std::to_string
faster than light

Афанасьев Иван

cpp sberia 2020

Intro

- *std::to_string / std::to_wstring*
- integers only
- libc++
- https://reviews.llvm.org/D59178

std::to_string

Defined in header <string>

`std::string to_string( int value );`	(1)	(since C++11)
`std::string to_string( long value );`	(2)	(since C++11)
`std::string to_string( long long value );`	(3)	(since C++11)
`std::string to_string( unsigned value );`	(4)	(since C++11)
`std::string to_string( unsigned long value );`	(5)	(since C++11)
`std::string to_string( unsigned long long value );`	(6)	(since C++11)
`std::string to_string( float value );`	(7)	(since C++11)
`std::string to_string( double value );`	(8)	(since C++11)
`std::string to_string( long double value );`	(9)	(since C++11)

Converts a numeric value to `std::string`.

Intro

- *std::to_string / std::to_wstring*
- integers only
- libc++
- https://reviews.llvm.org/D59178

std::to_string

Defined in header <string>

```
std::string to_string( int value );                (1)  (since C++11)
std::string to_string( long value );               (2)  (since C++11)
std::string to_string( long long value );          (3)  (since C++11)
std::string to_string( unsigned value );           (4)  (since C++11)
std::string to_string( unsigned long value );      (5)  (since C++11)
std::string to_string( unsigned long long value ); (6)  (since C++11)
std::string to_string( float value );              (7)  (since C++11)
std::string to_string( double value );             (8)  (since C++11)
std::string to_string( long double value );        (9)  (since C++11)
```

Converts a numeric value to `std::string`.

1) Converts a signed integer to a string with the same content as what `std::sprintf(buf, "%d", value)` would produce for sufficiently large `buf`.
2) Converts a signed integer to a string with the same content as what `std::sprintf(buf, "%ld", value)` would produce for sufficiently large `buf`.

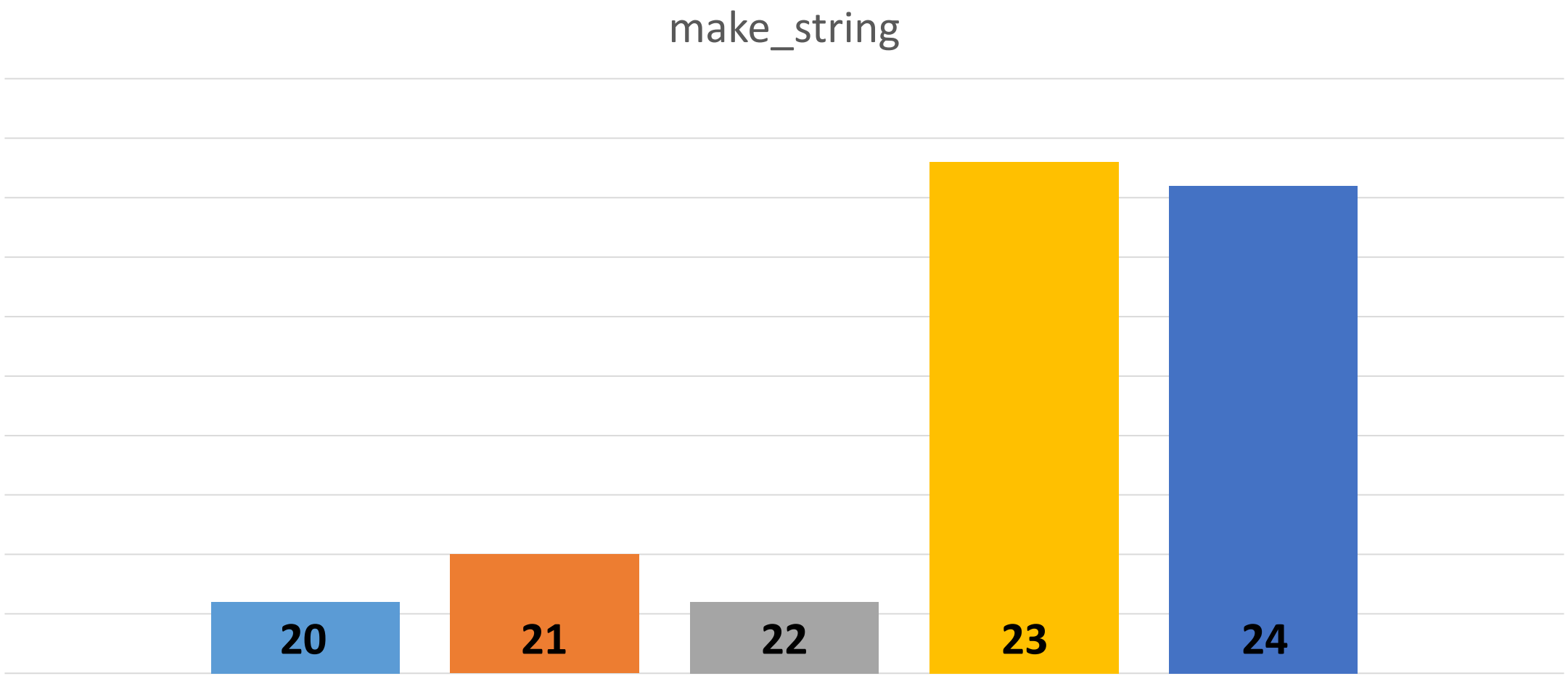
SSO

SSO

```
std::string make_string_20() { return "12345678901234567890"; }
std::string make_string_21() { return "123456789012345678901"; }
std::string make_string_22() { return "1234567890123456789012"; }
std::string make_string_23() { return "12345678901234567890123"; }
std::string make_string_24() { return "123456789012345678901234"; }
```

clang-9.0, 64bit, libc++, default layout

SSO



SSO

```
class string
{
  char* data;  // 8 byte
  size_t len;  // 8 byte
  size_t cap;  // 8 byte
};
```

clang-9.0, 64bit, libc++, default layout

SSO

```
class string
{
  char* data;  // 8 byte
  size_t len;  // 8 byte
  size_t cap;  // 8 byte
};
// SSO: 23 + 1 bytes
```

clang-9.0, 64bit, libc++, default layout

SSO

```
class string
{
  char* data;  // 8 byte
  size_t len;  // 8 byte
  size_t cap;  // 8 byte
};
// SSO: 23 + 1 bytes
```

Actually sizeof(std::string) == 32 …

clang-9.0, 64bit, libc++, default layout

copy elision / RVO / NRVO.

Antiquity

https://en.cppreference.com/w/cpp/language/copy_elision

copy elision / RVO / NRVO. Antique times

```
House build_house() {
  House house;
  house.add_roof(make_roof());
  return house;
}
```

https://en.cppreference.com/w/cpp/language/copy_elision

copy elision / RVO / NRVO. Antique times

RVO

```
House build_house() {
  return House(make_roof());
}
```

https://en.cppreference.com/w/cpp/language/copy_elision

copy elision / RVO / NRVO. Antique times

RVO

```
House build_house() {
  return House(make_roof());
}
```

NRVO

```
House build_house() {
  House house;
  house.add_roof(make_roof());
  return house;
}
```

https://en.cppreference.com/w/cpp/language/copy_elision

copy elision / RVO / NRVO. Antique times

NRVO fails if:

https://en.cppreference.com/w/cpp/language/copy_elision

copy elision / RVO / NRVO. Antique times

NRVO fails if:

- no single automatic storage duration object to return

```
House build_house(int id) {
  House stone_house;
  stone_house.add_roof(make_stone_roof());

  House brick_house;
  brick_house.add_roof(make_brick_roof());

  return is_stone_house(id) ? stone_house : brick_house;
}
```

https://en.cppreference.com/w/cpp/language/copy_elision

copy elision / RVO / NRVO. Antique times

NRVO fails if:

- no single automatic storage duration object to return
- return from function parameter

```
House build_house(House house) {
  house.add_roof(make_roof());
  return house;
}
```

https://en.cppreference.com/w/cpp/language/copy_elision

copy elision / RVO / NRVO. Antique times

NRVO fails if:

- no single automatic storage duration object to return
- return from function parameter
- another return type

```
House build_house() {
  DerivedHouse house;
  house.add_roof(make_());
  return house;
}
```

https://en.cppreference.com/w/cpp/language/copy_elision

copy elision / RVO / NRVO.

since C++17

https://en.cppreference.com/w/cpp/language/copy_elision

copy elision / RVO / NRVO. Since C++17

- “RVO is mandatory”

```
House build_house() {
  return House(make_roof());
}
```

```
House build_house(bool stone) {
  if (stone)
    return House(make_stone_roof());
  else
    return House(make_brick_roof());
}
```

https://en.cppreference.com/w/cpp/language/copy_elision

copy elision / RVO / NRVO. Since C++17

- “RVO is mandatory”
- “unmaterialized value passing”

```
T f() {
  return T();
}

// only one call to default constructor of T, to initialize x
T x = T(T(f()));
```

https://en.cppreference.com/w/cpp/language/copy_elision

std::to_string reference implementation

std::to_string reference implementation

```
string to_string(int val)
{ return as_string(snprintf, initial_string<string, int>()(),  "%d", val); }

string to_string(long val)
{ return as_string(snprintf, initial_string<string, long>()(), "%ld", val); }

...

wstring to_wstring(int val)
{ return as_string(get_swprintf(), initial_string<wstring, int>()(),  L"%d", val); }

wstring to_wstring(long val)
{ return as_string(get_swprintf(), initial_string<wstring, long>()(), L"%ld", val); }

...
```

std::to_string reference implementation

```
string to_string(int val)
{ return as_string(snprintf, initial_string<string, int>()(),  "%d", val); }

string to_string(long val)
{ return as_string(snprintf, initial_string<string, long>()(), "%ld", val); }

...

wstring to_wstring(int val)
{ return as_string(get_swprintf(), initial_string<wstring, int>()(),  L"%d", val); }

wstring to_wstring(long val)
{ return as_string(get_swprintf(), initial_string<wstring, long>()(), L"%ld", val); }

...
```

std::to_string reference implementation

```
string to_string(int val)
{ return as_string(snprintf, initial_string<string, int>()(),  "%d", val); }

string to_string(long val)
{ return as_string(snprintf, initial_string<string, long>()(), "%ld", val); }

...

wstring to_wstring(int val)
{ return as_string(get_swprintf(), initial_string<wstring, int>()(),  L"%d", val); }

wstring to_wstring(long val)
{ return as_string(get_swprintf(), initial_string<wstring, long>()(), L"%ld", val); }

...
```

std::to_string reference implementation

```
string to_string(int val)
{ return as_string(snprintf, initial_string<string, int>()(),  "%d", val); }

string to_string(long val)
{ return as_string(snprintf, initial_string<string, long>()(), "%ld", val); }

...

wstring to_wstring(int val)
{ return as_string(get_swprintf(), initial_string<wstring, int>()(),  L"%d", val); }

wstring to_wstring(long val)
{ return as_string(get_swprintf(), initial_string<wstring, long>()(), L"%ld", val); }

...
```

std::to_string reference implementation

```
string to_string(int val)
{ return as_string(snprintf, initial_string<string, int>()(),  "%d", val); }

string to_string(long val)
{ return as_string(snprintf, initial_string<string, long>()(), "%ld", val); }

...

wstring to_wstring(int val)
{ return as_string(get_swprintf(), initial_string<wstring, int>()(),  L"%d", val); }

wstring to_wstring(long val)
{ return as_string(get_swprintf(), initial_string<wstring, long>()(), L"%ld", val); }

...
```

std::to_string reference implementation

```
string to_string(int val)
{ return as_string(snprintf, initial_string<string, int>()(),  "%d", val); }

string to_string(long val)
{ return as_string(snprintf, initial_string<string, long>()(), "%ld", val); }

...

wstring to_wstring(int val)
{ return as_string(get_swprintf(), initial_string<wstring, int>()(),  L"%d", val); }

wstring to_wstring(long val)
{ return as_string(get_swprintf(), initial_string<wstring, long>()(), L"%ld", val); }

...
```

std::to_string reference implementation

```
template <class S, class V>
struct initial_string;
```

std::to_string reference implementation

```
template <class S, class V>
struct initial_string;
```

std::to_string reference implementation

```
template <class S, class V>
struct initial_string;
```

std::to_string reference implementation

```
template <class S, class V>
struct initial_string;


template <class V>
struct initial_string<string, V>
{
  string operator()() const
  {
    string s;
    s.resize(s.capacity());
    return s;
  }
};
```

std::to_string reference implementation

```
template <class S, class V>
struct initial_string;


template <class V>
struct initial_string<string, V>
{
  string operator()() const
  {
    string s;
    s.resize(s.capacity());
    return s;
  }
};
```

std::to_string reference implementation

```
template <class S, class V>
struct initial_string;


template <class V>
struct initial_string<string, V>
{
  string operator()() const
  {
    string s;
    s.resize(s.capacity());
    return s;
  }
};
```

std::to_string reference implementation

```
template <class S, class V>
struct initial_string;
```

```
template <class V>
struct initial_string<string, V>
{
  string operator()() const
  {
    string s;
    s.resize(s.capacity());
    return s;
  }
};
```

```
template <class V>
struct initial_string<wstring, V>
{
  wstring operator()() const
  {
    wstring s(23, wchar_t()); // 23 for ull
    s.resize(s.capacity());
    return s;
  }
};
```

std::to_string reference implementation

```
template <class S, class V>
struct initial_string;
```

```
template <class V>
struct initial_string<string, V>
{
  string operator()() const
  {
    string s;
    s.resize(s.capacity());
    return s;
  }
};
```

```
template <class V>
struct initial_string<wstring, V>
{
  wstring operator()() const
  {
    wstring s(23, wchar_t()); // 23 for ull
    s.resize(s.capacity());
    return s;
  }
};
```

std::to_string reference implementation

```
template <class S, class V>
struct initial_string;
```

```
template <class V>
struct initial_string<string, V>
{
  string operator()() const
  {
    string s;
    s.resize(s.capacity());
    return s;
  }
};
```

```
template <class V>
struct initial_string<wstring, V>
{
  wstring operator()() const
  {
    wstring s(23, wchar_t()); // 23 for ull
    s.resize(s.capacity());
    return s;
  }
};
```

std::to_string reference implementation

```
template<typename S, typename P, typename V>
inline S as_string(P sprintf_like, S s, const typename S::value_type* fmt, V a) {
  size_t available = s.size();
  while (true) {
    int status = sprintf_like(&s[0], available + 1, fmt, a);
    if (status >= 0) {
      size_t used = static_cast<size_t>(status);
      if (used <= available) {
        s.resize(used);  // success: fit size
        break;           // success: return
      }
      available = used; // assume this is advice of how much space we need.
    } else {
      available = available * 2 + 1;
    }
    s.resize(available);
  }
  return s;
}
```

std::to_string reference implementation

```
template<typename S, typename P, typename V>
inline S as_string(P sprintf_like, S s, const typename S::value_type* fmt, V a) {
  size_t available = s.size();
  while (true) {
    int status = sprintf_like(&s[0], available + 1, fmt, a);
    if (status >= 0) {
      size_t used = static_cast<size_t>(status);
      if (used <= available) {
        s.resize(used);  // success: fit size
        break;           // success: return
      }
      available = used; // assume this is advice of how much space we need.
    } else {
      available = available * 2 + 1;
    }
    s.resize(available);
  }
  return s;
}
```

std::to_string reference implementation

```
template<typename S, typename P, typename V>
inline S as_string(P sprintf_like, S s, const typename S::value_type* fmt, V a) {
  size_t available = s.size();
  while (true) {
    int status = sprintf_like(&s[0], available + 1, fmt, a);
    if (status >= 0) {
      size_t used = static_cast<size_t>(status);
      if (used <= available) {
        s.resize(used);  // success: fit size
        break;           // success: return
      }
      available = used; // assume this is advice of how much space we need.
    } else {
      available = available * 2 + 1;
    }
    s.resize(available);
  }
  return s;
}
```

std::to_string reference implementation

```
template<typename S, typename P, typename V>
inline S as_string(P sprintf_like, S s, const typename S::value_type* fmt, V a) {
  size_t available = s.size();
  while (true) {
    int status = sprintf_like(&s[0], available + 1, fmt, a);
    if (status >= 0) {
      size_t used = static_cast<size_t>(status);
      if (used <= available) {
        s.resize(used);  // success: fit size
        break;           // success: return
      }
      available = used; // assume this is advice of how much space we need.
    } else {
      available = available * 2 + 1;
    }
    s.resize(available);
  }
  return s;
}
```

std::to_string reference implementation

```
template<typename S, typename P, typename V>
inline S as_string(P sprintf_like, S s, const typename S::value_type* fmt, V a) {
  size_t available = s.size();
  while (true) {
    int status = sprintf_like(&s[0], available + 1, fmt, a);
    if (status >= 0) {
      size_t used = static_cast<size_t>(status);
      if (used <= available) {
        s.resize(used);  // success: fit size
        break;           // success: return
      }
      available = used; // assume this is advice of how much space we need.
    } else {
      available = available * 2 + 1;
    }
    s.resize(available);
  }
  return s;
}
```

std::to_string reference implementation

```
template<typename S, typename P, typename V>
inline S as_string(P sprintf_like, S s, const typename S::value_type* fmt, V a) {
  size_t available = s.size();
  while (true) {
    int status = sprintf_like(&s[0], available + 1, fmt, a);
    if (status >= 0) {
      size_t used = static_cast<size_t>(status);
      if (used <= available) {
        s.resize(used);  // success: fit size
        break;           // success: return
      }
      available = used; // assume this is advice of how much space we need.
    } else {
      available = available * 2 + 1;
    }
    s.resize(available);
  }
  return s;
}
```

std::to_string reference implementation

```
template<typename S, typename P, typename V>
inline S as_string(P sprintf_like, S s, const typename S::value_type* fmt, V a) {
  size_t available = s.size();
  while (true) {
    int status = sprintf_like(&s[0], available + 1, fmt, a);
    if (status >= 0) {
      size_t used = static_cast<size_t>(status);
      if (used <= available) {
        s.resize(used);  // success: fit size
        break;           // success: return
      }
      available = used; // assume this is advice of how much space we need.
    } else {
      available = available * 2 + 1;
    }
    s.resize(available);
  }
  return s;
}
```

std::to_string reference implementation

- *to_string*:
  - SSO + *sprintf*
- *to_wstring*:
  - allocation + *sprintf* (missing SSO opportunities)
- missing copy elision opportunities

std::to_string reference implementation

- *to_string*:
  - SSO + *sprintf*
- *to_wstring*:
  - allocation + *sprintf* (missing SSO opportunities)
- missing copy elision opportunities

Test matrix	1	ULLONG_MAX
to_string	*to_string(1)*	*to_string(ULLONG_MAX)*
to_wstring	*to_wstring(1)*	*to_wstring(ULLONG_MAX)*

Proposal 1

Proposal 1

Idea:

- use char buffer on stack
- call sprintf into stack buffer
- return string from buffer on success
- fallback to the previous algorithm on failure

Proposal 1

```
template<typename S, typename P, typename V>
inline S as_string(P sprintf_like, const typename S::value_type* fmt, V a)
{
  // fast path for nice sprintf functions
  constexpr size_t size = BIG_ENOUGH_SIZE_FOR_TYPE_V;
  typename S::value_type tmp[size] = {};
  const int len = sprintf_like(tmp, size, fmt, a);
  if (len <= size)
    return S(tmp, tmp + len);  // copy elision guarantee since C++17

  // fallback to previous algorithm for weird sprintf functions
  S s;
  ...
  return s;
}
```

Proposal 1

```
template<typename S, typename P, typename V>
inline S as_string(P sprintf_like, const typename S::value_type* fmt, V a)
{
  // fast path for nice sprintf functions
  constexpr size_t size = BIG_ENOUGH_SIZE_FOR_TYPE_V;
  typename S::value_type tmp[size] = {};
  const int len = sprintf_like(tmp, size, fmt, a);
  if (len <= size)
    return S(tmp, tmp + len);  // copy elision guarantee since C++17

  // fallback to previous algorithm for weird sprintf functions
  S s;
  ...
  return s;
}
```

Proposal 1

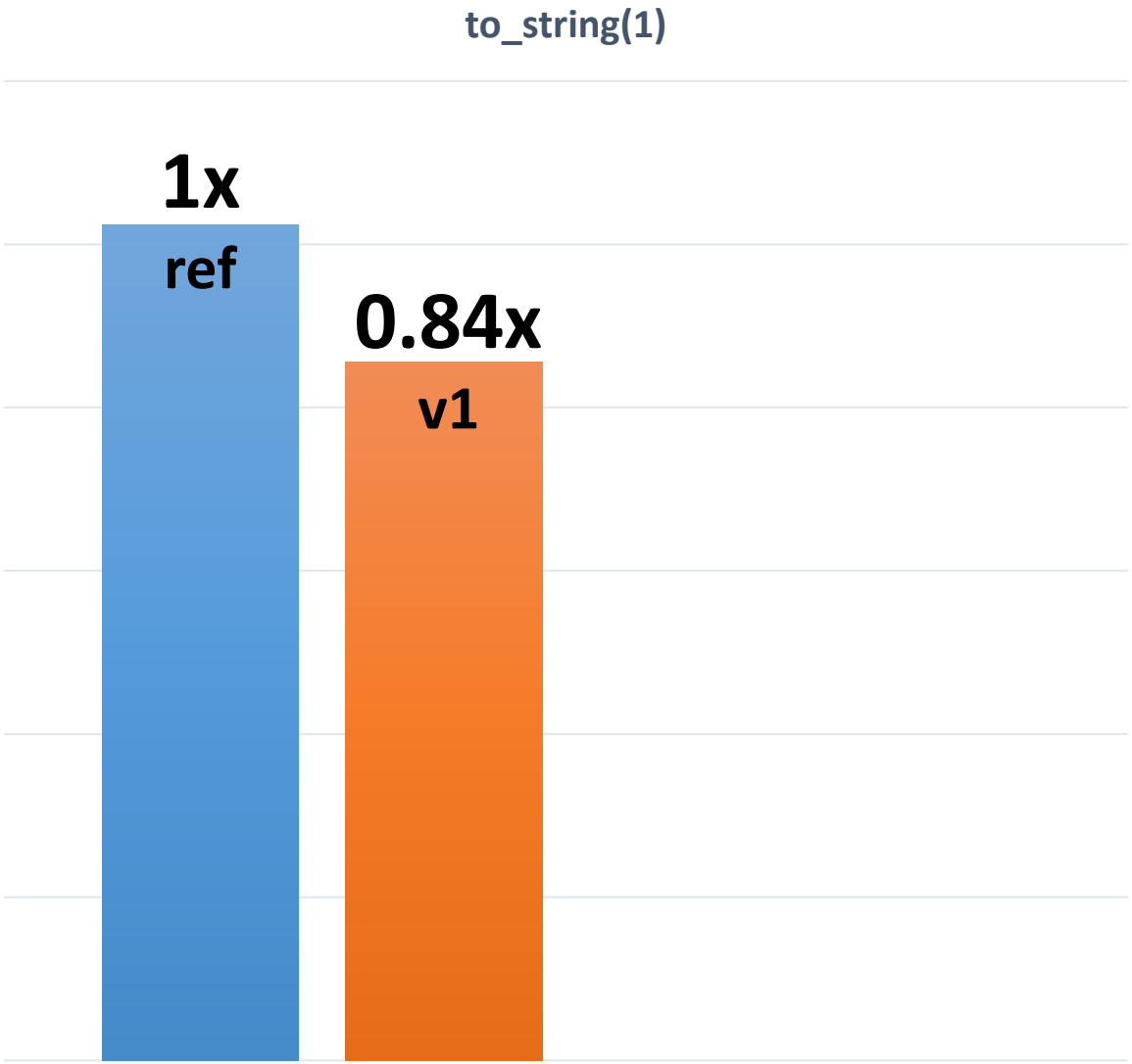
```
template<typename S, typename P, typename V>
inline S as_string(P sprintf_like, const typename S::value_type* fmt, V a)
{
  // fast path for nice sprintf functions
  constexpr size_t size = BIG_ENOUGH_SIZE_FOR_TYPE_V;
  typename S::value_type tmp[size] = {};
  const int len = sprintf_like(tmp, size, fmt, a);
  if (len <= size)
    return S(tmp, tmp + len);  // copy elision guarantee since C++17

  // fallback to previous algorithm for weird sprintf functions
  S s;
  ...
  return s;
}
```

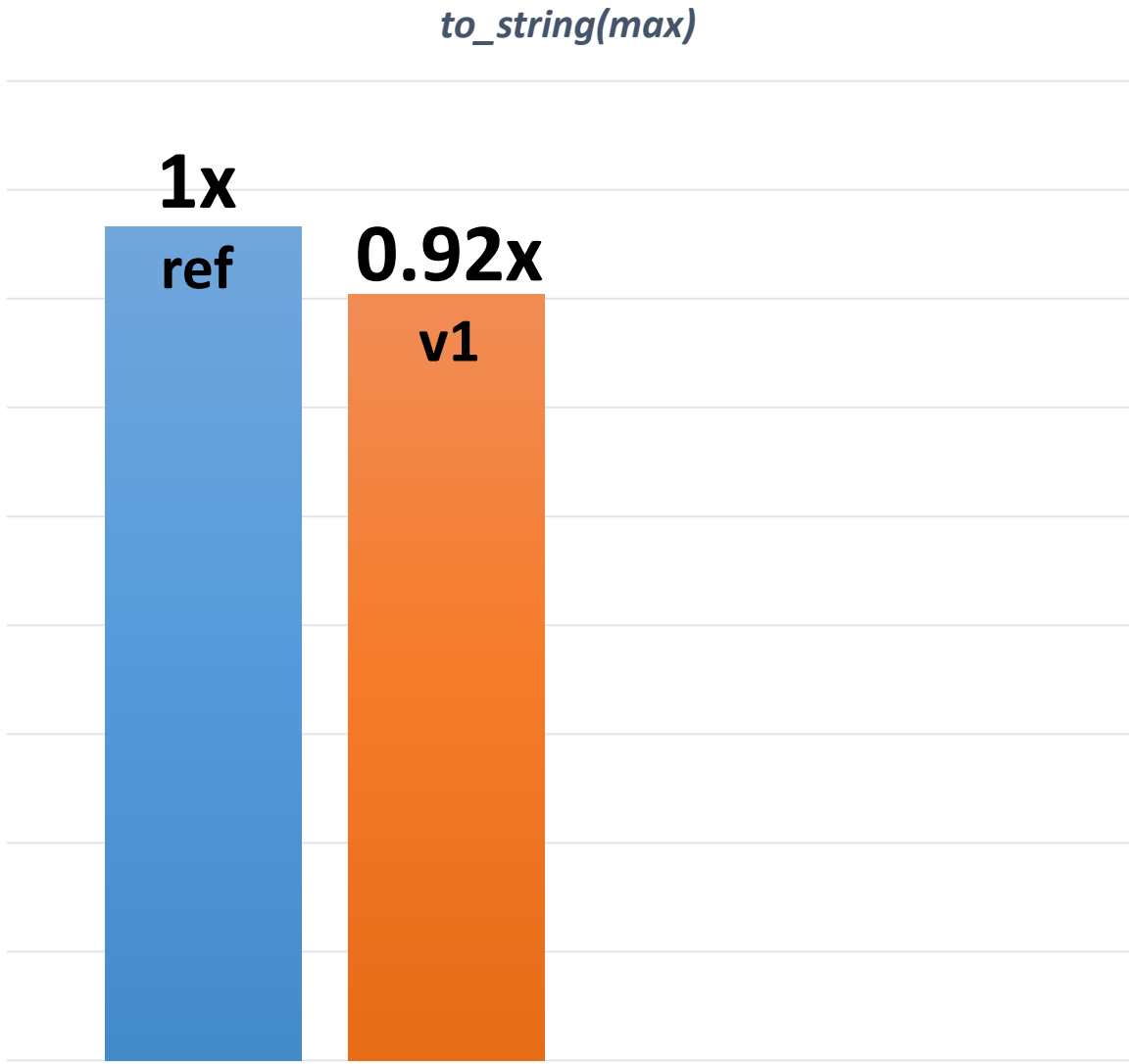
Proposal 1

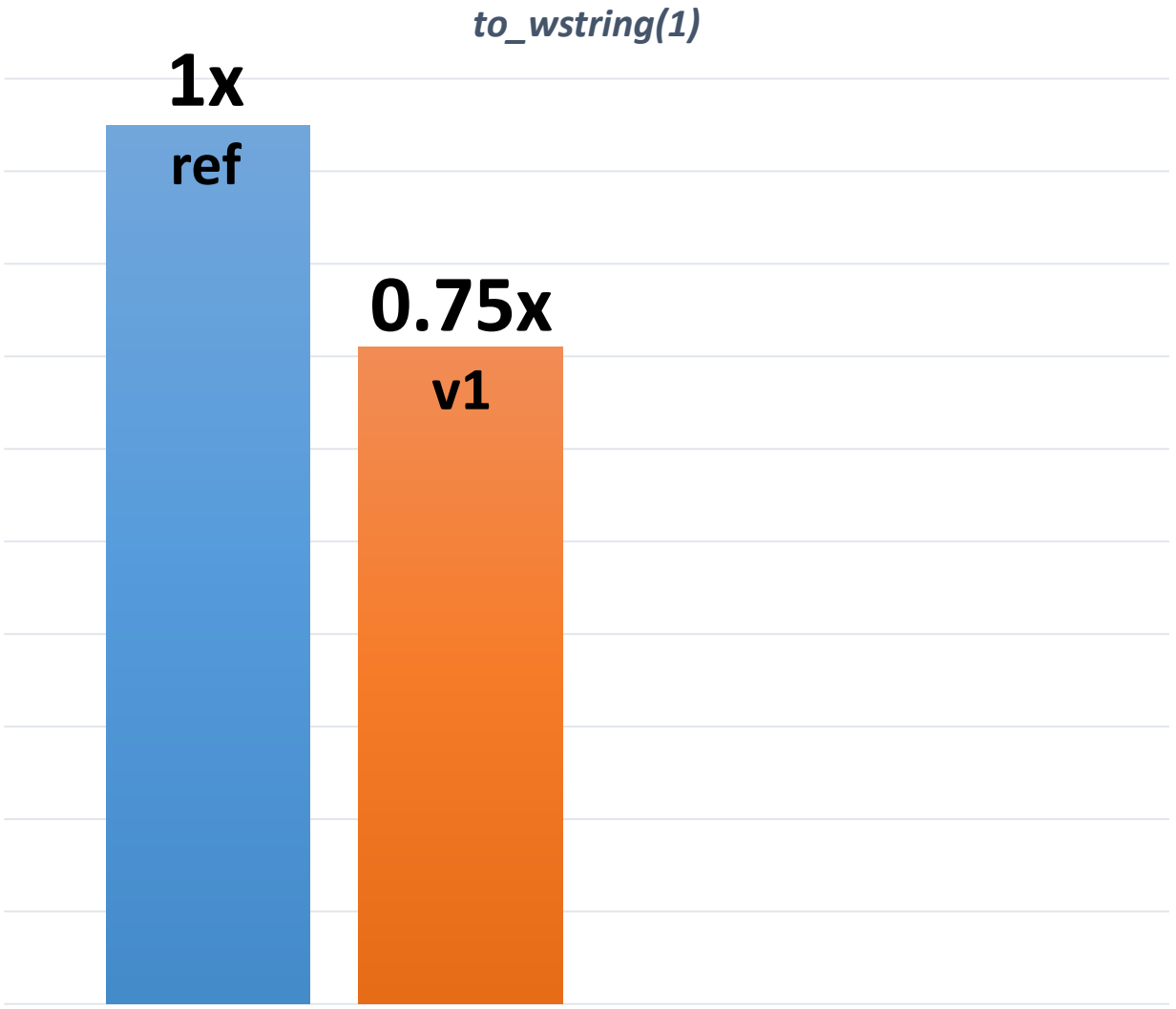
```
template<typename S, typename P, typename V>
inline S as_string(P sprintf_like, const typename S::value_type* fmt, V a)
{
  // fast path for nice sprintf functions
  constexpr size_t size = BIG_ENOUGH_SIZE_FOR_TYPE_V;
  typename S::value_type tmp[size] = {};
  const int len = sprintf_like(tmp, size, fmt, a);
  if (len <= size)
    return S(tmp, tmp + len);  // copy elision guarantee since C++17

  // fallback to previous algorithm for weird sprintf functions
  S s;
  ...
  return s;
}
```

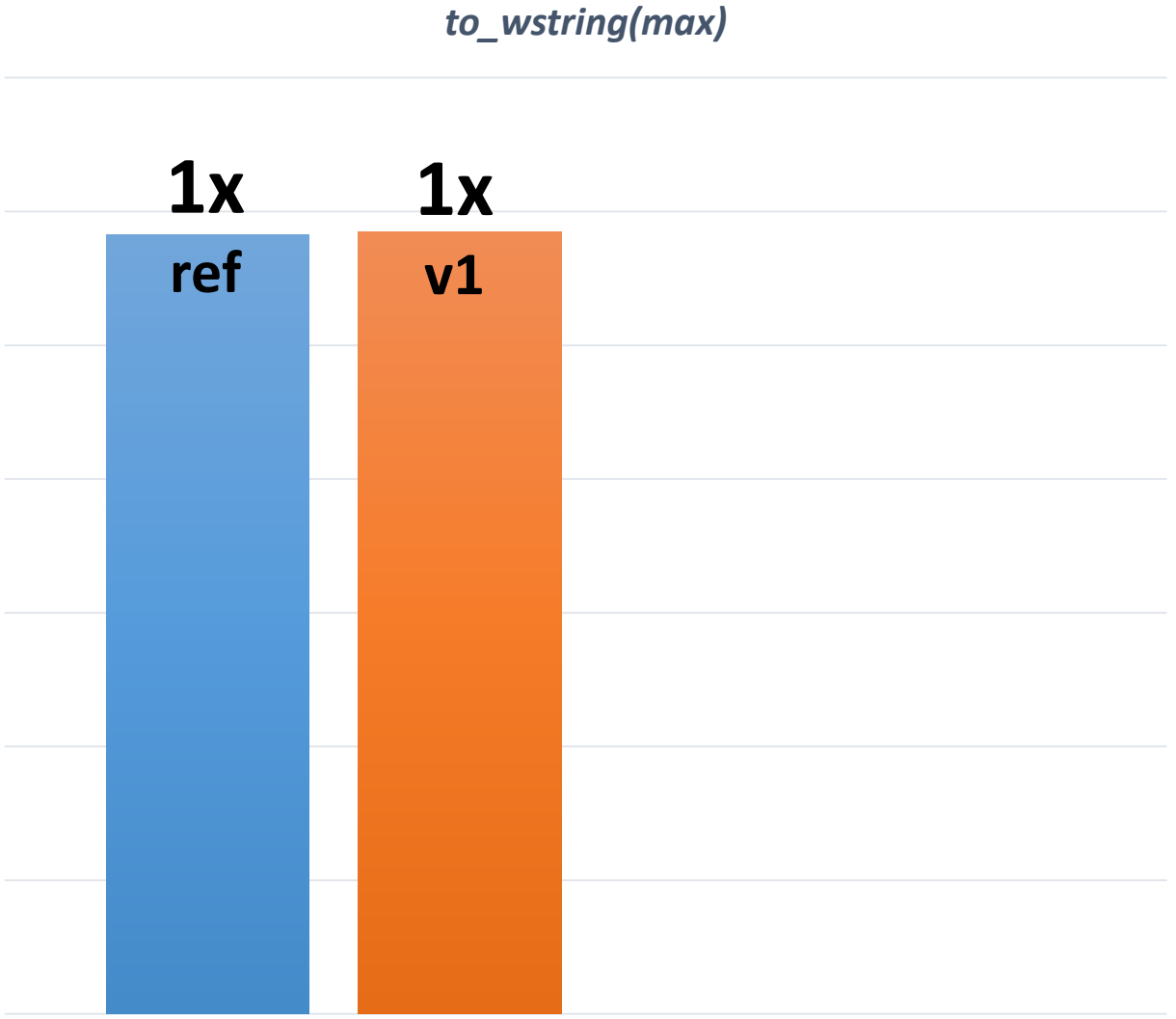
Proposal 1

```
template<typename S, typename P, typename V>
inline S as_string(P sprintf_like, const typename S::value_type* fmt, V a)
{
  // fast path for nice sprintf functions
  constexpr size_t size = BIG_ENOUGH_SIZE_FOR_TYPE_V;
  typename S::value_type tmp[size] = {};
  const int len = sprintf_like(tmp, size, fmt, a);
  if (len <= size)
    return S(tmp, tmp + len);  // copy elision guarantee since C++17

  // fallback to previous algorithm for weird sprintf functions
  S s;
  ...
  return s;
}
```

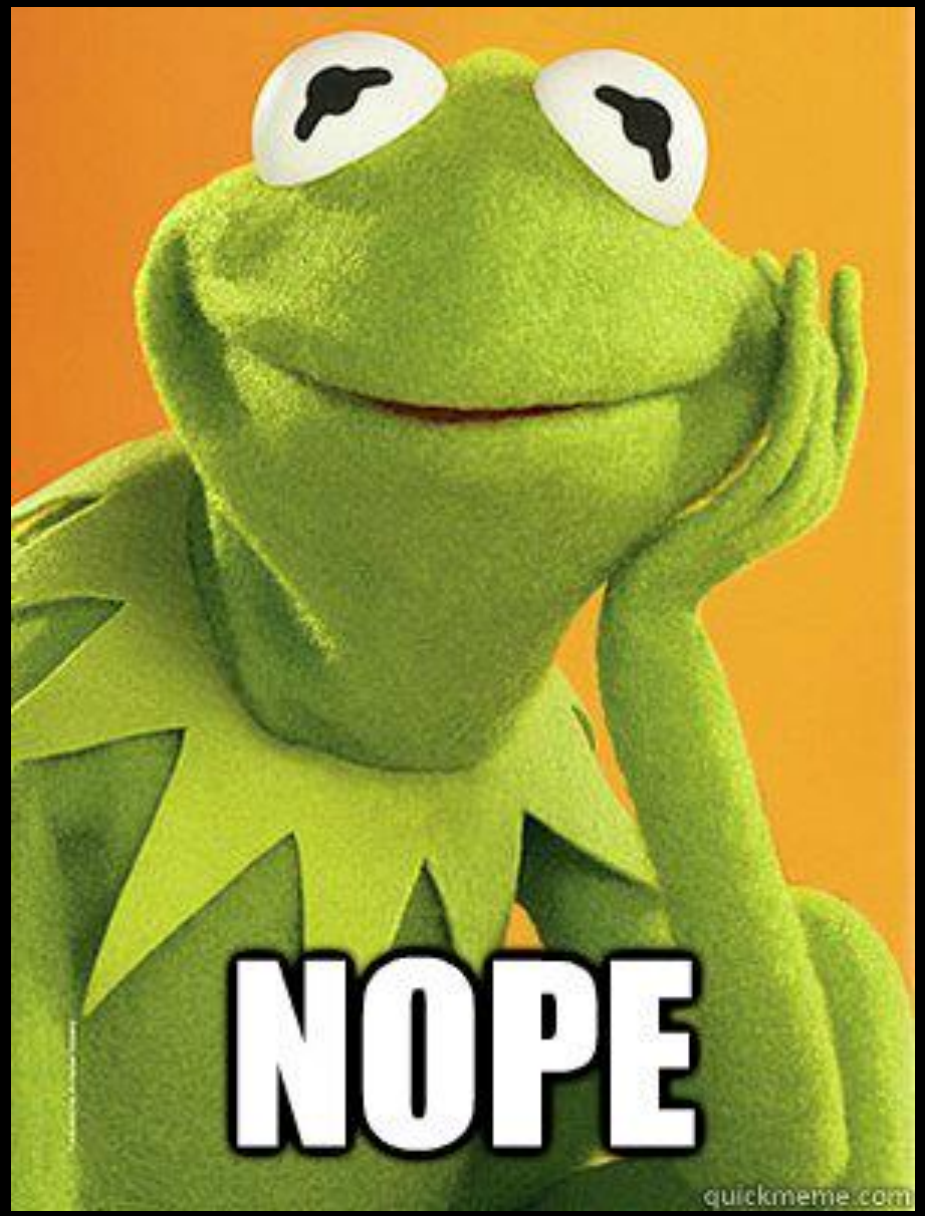

to_string(1)
1x
ref
0.84x
v1


to_string(max)
1x
ref
0.92x
v1


to_wstring(1)
1x
ref
0.75x
v1


to_wstring(max)
1x
ref
1x
v1

25% speedup… seems nice?


NOPE
quickmeme.com

Proposal 2

Proposal 2

Ed Schouten: “If performance matters, it makes sense to handroll this for the integer cases”

Proposal 2

- use *val % 10* and val / 10 (2 division ops per char)

Proposal 2

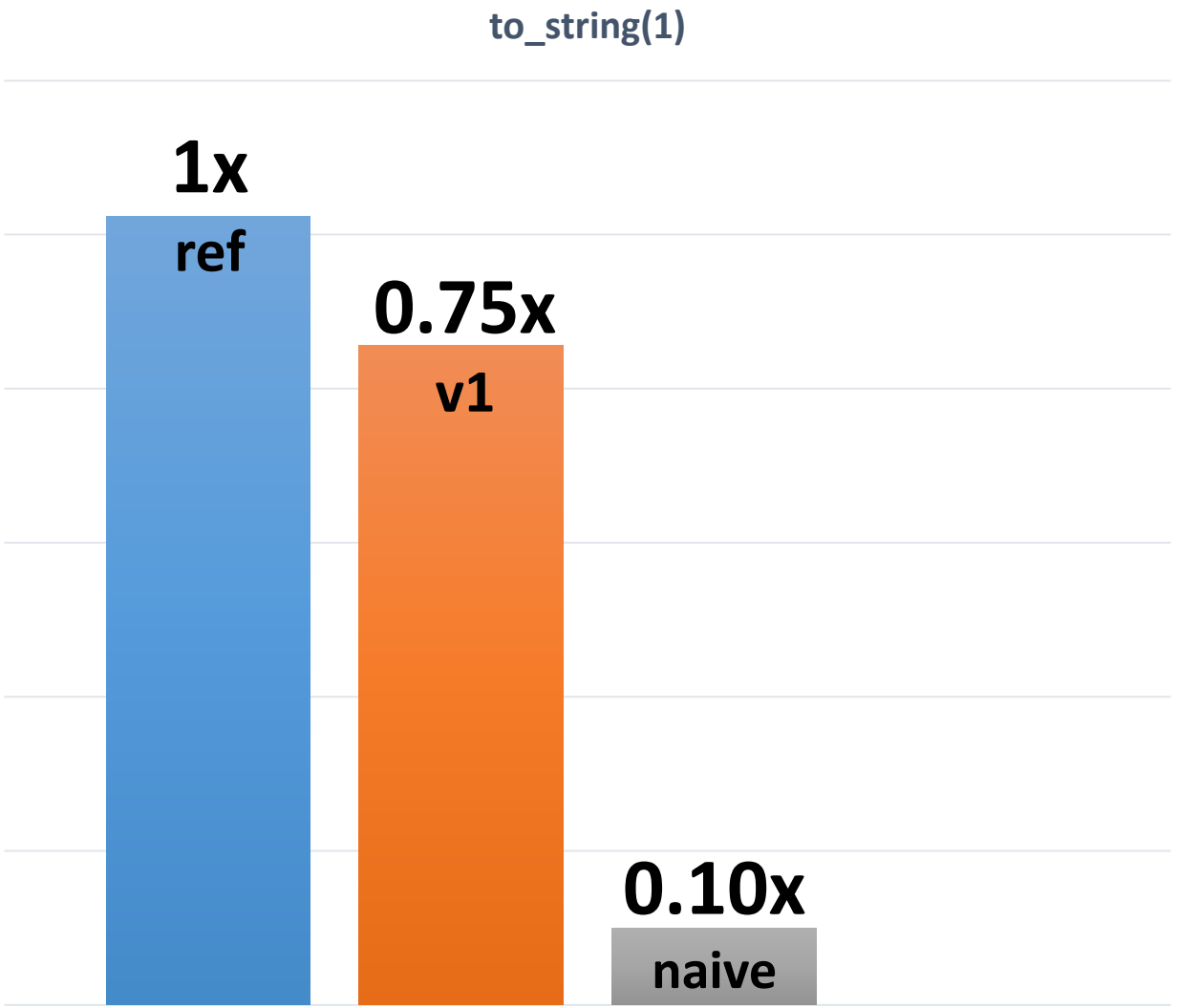
- use *val % 10* and val / 10 (2 division ops per char)
- fill stack buffer from right to left

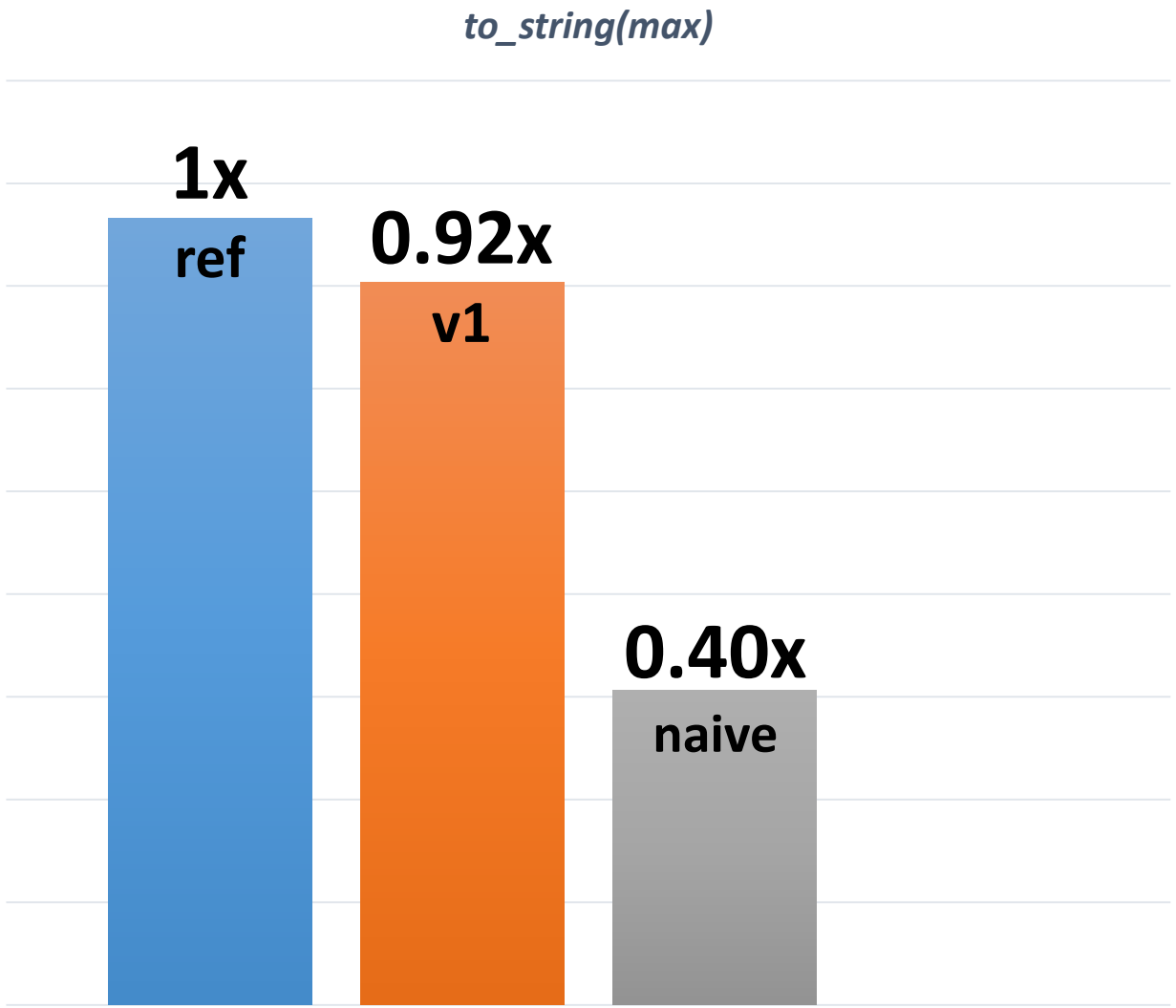
Proposal 2

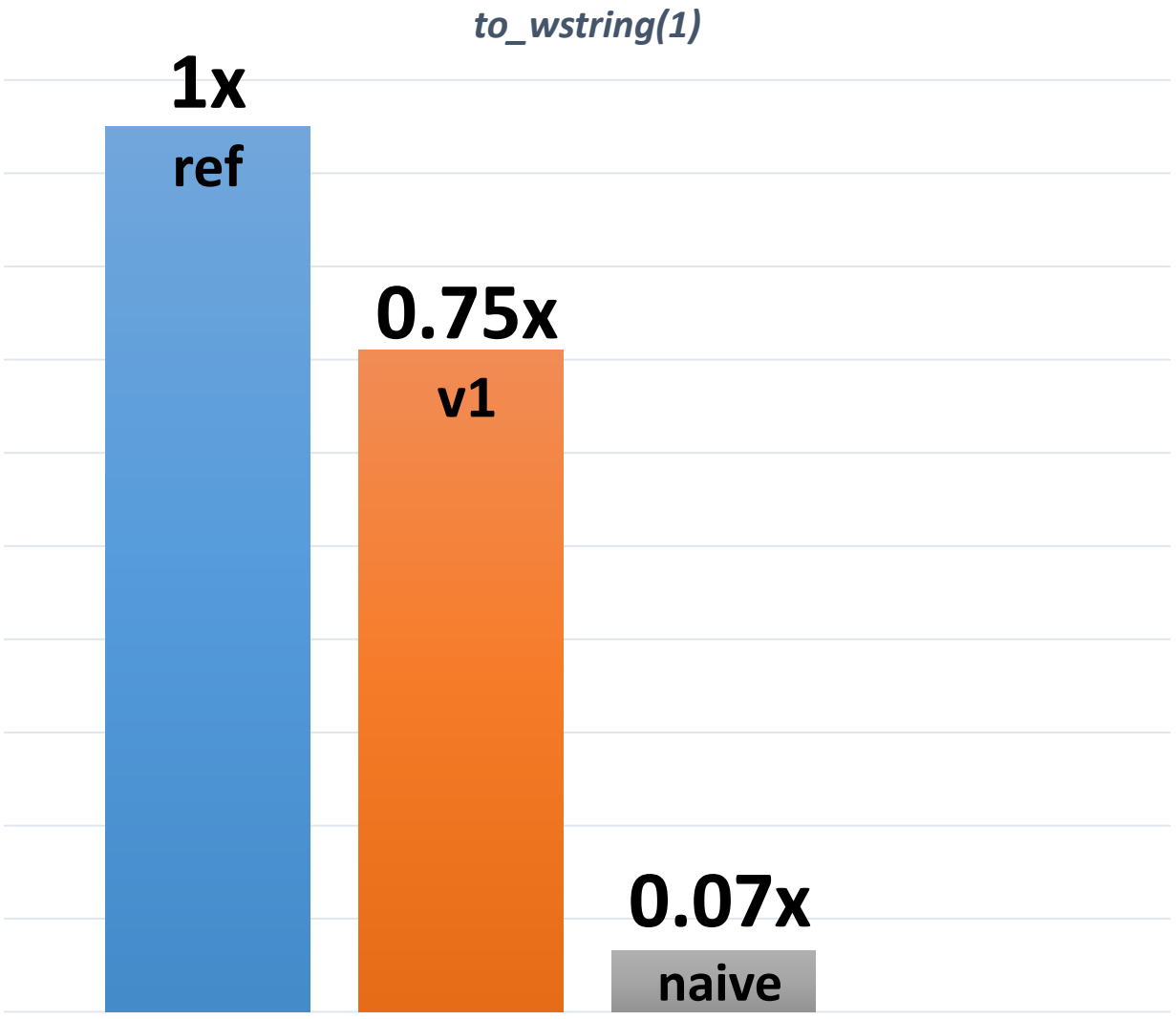
- use *val % 10* and val / 10 (2 division ops per char)
- fill stack buffer from right to left
- do not forget about negative numbers

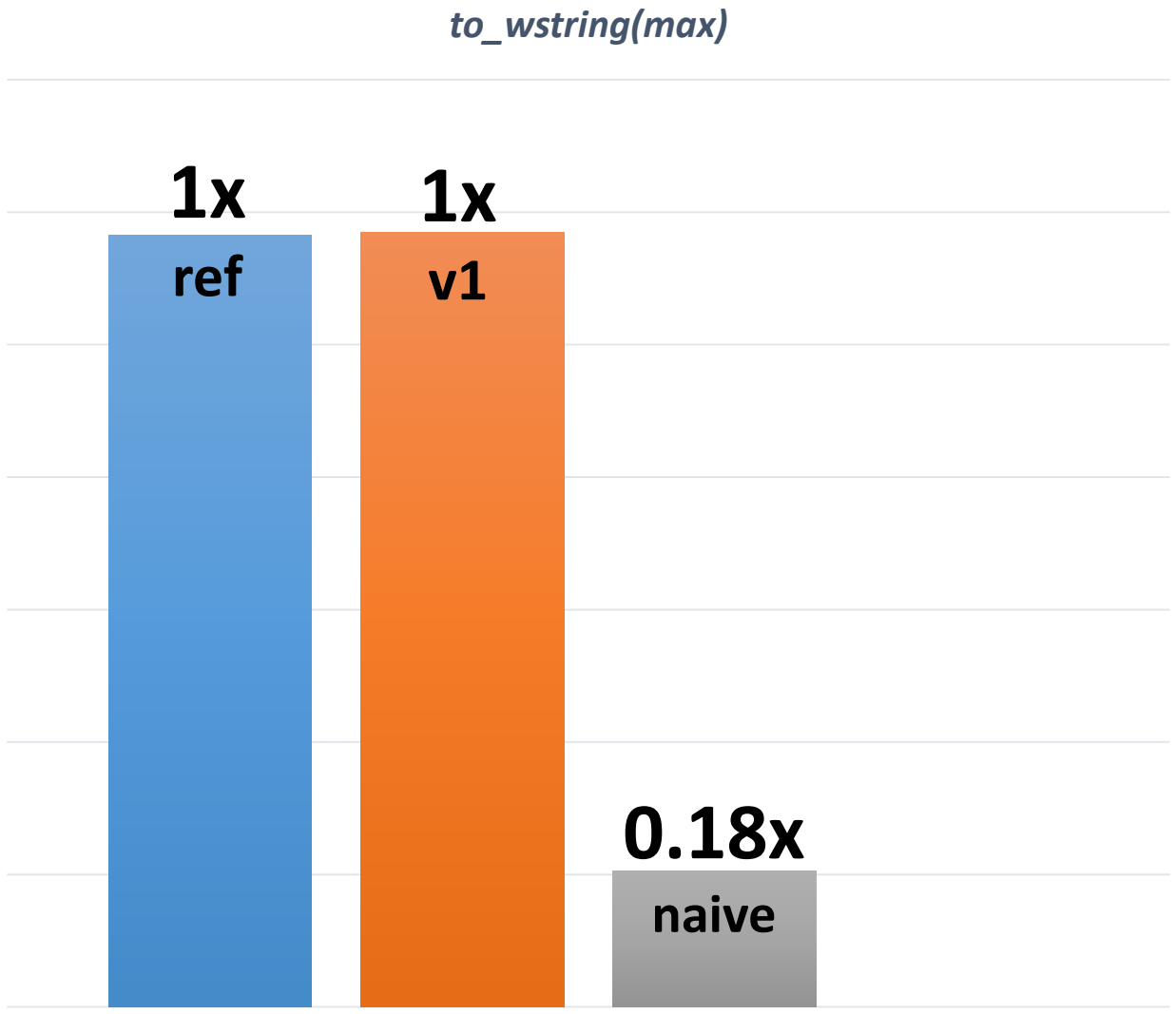
Proposal 2

- use *val % 10* and val / 10 (2 division ops per char)
- fill stack buffer from right to left
- do not forget about negative numbers
  - (val % 10) <= 0 according to C++ standard par. "Multiplicative operators" (ref. to "truncation towards zero“ since C99)
  - need to deal with sign
  - *std::numeric_limits<V>::min()*


to_string(1)
1x
ref
0.75x
v1
0.10x
naive


to_string(max)
1x
ref
0.92x
v1
0.40x
naive


to_wstring(1)
1x
ref
0.75x
v1
0.07x
naive


to_wstring(max)
1x
ref
1x
v1
0.18x
naive

Why sprintf is so slow?

?

Why sprintf is so slow?

- it is general-purpose algorithm

Why sprintf is so slow?

- it is general-purpose algorithm
- format parsing?

Why sprintf is so slow?

- it is general-purpose algorithm
- format parsing?
- varargs?

Why sprintf is so slow?

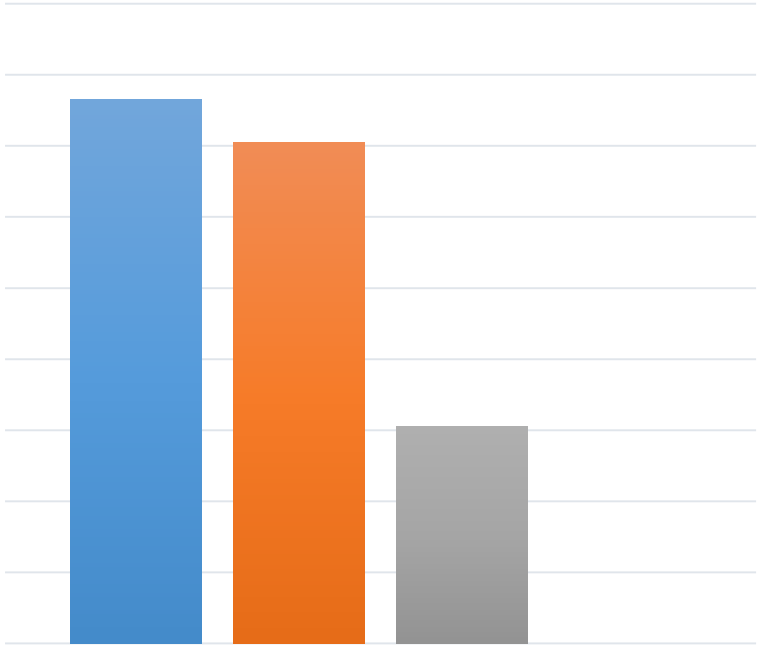
- it is general-purpose algorithm
- format parsing?
- varargs?
- locale and synchronizations?

Why sprintf is so slow?

- it is general-purpose algorithm
- format parsing?
- varargs?
- locale and synchronizations?

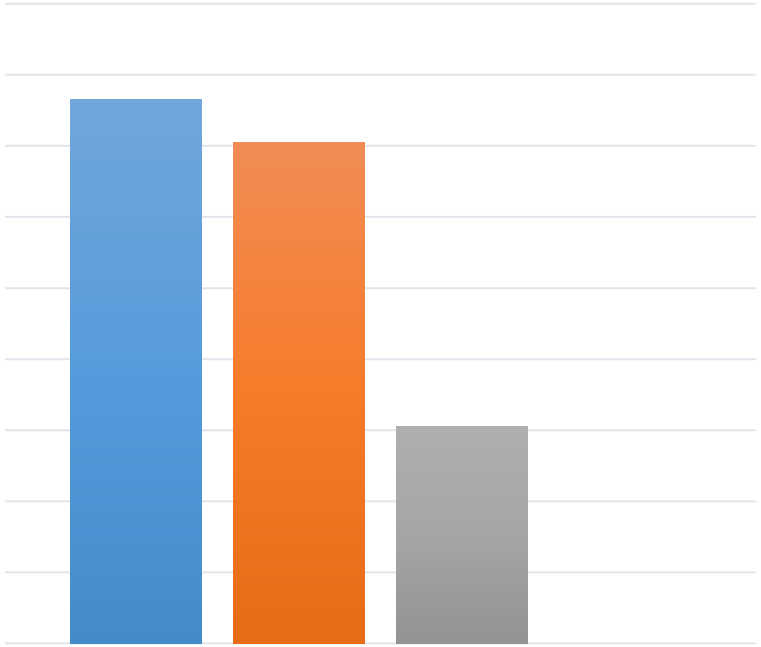
Let's profile it!

sprintf profile for ULLONG_MAX



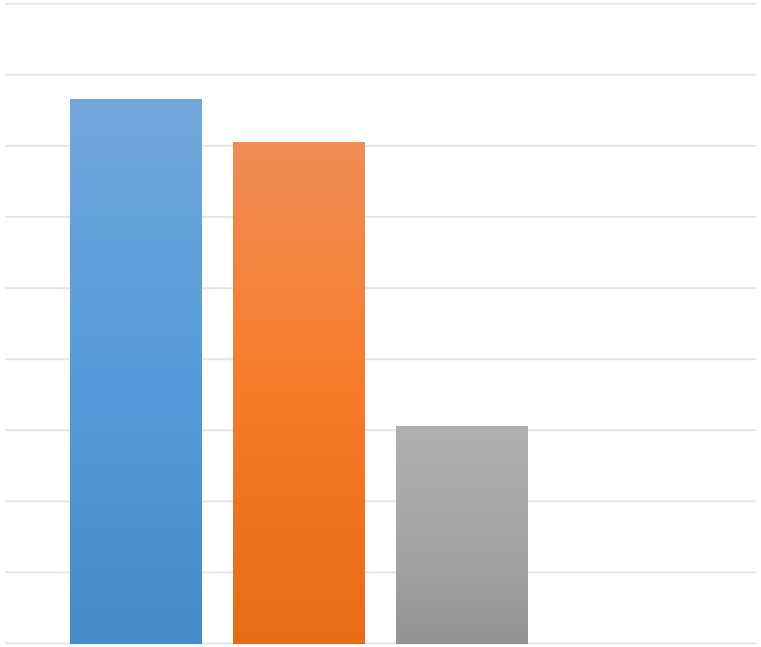
% time	cumulative seconds	self seconds	calls	self Ts / call	total Ts / call	name
38.99	43.04	43.04				_itoa_word
29.43	75.53	32.49				vfprintf
10.84	87.50	11.97				_IO_default_xsputn
4.22	92.16	4.66				___vsprintf_chk
4.16	96.75	4.60				__strchrnul_avx2
3.18	100.27	3.52				_IO_str_init_static_internal
2.60	103.13	2.87				_IO_no_init
...						

sprintf profile for ULLONG_MAX



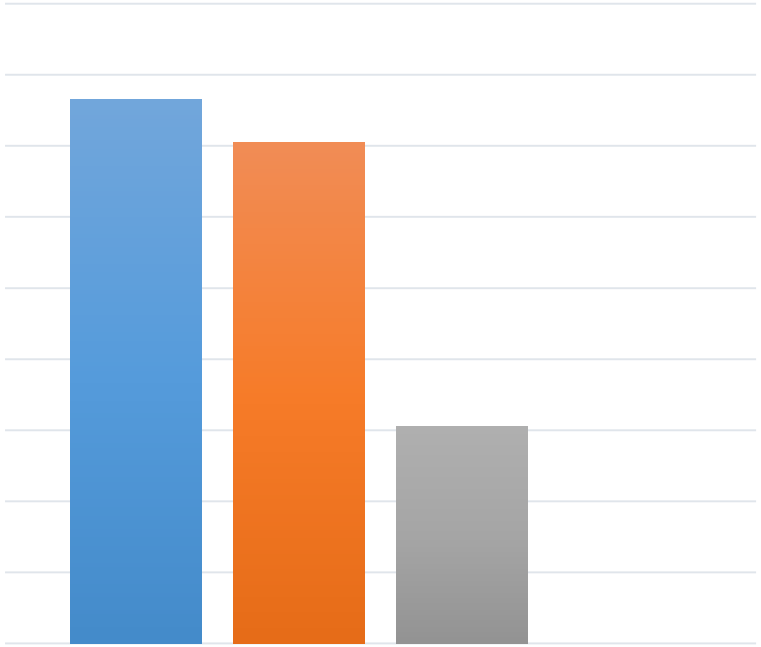
% time	cumulative seconds	self seconds	calls	self Ts / call	total Ts / call	name
38.99	43.04	43.04				_itoa_word
29.43	75.53	32.49				vfprintf
10.84	87.50	11.97				_IO_default_xsputn
4.22	92.16	4.66				___vsprintf_chk
4.16	96.75	4.60				__strchrnul_avx2
3.18	100.27	3.52				_IO_str_init_static_internal
2.60	103.13	2.87				_IO_no_init
...						

sprintf profile for ULLONG_MAX



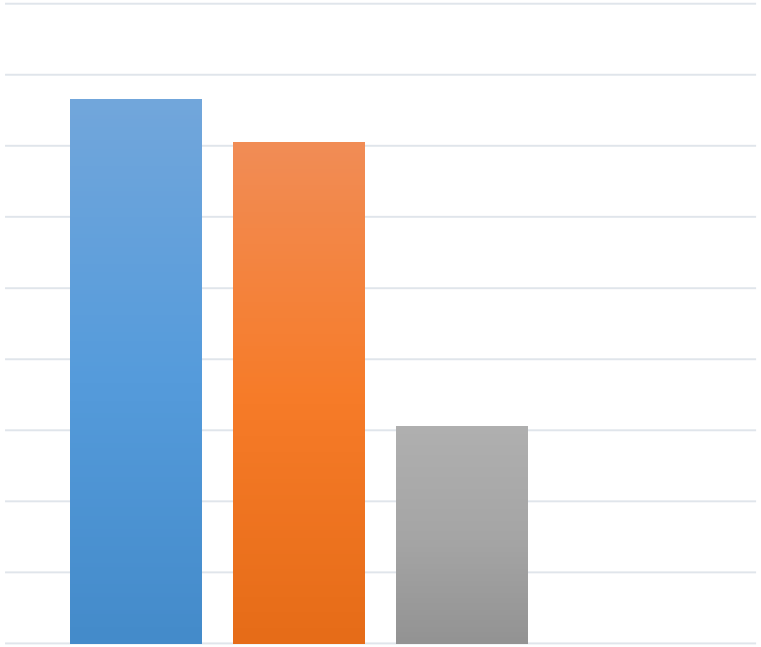
% time	cumulative seconds	self seconds	calls	self Ts / call	total Ts / call	name
38.99	43.04	43.04				_itoa_word
29.43	75.53	32.49				vfprintf
10.84	87.50	11.97				_IO_default_xsputn
4.22	92.16	4.66				___vsprintf_chk
4.16	96.75	4.60				__strchrnul_avx2
3.18	100.27	3.52				_IO_str_init_static_internal
2.60	103.13	2.87				_IO_no_init
...						

sprintf profile for ULLONG_MAX



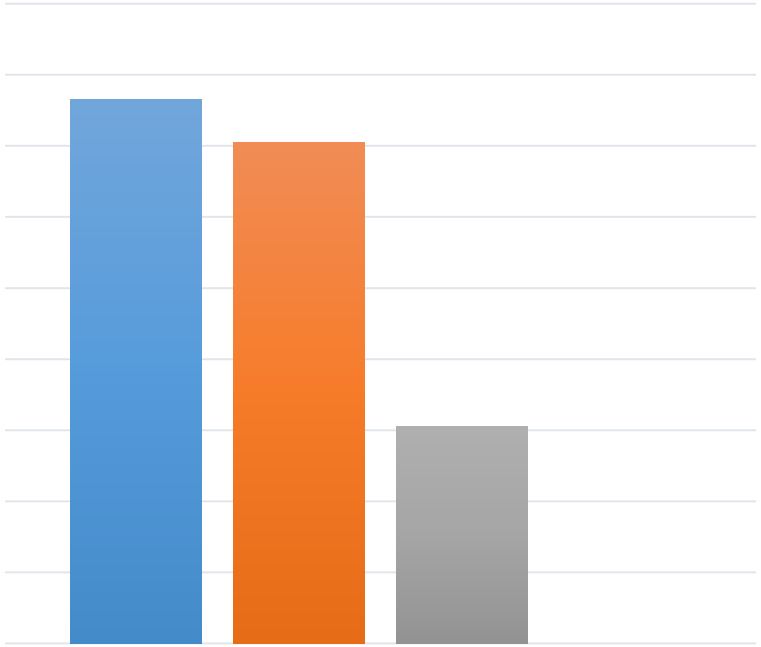
% time	cumulative seconds	self seconds	calls	self Ts / call	total Ts / call	name
38.99	43.04	43.04				_itoa_word
29.43	75.53	32.49				vfprintf
10.84	87.50	11.97				_IO_default_xsputn
4.22	92.16	4.66				___vsprintf_chk
4.16	96.75	4.60				__strchrnul_avx2
3.18	100.27	3.52				_IO_str_init_static_internal
2.60	103.13	2.87				_IO_no_init
...						

sprintf profile for ULLONG_MAX



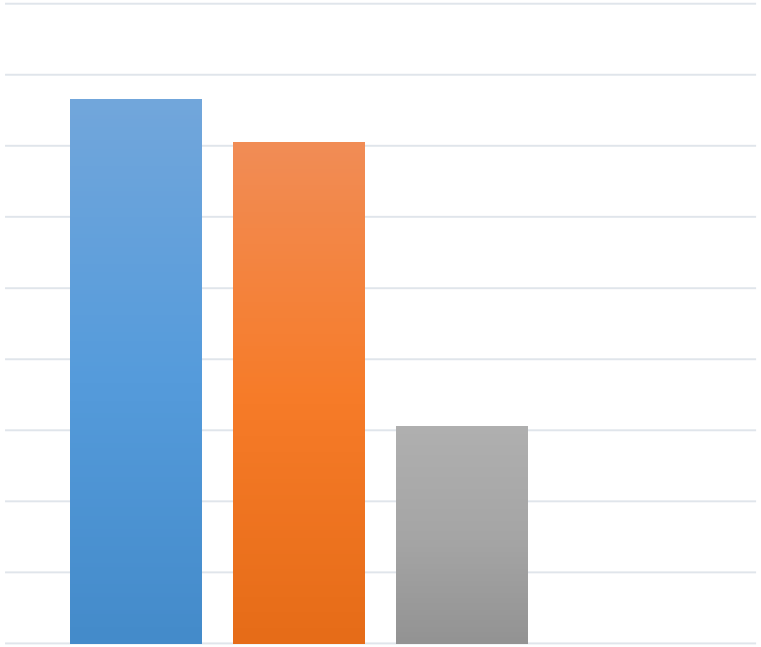
% time	cumulative seconds	self seconds	calls	self Ts / call	total Ts / call	name
38.99	43.04	43.04				_itoa_word
29.43	75.53	32.49				vfprintf
10.84	87.50	11.97				_IO_default_xsputn
4.22	92.16	4.66				___vsprintf_chk
4.16	96.75	4.60				__strchrnul_avx2
3.18	100.27	3.52				_IO_str_init_static_internal
2.60	103.13	2.87				_IO_no_init
...						

sprintf profile for ULLONG_MAX



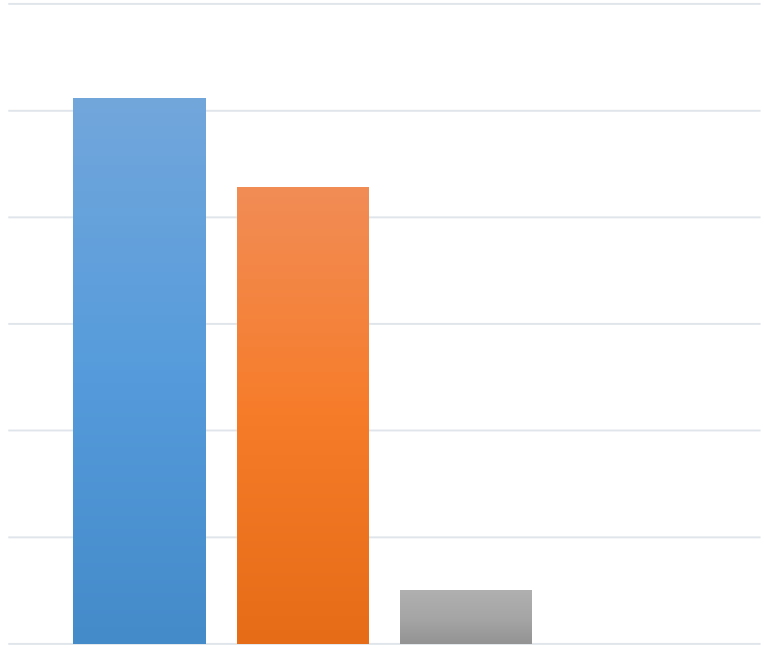
% time	cumulative seconds	self seconds	calls	self Ts / call	total Ts / call	name
38.99	43.04	43.04				_itoa_word
29.43	75.53	32.49				vfprintf
10.84	87.50	11.97				_IO_default_xsputn
4.22	92.16	4.66				___vsprintf_chk
4.16	96.75	4.60				__strchrnul_avx2
3.18	100.27	3.52				_IO_str_init_static_internal
2.60	103.13	2.87				_IO_no_init
...						

sprintf profile for ULLONG_MAX



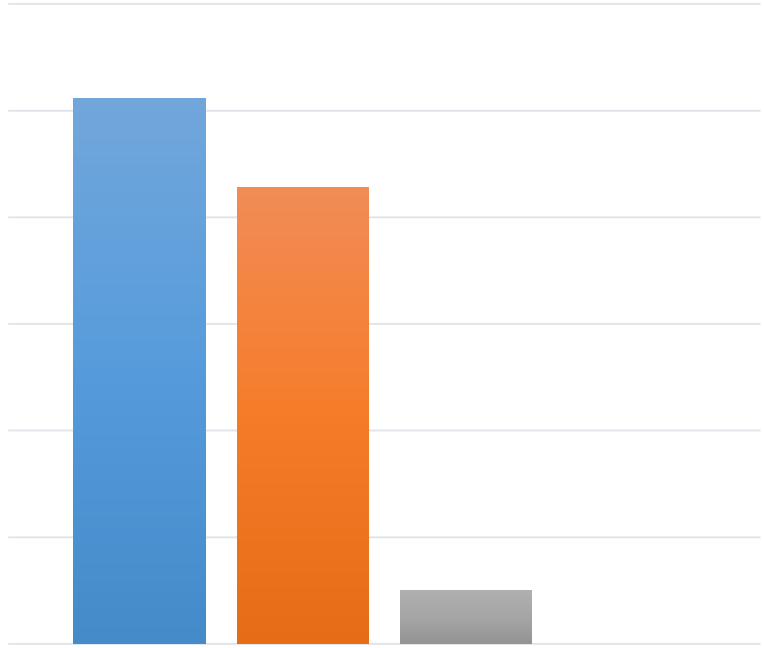
% time	cumulative seconds	self seconds	calls	self Ts / call	total Ts / call	name
38.99	43.04	43.04				_itoa_word
29.43	75.53	32.49				vfprintf
10.84	87.50	11.97				_IO_default_xsputn
4.22	92.16	4.66				___vsprintf_chk
4.16	96.75	4.60				__strchrnul_avx2
3.18	100.27	3.52				_IO_str_init_static_internal
2.60	103.13	2.87				_IO_no_init
...						

sprintf profile for 1



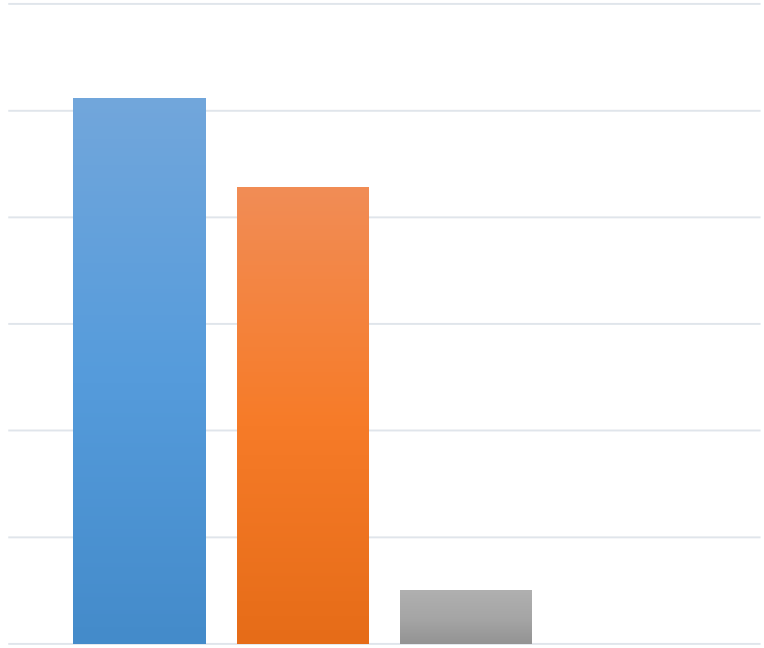
```
 %   cumulative   self              self     total
  time   seconds   seconds    calls  Ts / call  Ts / call  name
  47.45     31.87    31.87                             vfprintf
  10.35     38.82     6.95                             __strchrnul_avx2
  8.14     44.29     5.47                             _IO_default_xsputn
  6.97     48.97     4.68                             ___vsprintf_chk
...
```

sprintf profile for 1



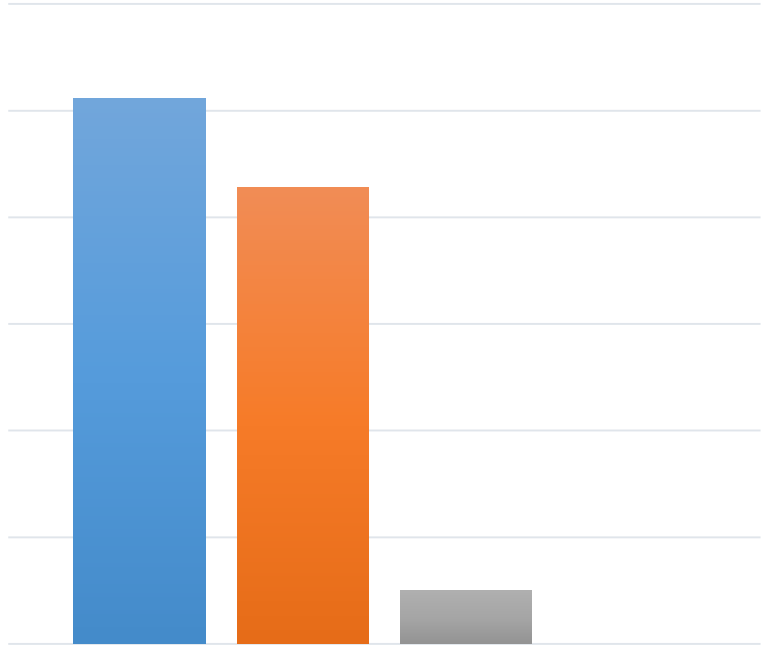
```
 %   cumulative   self              self     total
  time   seconds   seconds    calls  Ts / call  Ts / call  name
  47.45     31.87    31.87                             vfprintf
  10.35     38.82     6.95                             __strchrnul_avx2
  8.14     44.29     5.47                             _IO_default_xsputn
  6.97     48.97     4.68                             ___vsprintf_chk
...
```

sprintf profile for 1



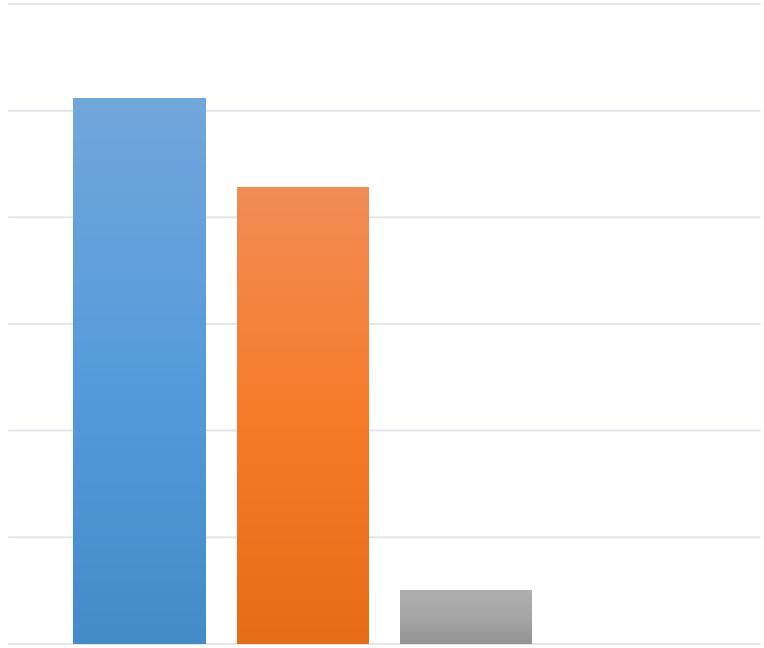
```
%   cumulative   self              self     total
 time   seconds   seconds    calls  Ts / call  Ts / call  name
 47.45     31.87    31.87                             vfprintf
 10.35     38.82     6.95                             __strchrnul_avx2
 8.14      44.29     5.47                             _IO_default_xsputn
 6.97      48.97     4.68                             ___vsprintf_chk
...
```

sprintf profile for 1



```
%   cumulative   self              self     total
 time   seconds   seconds    calls  Ts / call  Ts / call  name
 47.45     31.87    31.87                             vfprintf
 10.35     38.82     6.95                             __strchrnul_avx2
 8.14     44.29     5.47                             _IO_default_xsputn
 6.97     48.97     4.68                             ___vsprintf_chk
...
```

sprintf profile for 1



```
%   cumulative   self              self     total
 time   seconds   seconds    calls  Ts / call  Ts / call  name
 47.45     31.87    31.87                             vfprintf
 10.35     38.82     6.95                             __strchrnul_avx2
 8.14      44.29     5.47                             _IO_default_xsputn
 6.97      48.97     4.68                             ___vsprintf_chk
 6.70      53.47     4.50                             _IO_no_init
 5.07      56.88     3.41                             _IO_str_init_static_internal
 4.23      59.72     2.84                             _itoa_word
...
```

sprintf profile conclusions

sprintf profile conclusions

- format parsing is expensive

sprintf profile conclusions

- format parsing is expensive
- internal structures overhead

sprintf profile conclusions

- format parsing is expensive
- internal structures overhead
- sanity checks overhead

sprintf profile conclusions

- format parsing is expensive
- internal structures overhead
- sanity checks overhead
- locale access is lazy (no synchronizations)

93% speedup… seems nice?


HAHAHA
NOPE
memegenerator.net

Proposal 3

Proposal 3

Marshall Clow: “just reuse *std::to_chars*”

std::to_chars

Defined in header <charconv>

```
std::to_chars_result to_chars(char* first, char* last,
                              /*see below*/ value, int base = 10);          (1)

struct to_chars_result {
    char* ptr;
    std::errc ec;
};                                                                          (5)
```

Proposal 3

Marshall Clow: “just reuse *std::to_chars*”

std::to_chars

Defined in header <charconv>

```
std::to_chars_result to_chars(char* first, char* last,
                              /*see below*/ value, int base = 10);
```

(1) (since C++17)

```
struct to_chars_result {
    char* ptr;
    std::errc ec;
};
```

(5) (since C++17)

Proposal 3

Marshall Clow: “just reuse *std::to_chars*”

std::to_chars

Defined in header <charconv>

```
std::to_chars_result to_chars(char* first, char* last,
                              /*see below*/ value, int base = 10);
```

(1) (since C++17)

```
struct to_chars_result {
    char* ptr;
    std::errc ec;
};
```

(5) (since C++17)

std::to_string

Defined in header <string>

```
std::string to_string( int value );
```

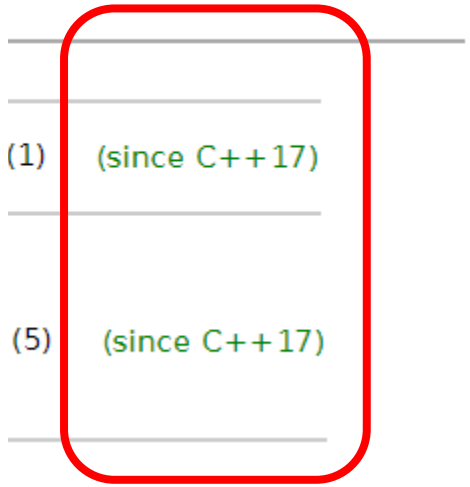
(1) (since C++11)

Proposal 3

Marshall Clow: “just reuse *std::to_chars*”

Marshall Clow: “not a problem, *std::to_chars* is since C++11 now:” (libc++ only)

https://reviews.llvm.org/D59598



std::to_string

Defined in header <string>

```
std::string to_string( int value );
```

(1) (since C++11)

Proposal 3

```
template <typename S, typename V>
S i_to_string(const V v)
{
  constexpr size_t bufsize = numeric_limits<V>::digits10 + 2;
  char buf[bufsize];
  const auto res = to_chars(buf, buf + bufsize, v);
  return S(buf, res.ptr);
}
```

Proposal 3

```
template <typename S, typename V>
S i_to_string(const V v)
{
  constexpr size_t bufsize = numeric_limits<V>::digits10 + 2;
  char buf[bufsize];
  const auto res = to_chars(buf, buf + bufsize, v);
  return S(buf, res.ptr);
}
```

Proposal 3

```
template <typename S, typename V>
S i_to_string(const V v)
{
  constexpr size_t bufsize = numeric_limits<V>::digits10 + 2;
  char buf[bufsize];
  const auto res = to_chars(buf, buf + bufsize, v);
  return S(buf, res.ptr);
}
```

Proposal 3

```
template <typename S, typename V>
S i_to_string(const V v)
{
  constexpr size_t bufsize = numeric_limits<V>::digits10 + 2;
  char buf[bufsize];
  const auto res = to_chars(buf, buf + bufsize, v);
  return S(buf, res.ptr);
}
```

Proposal 3

```
template <typename S, typename V>
S i_to_string(const V v)
{
  constexpr size_t bufsize = numeric_limits<V>::digits10 + 2;
  char buf[bufsize];
  const auto res = to_chars(buf, buf + bufsize, v);
  return S(buf, res.ptr);
}
```

Proposal 3

```
template <typename S, typename V>
S i_to_string(const V v)
{
  constexpr size_t bufsize = numeric_limits<V>::digits10 + 2;
  char buf[bufsize];
  const auto res = to_chars(buf, buf + bufsize, v);
  return S(buf, res.ptr);
}
```

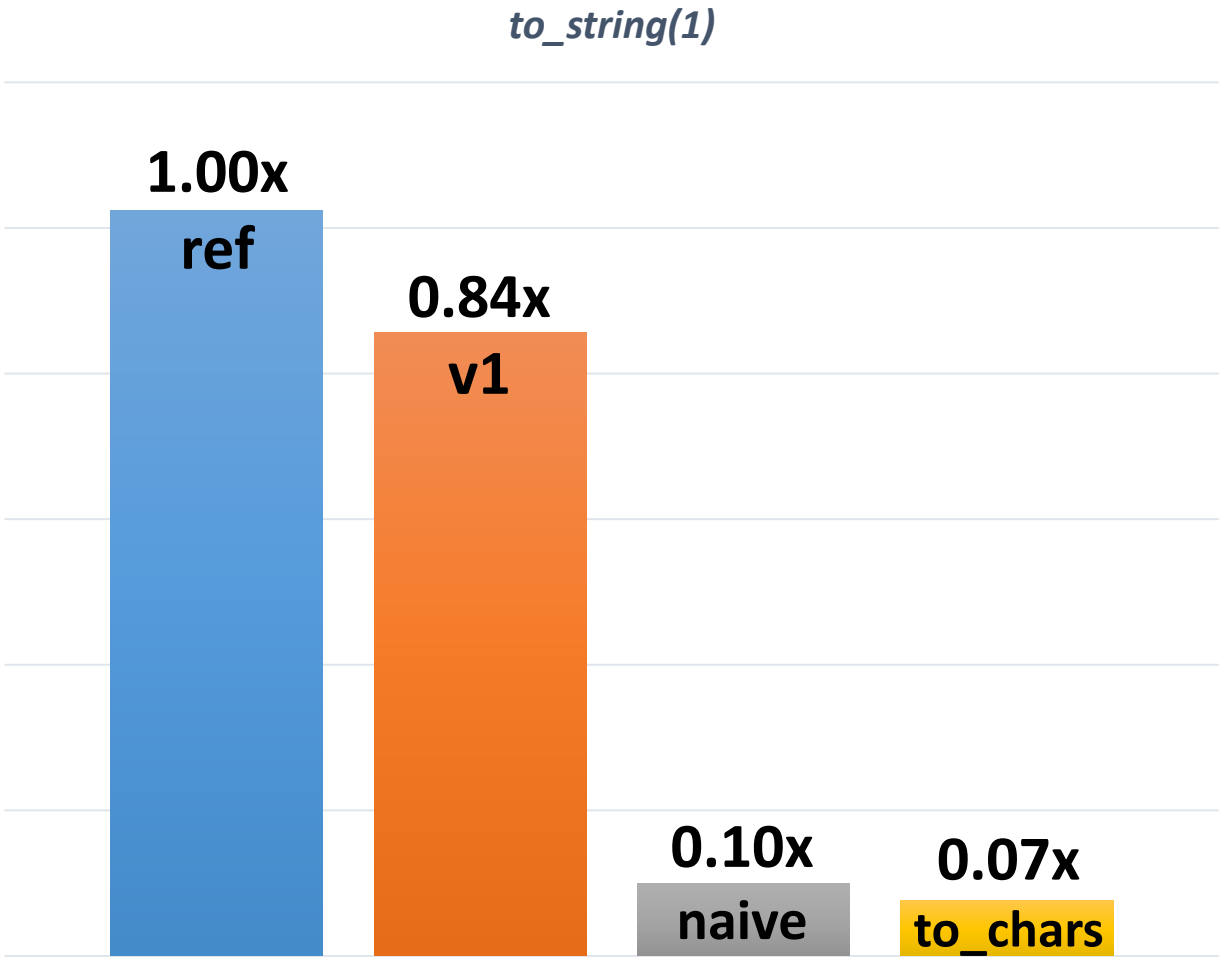
The value of `std::numeric_limits<T>::digits10` is the number of base-10 digits that can be represented by the type `T` without change.

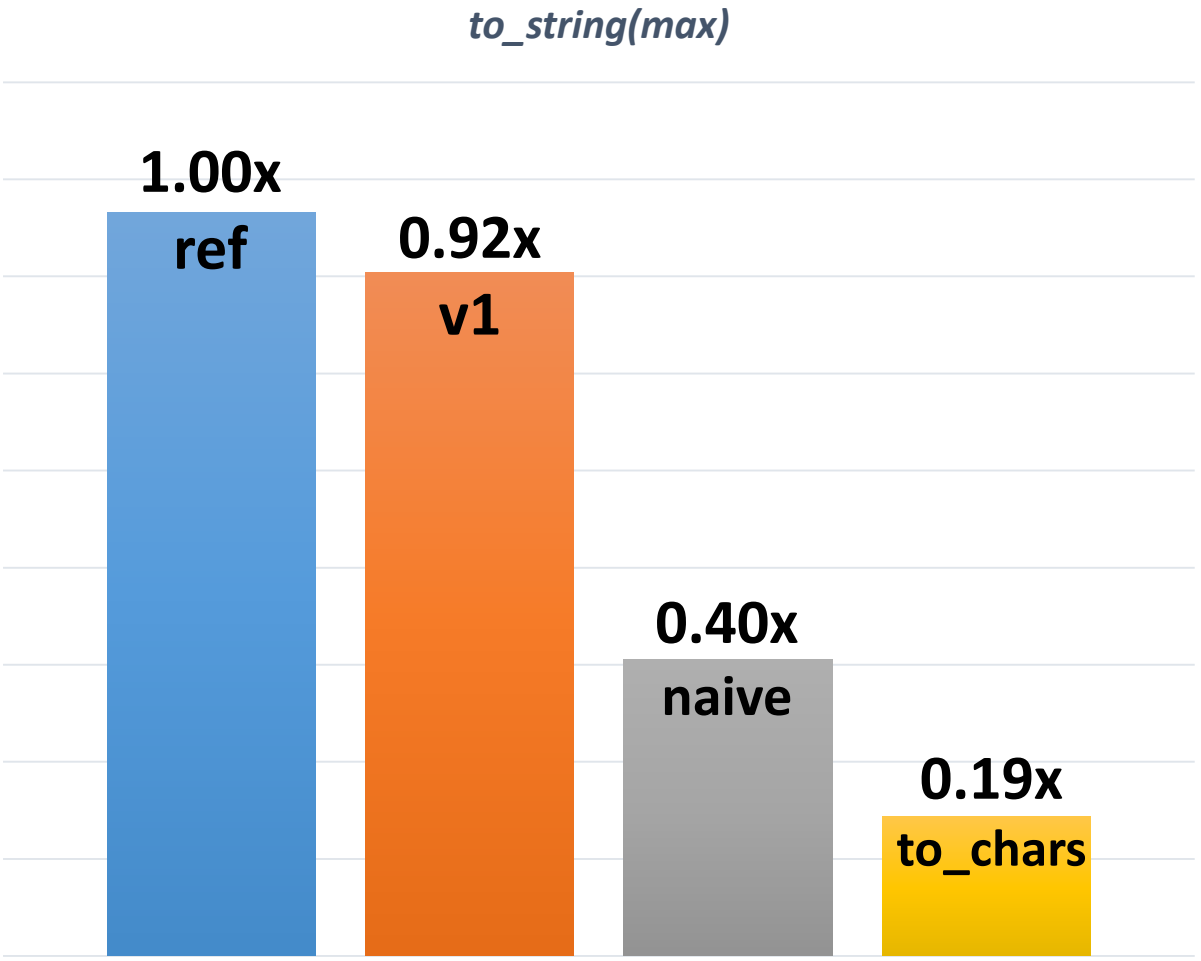
Proposal 3

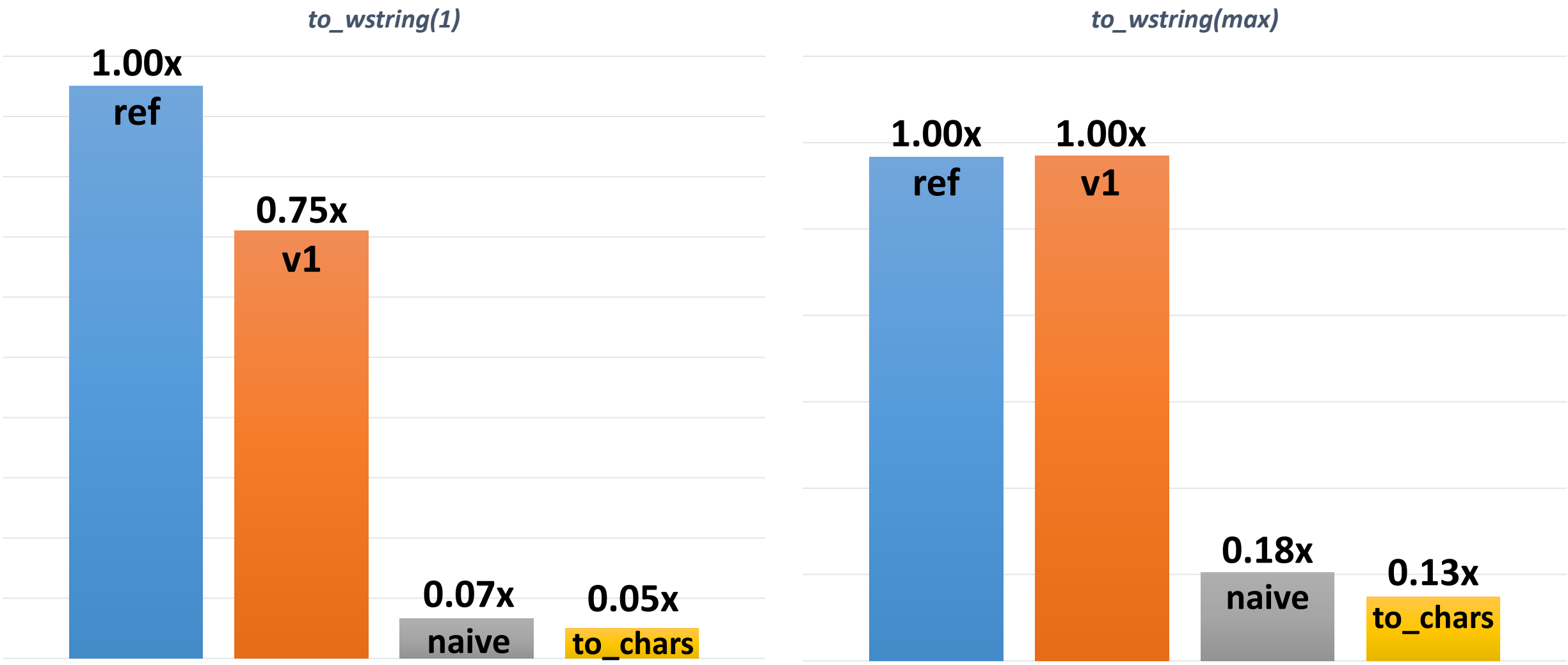
```
template <typename S, typename V>
S i_to_string(const V v)
{
  constexpr size_t bufsize = numeric_limits<V>::digits10 + 2;
  char buf[bufsize];
  const auto res = to_chars(buf, buf + bufsize, v);
  return S(buf, res.ptr);
}
```

The value of `std::numeric_limits<T>::digits10` is the number of base-10 digits that can be represented by the type `T` without change.

`std::numeric_limits<std::uint8_t>::digits10 == 2`:
any number in [0, 99] can be represented as `std::uint8_t` and [256, 999] can not.


to_string(1)
1.00x
ref
0.84x
v1
0.10x
naive
0.07x
to_chars


to_string(max)
1.00x
ref
0.92x
v1
0.40x
naive
0.19x
to_chars


to_wstring(1)
1.00x
ref
0.75x
v1
0.07x
naive
0.05x
to_chars
to_wstring(max)
1.00x
ref
1.00x
v1
0.18x
naive
0.13x
to_chars

std::to_chars

divisions count

- naïve algorithm executes 2 divisions per character

divisions count

- naïve algorithm executes 2 divisions per character
- who cares?

divisions count

- naïve algorithm executes 2 divisions per character
- who cares? *idiv* instruction cost:

arch	cost (cycles) 32 bit	cost (cycles) 64 bit

https://gmplib.org/~tege/x86-timing.pdf

divisions count

- naïve algorithm executes 2 divisions per character
- who cares? *idiv* instruction cost:

arch	cost (cycles) 32 bit	cost (cycles) 64 bit
Intel Core 2	40	116

https://gmplib.org/~tege/x86-timing.pdf

divisions count

- naïve algorithm executes 2 divisions per character
- who cares? *idiv* instruction cost:

arch	cost (cycles) 32 bit	cost (cycles) 64 bit
Intel Core 2	40	116
Intel Nehalem	26	89

https://gmplib.org/~tege/x86-timing.pdf

divisions count

- naïve algorithm executes 2 divisions per character
- who cares? *idiv* instruction cost:

arch	cost (cycles) 32 bit	cost (cycles) 64 bit
Intel Core 2	40	116
Intel Nehalem	26	89
Intel Sandy Bridge	26	92

https://gmplib.org/~tege/x86-timing.pdf

divisions count

- naïve algorithm executes 2 divisions per character
- who cares? *idiv* instruction cost:

arch	cost (cycles) 32 bit	cost (cycles) 64 bit
Intel Core 2	40	116
Intel Nehalem	26	89
Intel Sandy Bridge	26	92
AMD K10	45	77

https://gmplib.org/~tege/x86-timing.pdf

divisions count

- naïve algorithm executes 2 divisions per character
- who cares? *idiv* instruction cost:

arch	cost (cycles) 32 bit	cost (cycles) 64 bit
Intel Core 2	40	116
Intel Nehalem	26	89
Intel Sandy Bridge	26	92
AMD K10	45	77
Intel Atom	50	191

https://gmplib.org/~tege/x86-timing.pdf

divisions count

- naïve algorithm executes 2 divisions per character
- who cares? *idiv* instruction cost:

uint64_t numbers processing should fall into 32-bit arithmetic!

cost (cycles) 32 bit	cost (cycles) 64 bit
40	116
26	89
26	92
45	77
50	191

https://gmplib.org/~tege/x86-timing.pdf

divisions count

- naïve algorithm executes 2 divisions per character
- who cares? *idiv* instruction cost:

uint64_t numbers processing should fall into 32-bit arithmetic!

(actually, no, wait a bit)

cost (cycles) 32 bit	cost (cycles) 64 bit
40	116
26	89
26	92
45	77
50	191

https://gmplib.org/~tege/x86-timing.pdf

divisions count

- naïve algorithm executes 2 divisions per character
- *std::to_chars* executes 1 division per character

```
static char digitLuts[201] =
  "000102030405060708090"
  "101112131415161718190"
  "202122232425262728290"
  "303132333435363738390"
  "404142434445464748490"
  "505152535455565758590"
  "606162636465666768690"
  "707172737475767778790"
  "808182838485868788890"
  "909192939495969798990";
```

```
i = val % 100;

digitLuts[2 * i]
digitLuts[2 * i + 1]

val /= 100
```

divisions count

- naïve algorithm executes 2 divisions per character
- *std::to_chars* executes 1 division per character

?

(compiler)

divisions count

- naïve algorithm executes 2 divisions per character
- *std::to_chars* executes 1 division per character

```
using T = uint64_t;

T f(T x, T y)
{
  return x / y;
}


pair<T, T> g(T x, T y)
{
  return { x / y, x % y };
}
```

divisions count

- naïve algorithm executes 2 divisions per character
- *std::to_chars* executes 1 division per character

```
using T = uint64_t;

T f(T x, T y)
{
  return x / y;
}


pair<T, T> g(T x, T y)
{
  return { x / y, x % y };
}
```

```
f(unsigned long, unsigned long)
  mov     rax, rdi
  xor     edx, edx
  div     rsi
  ret

g(unsigned long, unsigned long)
  mov     rax, rdi
  xor     edx, edx
  div     rsi
  ret
```

clang-9.0, -O2, **x86**

divisions count

1

- naïve algorithm executes ~~2~~ divisions per character
- *std::to_chars* executes ~~1~~ division per character

0,5

divisions count

- naïve algorithm executes 1 division per character
- *std::to_chars* executes 0,5 divisions per character

integers division

```
int f(int x)
{
  return x / 10;
}
```

```
int g(int x, int y)
{
  return x / y;
}
```

integers division

```
int f(int x)
{
  return x / 10;
}
```

```
f(int) : # @f(int)
    movsxd  rax, edi
    imul    rax, rax, 1717986919
    mov     rcx, rax
    shr     rcx, 63
    sar     rax, 34
    add     eax, ecx
    ret
```

```
int g(int x, int y)
{
  return x / y;
}
```

```
g(int, int) : # @g(int, int)
    mov     eax, edi
    cdq
    idiv    esi
    ret
```

integers division

```
int f(int x)
{
  return x / 10;
}
```

```
f(int) : # @f(int)
    movsxd  rax, edi
    imul    rax, rax, 1717986919
    mov     rcx, rax
    shr     rcx, 63
    sar     rax, 34
    add     eax, ecx
    ret
```

```
int g(int x, int y)
{
  return x / y;
}
```

```
g(int, int) : # @g(int, int)
    mov     eax, edi
    cdq
    idiv    esi
    ret
```

integers division

```
int f(int x)
{
  return x / 10;
}
```

```
f(int) : # @f(int)
    movsxd  rax, edi
    imul    rax, rax, 1717986919
    mov     rcx, rax
    shr     rcx, 63
    sar     rax, 34
    add     eax, ecx
    ret
```

```
int g(int x, int y)
{
  return x / y;
}
```

```
g(int, int) : # @g(int, int)
    mov     eax, edi
    cdq
    idiv    esi
    ret
```

https://libdivide.com/

integers division

```
int f(int x)
{
  return x / 10;
}
```

```
f(int) : # @f(int)
    movsxd  rax, edi
    imul    rax, rax, 1717986919
    mov     rcx, rax
    shr     rcx, 63
    sar     rax, 34
    add     eax, ecx
    ret
```

```
int g(int x, int y)
{
  return x / y;
}
```

```
g(int, int) : # @g(int, int)
    mov     eax, edi
    cdq
    idiv    esi
    ret
```

https://lemire.me/blog/2019/02/08/faster-remainders-when-the-divisor-is-a-constant-beating-compilers-and-libdivide/

https://libdivide.com/

integers division

```
using T = std::uint64_t;

T f(T x)
{
  return x / 10;
}
```

```
pair<T, T> g(T x)
{
  return { x / 10, x % 10 };
}
```

integers division

```
using T = std::uint64_t;

T f(T x)
{
  return x / 10;
}
```

```
pair<T, T> g(T x)
{
  return { x / 10, x % 10 };
}
```

```
f(unsigned long) :
  mov     rax, rdi
  movabs  rcx, -3689348814741910323
  mul     rcx
  mov     rax, rdx
  shr     rax, 3
  ret
```

```
g(unsigned long) :
  movabs  rcx, -3689348814741910323
  mov     rax, rdi
  mul     rcx
  mov     rax, rdx
  shr     rax, 3
  lea     rcx, [rax + rax]
  lea     rcx, [rcx + 4 * rcx]
  sub     rdi, rcx
  mov     rdx, rdi
  ret
```

integers division

```
using T = std::uint64_t;

T f(T x)
{
  return x / 10;
}
```

```
f(unsigned long) :
  mov     rax, rdi
  movabs  rcx, -3689348814741910323
  mul     rcx
  mov     rax, rdx
  shr     rax, 3
  ret
```

```
pair<T, T> g(T x)
{
  return { x / 10, x % 10 };
}
```

```
g(unsigned long) :
  movabs  rcx, -3689348814741910323
  mov     rax, rdi
  mul     rcx
  mov     rax, rdx
  shr     rax, 3
  lea     rcx, [rax + rax]
  lea     rcx, [rcx + 4 * rcx]
  sub     rdi, rcx
  mov     rdx, rdi
  ret
```

divisions count

- naïve algorithm executes 1 division per character
- *std::to_chars* executes 0,5 divisions per character

divisions count

optimized integer “divmod” op

- naïve algorithm executes 1 ~~division~~ per character
- *std::to_chars* executes 0,5 ~~divisions~~ per character

optimized integer “divmod” op

divisions count

- naïve algorithm executes 1 optimized divmod per character
- *std::to_chars* executes 0,5 optimized divmod per character

length detection

```
if      (val < 10)
  ...
else if (val < 100)
  ...
else if (val < 1000)
  ...
else if (val < 10000)
  ...
else if (val < 100000)
  ...
else if (val < 1000000)
  ...
else if (val < 10000000)
```

memcpy

```
static char digitLuts[201] =
  "00010203040506070809"
  "10111213141516171819"
  "20212223242526272829"
  "30313233343536373839"
  "40414243444546474849"
  "50515253545556575859"
  "60616263646566676869"
  "70717273747576777879"
  "80818283848586878889"
  "90919293949596979899”;
```

```
char* append2(char* buffer, std::uint32_t i)
{
  std::memcpy(buffer, &digitLuts[(i) * 2], 2);
  return buffer + 2;
}
```

memcpy

```
static char digitLuts[201] =
  "00010203040506070809"
  "10111213141516171819"
  "20212223242526272829"
  "30313233343536373839"
  "40414243444546474849"
  "50515253545556575859"
  "60616263646566676869"
  "70717273747576777879"
  "80818283848586878889"
  "90919293949596979899”;
```

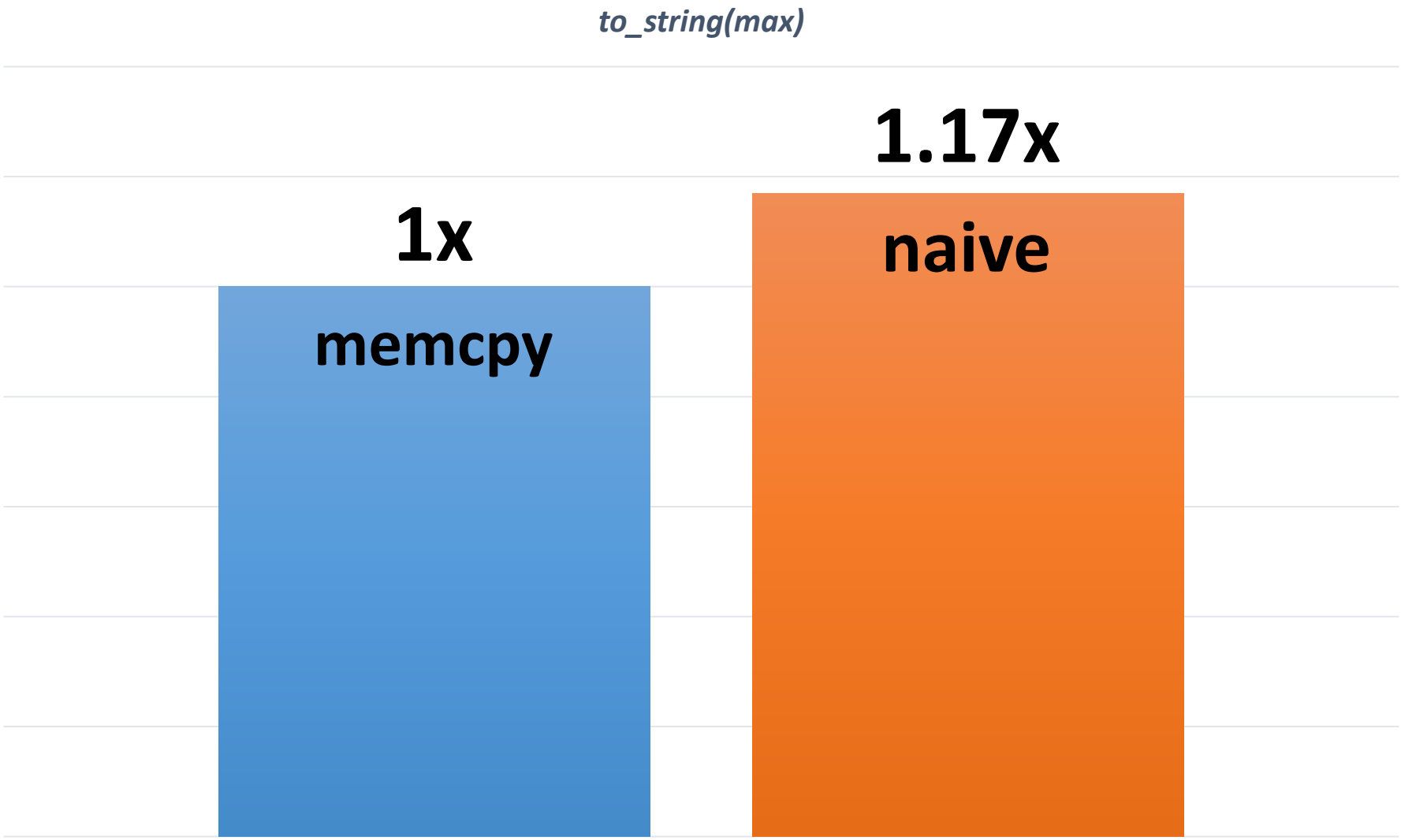
```
char* append2(char* buffer, std::uint32_t i)
{
  std::memcpy(buffer, &digitLuts[(i) * 2], 2);
  return buffer + 2;
}
```

memcpy

```
static char digitLuts[201] =
  "00010203040506070809"
  "10111213141516171819"
  "20212223242526272829"
  "30313233343536373839"
  "40414243444546474849"
  "50515253545556575859"
  "60616263646566676869"
  "70717273747576777879"
  "80818283848586878889"
  "90919293949596979899”;
```

```
char* append2(char* buffer, std::uint32_t i)
{
  std::memcpy(buffer, &digitLuts[(i) * 2], 2);
  return buffer + 2;
}
```

```
char* append2(char* buffer, std::uint32_t i)
{
  *buffer = digitLuts[i * 2];
  *(buffer + 1) = digitLuts[i * 2 + 1];
  return buffer + 2;
}
```


to_string(max)
1x
memcpy
1.17x
naive

memcpy

```
namespace ref
{
  char* append2(char* buffer, std::uint32_t i)
  {
    std::memcpy(buffer, &cDigitsLut[(i) * 2], 2);
    return buffer + 2;
  }
}
```

```
namespace tgt
{
  char* append2(char* buffer, std::uint32_t i)
  {
    *buffer = cDigitsLut[i * 2];
    *(buffer + 1) = cDigitsLut[i * 2 + 1];
    return buffer + 2;
  }
}
```

memcpy

```
namespace ref
{
  char* append2(char* buffer, std::uint32_t i)
  {
    std::memcpy(buffer, &cDigitsLut[(i) * 2], 2);
    return buffer + 2;
  }
}
```

```
ref::append2(char*, unsigned int)
    add     esi, esi
    movzx   eax, word ptr[rsi + cDigitsLut]
    mov     word ptr[rdi], ax
    lea     rax, [rdi + 2]
    ret
```

```
namespace tgt
{
  char* append2(char* buffer, std::uint32_t i)
  {
    *buffer = cDigitsLut[i * 2];
    *(buffer + 1) = cDigitsLut[i * 2 + 1];
    return buffer + 2;
  }
}
```

```
tgt::append2(char*, unsigned int)
    add     esi, esi
    mov     al, byte ptr[rsi + cDigitsLut]
    mov     byte ptr[rdi], al
    mov     al, byte ptr[rsi + cDigitsLut + 1]
    mov     byte ptr[rdi + 1], al
    lea     rax, [rdi + 2]
    ret
```

memcpy

```
namespace ref
{
  char* append2(char* buffer, std::uint32_t i)
  {
    std::memcpy(buffer, &cDigitsLut[(i) * 2], 2);
    return buffer + 2;
  }
}
```

```
ref::append2(char*, unsigned int)
    add     esi, esi
    movzx   eax, word ptr[rsi + cDigitsLut]
    mov     word ptr[rdi], ax
    lea     rax, [rdi + 2]
    ret
```

```
namespace tgt
{
  char* append2(char* buffer, std::uint32_t i)
  {
    *buffer = cDigitsLut[i * 2];
    *(buffer + 1) = cDigitsLut[i * 2 + 1];
    return buffer + 2;
  }
}
```

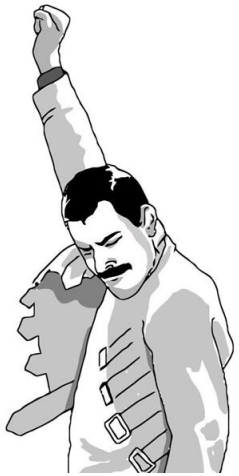
```
tgt::append2(char*, unsigned int)
    add     esi, esi
    mov     al, byte ptr[rsi + cDigitsLut]
    mov     byte ptr[rdi], al
    mov     al, byte ptr[rsi + cDigitsLut + 1]
    mov     byte ptr[rdi + 1], al
    lea     rax, [rdi + 2]
    ret
```

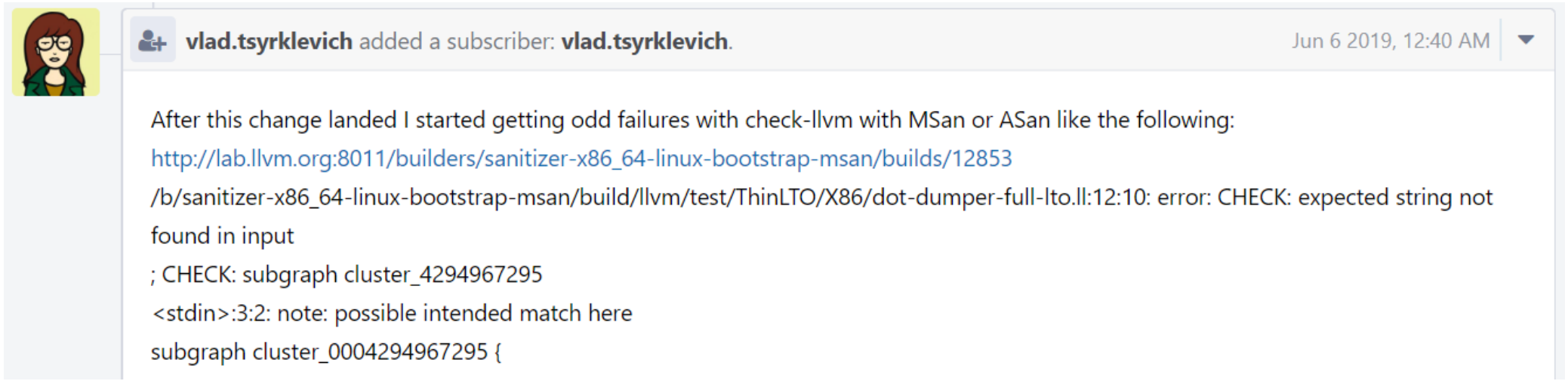
95% speedup… seems nice?

2 months later…

MERGED!

MERGED!

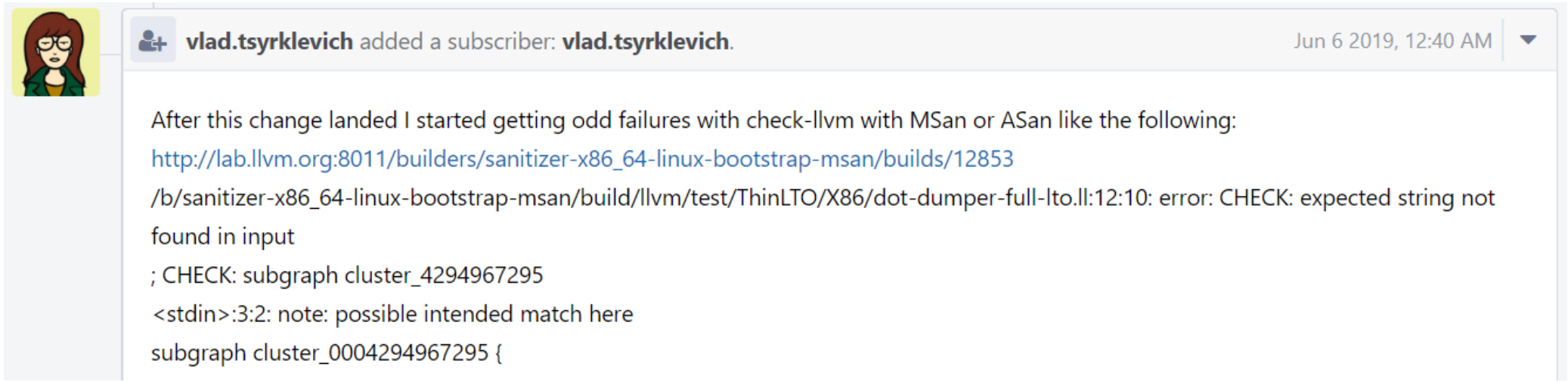




vlad.tsyrklevich added a subscriber: **vlad.tsyrklevich**. Jun 6 2019, 12:40 AM

After this change landed I started getting odd failures with check-llvm with MSan or ASan like the following:
http://lab.llvm.org:8011/builders/sanitizer-x86_64-linux-bootstrap-msan/builds/12853
/b/sanitizer-x86_64-linux-bootstrap-msan/build/llvm/test/ThinLTO/X86/dot-dumper-full-lto.ll:12:10: error: CHECK: expected string not found in input
; CHECK: subgraph cluster_4294967295
<stdin>:3:2: note: possible intended match here
subgraph cluster_0004294967295 {

… and reverted in 5 hrs

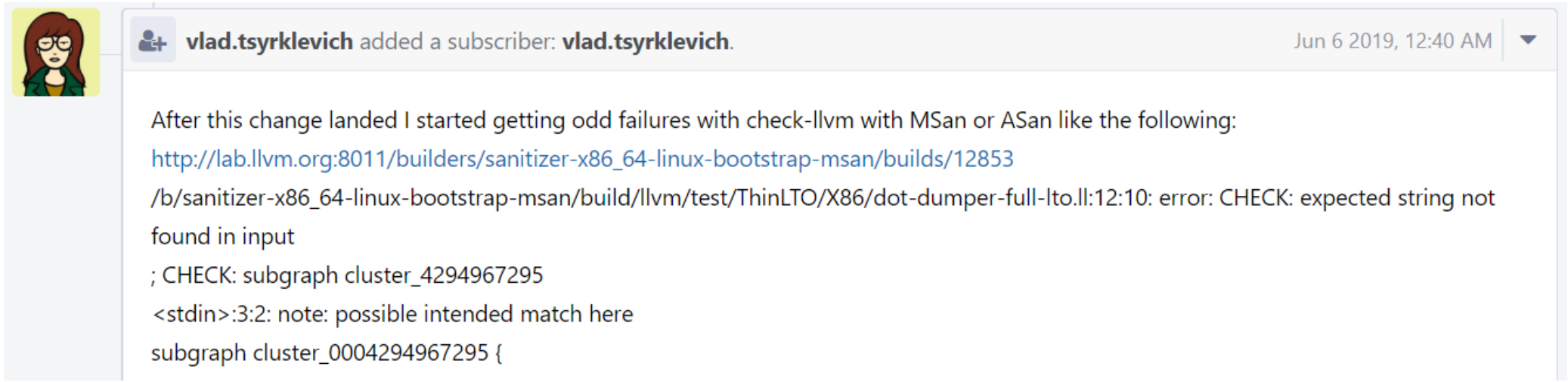


vlad.tsyrklevich added a subscriber: **vlad.tsyrklevich**. Jun 6 2019, 12:40 AM

After this change landed I started getting odd failures with check-llvm with MSan or ASan like the following:
http://lab.llvm.org:8011/builders/sanitizer-x86_64-linux-bootstrap-msan/builds/12853
/b/sanitizer-x86_64-linux-bootstrap-msan/build/llvm/test/ThinLTO/X86/dot-dumper-full-lto.ll:12:10: error: CHECK: expected string not found in input
; CHECK: subgraph cluster_4294967295
<stdin>:3:2: note: possible intended match here
subgraph cluster_0004294967295 {

… and reverted in 5 hrs:

clang checks failed in asan/msan mode



… and reverted in 5 hrs:

clang checks failed in asan/msan mode

to_string((uint64_t)0xffffffff) == "0004294967295"

leading zeros problem

std::to_chars (uint64_t) adds redundant leading zeros for specific range of values

leading zeros problem

std::to_chars (uint64_t) adds redundant leading zeros for specific range of values

Converts `value` into a character string by successively filling the range `[first, last)`, where `[first, last)` is required to be a valid range.

1) Integer formatters: `value` is converted to a string of digits in the given base (with no redundant leading zeroes). Digits in the range `10..35` (inclusive) are represented as lowercase characters `a..z`. If `value` is less than zero, the representation starts with a minus sign. The library provides overloads for all signed and unsigned integer types and for the type `char` as the type of the parameter `value`.

leading zeros problem

std::to_chars (uint64_t) adds redundant leading zeros for specific range of values

Converts `value` into a character string by successively filling the range `[first, last)`, where `[first, last)` is required to be a valid range.

1) Integer formatters: `value` is converted to a string of digits in the given base (with no redundant leading zeroes). Digits in the range `10..35` (inclusive) are represented as lowercase characters `a..z`. If `value` is less than zero, the representation starts with a minus sign. The library provides overloads for all signed and unsigned integer types and for the type `char` as the type of the parameter `value`.

to_chars can puts leading zeros on numbers in $[10^9, 10^{12})$

https://bugs.llvm.org/show_bug.cgi?id=42166

leading zeros problem

std::to_chars (uint64_t) adds redundant leading zeros for specific range of values

Converts `value` into a character string by successively filling the range `[first, last)`, where `[first, last)` is required to be a valid range.

1) Integer formatters: `value` is converted to a string of digits in the given base (with no redundant leading zeroes). Digits in the range `10..35` (inclusive) are represented as lowercase characters `a..z`. If value is less than zero, the representation starts with a minus sign. The library provides overloads for all signed and unsigned integer types and for the type `char` as the type of the parameter `value`.

to_chars can puts leading zeros on numbers in $[10^9, 10^{12})$
https://bugs.llvm.org/show_bug.cgi?id=42166

fix leading zeros in *std::to_chars*
https://reviews.llvm.org/D63047

MERGED!

floating point numbers

```
std::string to_string( float value );                (7)   (since C++11)
std::string to_string( double value );               (8)   (since C++11)
std::string to_string( long double value );          (9)   (since C++11)
```

floating point numbers

```
std::string to_string( float value );                (7)   (since C++11)
std::string to_string( double value );               (8)   (since C++11)
std::string to_string( long double value );          (9)   (since C++11)
```

7,8) Converts a floating point value to a string with the same content as what `std::sprintf(buf, "%f", value)` would produce for sufficiently large `buf`.

9) Converts a floating point value to a string with the same content as what `std::sprintf(buf, "%Lf", value)` would produce for sufficiently large `buf`.

floating point numbers

```
std::string to_string( float value );          (7)  (since C++11)
std::string to_string( double value );         (8)  (since C++11)
std::string to_string( long double value );    (9)  (since C++11)
```

7,8) Converts a floating point value to a string with the same content as what `std::sprintf(buf, "%f", value)` would produce for sufficiently large `buf`.

9) Converts a floating point value to a string with the same content as what `std::sprintf(buf, "%Lf", value)` would produce for sufficiently large `buf`.

Problems:

- *sprintf* depends on locale, *to_chars* is locale independent!
- *to_chars* for floating point numbers is not implemented yet
- *to_chars* guarantees precise value recovery, *to_string* does not

floating point numbers

```
std::string to_string( float value );          (7)  (since C++11)
std::string to_string( double value );         (8)  (since C++11)
std::string to_string( long double value );    (9)  (since C++11)
```

7,8) Converts a floating point value to a string with the same content as what `std::sprintf(buf, "%f", value)` would produce for sufficiently large `buf`.

9) Converts a floating point value to a string with the same content as what `std::sprintf(buf, "%Lf", value)` would produce for sufficiently large `buf`.

Problems:

- *sprintf* depends on locale, *to_chars* is locale independent!
- *to_chars* for floating point numbers is not implemented yet
- *to_chars* guarantees precise value recovery, *to_string* does not

use proposal 1 to speedup *std::to_string* for floating point numbers (1.0x – 5.2x):

https://reviews.llvm.org/D64341

floating point numbers

```
std::string to_string( float value );          (7)  (since C++11)
std::string to_string( double value );         (8)  (since C++11)
std::string to_string( long double value );    (9)  (since C++11)
```

7,8) Converts a floating point value to a string with the same content as what `std::sprintf(buf, "%f", value)` would produce for sufficiently large `buf`.

9) Converts a floating point value to a string with the same content as what `std::sprintf(buf, "%Lf", value)` would produce for sufficiently large `buf`.

Problems:

- *sprintf* depends on locale, *to_chars* is locale independent!
- *to_chars* for floating point numbers is not implemented yet
- *to_chars* guarantees precise value recovery, *to_string* does not

use proposal 1 to speedup *std::to_string* for floating point numbers (1.0x – 5.2x):

https://reviews.llvm.org/D64341

Rejected

libstdc++(GNU) and MS STL

	then	now (23 nov. 2019)
libstdc++(GNU)		
MS STL		

libstdc++(GNU) and MS STL

	then	now (23 nov. 2019)
libstdc++(GNU)	proposal 1 success path (*sprintf*)	
MS STL	-	

libstdc++(GNU) and MS STL

	then	now (23 nov. 2019)
libstdc++(GNU)	proposal 1 success path (*sprintf*)	*to_chars*
MS STL	-	naïve algorithm

Results

- *to_string* / *to_wstring* performance improved up to 20x times
  https://reviews.llvm.org/D59178

- fixed leading zeros in std::to_chars
  https://reviews.llvm.org/D63047

Thank you