening for libstdc++-v3 too in the so-called "generic" locale model, which doesn't have the GNU extensions available. Discussing `codecvt` is therefore also an opportunity to clarify that the QoI provided by the library in that model is sometimes lower than in the GNU model. Improving this situation is feasible but requires

[6] Established June, 2003.

more help from people on platforms not based on glibc, hereby strongly solicited!

4 The Weird Loop, Outlook

An interesting feature of the C++ `cmath` and `complex` facilities is the presence of additional `pow` overloads for *integer* exponent, not present in the C Standard, that, in principle at least, enable a wide range of additional opportunities for optimization. The library implements those overloads using a function that computes the power via the well known "Russian peasant algorithm" (Figure 4) which requires only `O(log n)` multiplications.

```
template<typename _Tp>
  inline _Tp
  __cmath_power(_Tp __x, unsigned int __n)
  {
    _Tp __y = __n % 2 ? __x : 1;
    while (__n >>= 1)
      {
        __x = __x * __x;
        if (__n % 2)
          __y = __y * __x;
      }
    return __y;
  }
```

Figure 4: Helper function used by `pow`

As evident from the actual code, the loop is very simple but nonetheless characterized by a *non-linear* induction variable, not handled until a few months ago neither by the old unroller nor by the new one, present in the lno-branch and actively developed by Zdenek Dvorak and others.

"Officially" Zdenek considered non-linear IVs rare and low priority[7], but actually he was just looking for an interesting example of application, nothing more! In a matter of *few* weeks a complete framework for canonical IVs creation was ready and beautifully effective: in it, loops such as the above can be fully unrolled in case of a constant `__n` thus leading to just *perfect* assembly.

Besides the technical details of the episode—who knows, perhaps by the time the lno-branch is merged the library will not use the very same algorithm—its lesson seems definitely an invitation to more frequent and strict exchanges between the library and compiler people.

Acknowledgments

Many thanks go to SUSE for the enthusiastic support of my work; to Benjamin Kosnik, who trusted and encouraged me back in 2001 (and still does!); to Nathan "Less is More" Myers, a constant source of inspiration; to all my Italian friends, especially a little smiling hamster (")

[7] See the audit trail of PR 11710.

```
_M_out_buf_size()
{
  off_type __ret = 0;
  if (_M_out_cur)
    if (_M_out_beg == _M_buf)
      __ret = (_M_out_beg + _M_buf_size
               - _M_out_cur)
    else
      __ret = _M_out_end - _M_out_cur;
  return __ret;
}

_M_out_cur_move(off_type __n)
{
  bool __testin = _M_in_cur;
  _M_out_cur += __n;
  if (__testin && _M_buf_unified)
    _M_in_cur += __n;
  if (_M_out_cur > _M_out_end)
    {
      _M_out_end = _M_out_cur;
      if (__testin)
        _M_in_end += __n;
    }
}

sputc(char_type __c)
{
  int_type __ret;
  if (_M_out_buf_size())
    {
      *_M_out_cur = __c;
      _M_out_cur_move(1);
      __ret = traits_type::to_int_type(__c);
    }
  else
    __ret = this->overflow(
            traits_type::to_int_type(__c));
  return __ret;
}
```

(a) GCC 3.3

```
sputc(char_type __c)
{
  int_type __ret;
  if (this->pptr() < this->epptr())
    {
      *this->pptr() = __c;
      this->pbump(1);
      __ret = traits_type::to_int_type(__c);
    }
  else
    __ret = this->overflow(
            traits_type::to_int_type(__c));
  return __ret;
}
```

(b) GCC 3.4

Figure 2: GCC 3.3 (a) vs GCC 3.4 (b) code for `sputc` (slightly simplified)