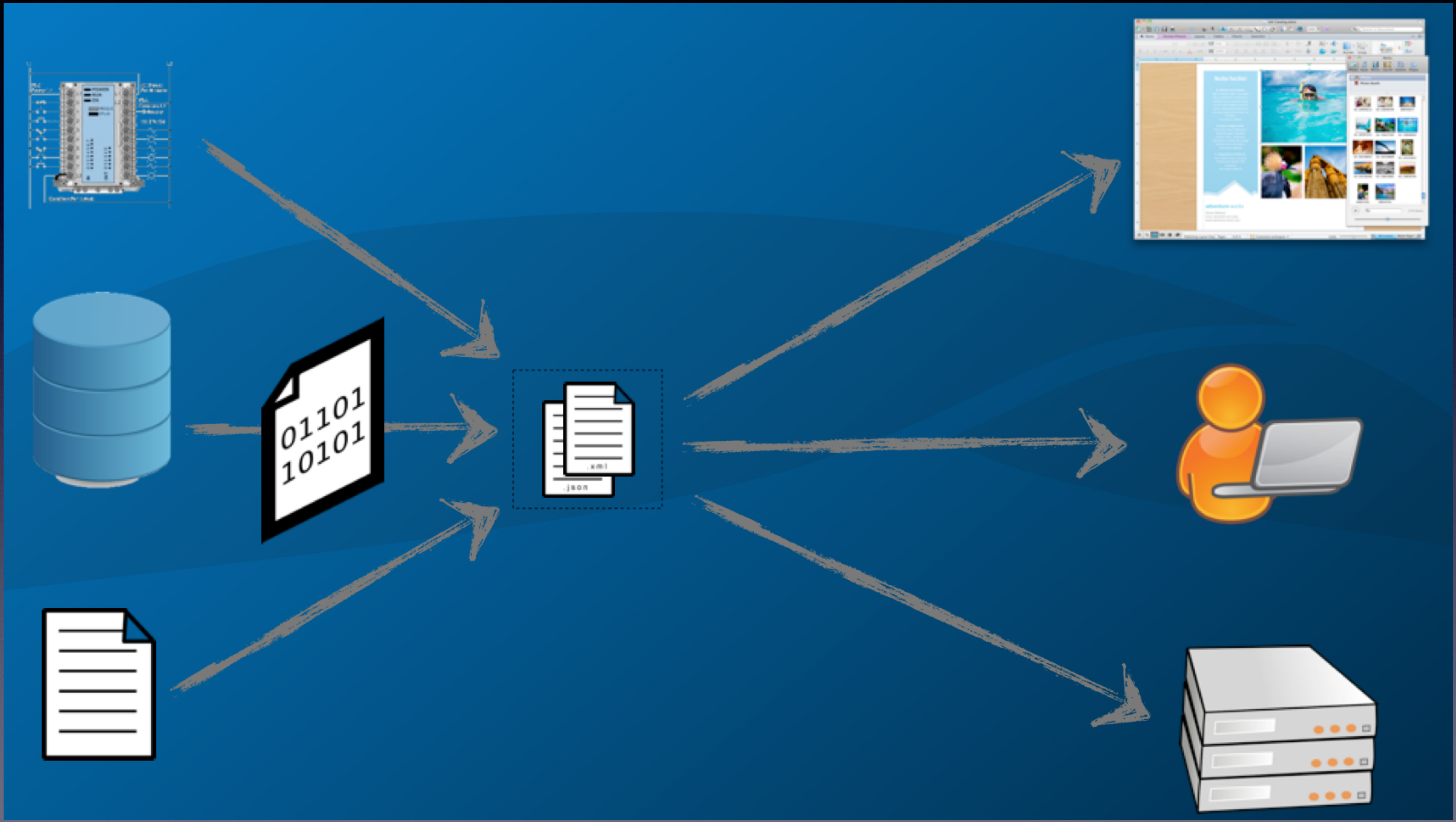
Dynamic C++ Performance

Alex Fabijanic
alex@pocoproject.org

"As to which is more important, Dynamic or static, both are absolutely essential, even when they are in conflict."

-- Robert Pirsig, Metaphysics of Quality

The Problem



Dynamic C++ Concerns

- storing value
- performing operations (mostly conversions)
- retrieving value
- runtime performance
  - speed
  - memory usage
- accuracy
- code size
- ease of use

Dynamic Data Storage Strategies

- heap[1] (void* + new)
  - allocation overhead
  - memory cleanup
- stack[1] (union[2] + placement new)
  - size
  - alignment
  - ~destruction
- hybrid (a.k.a. small object optimization)
  - runtime detection performance penalty

---

[1] Commonly used nomenclature – internal/external would be more appropriate
[2] Usually raw storage and alignment (plus pointer for SOO)

Dynamic Data Operations Support

- type conversions
  - static[1]
  - dynamic
- standard language operations (+, -, ==, ...)
- custom operations

---

[1] Delegated to compiler, with runtime narrowing check

boost::any

- a container for values of any type
- does not attempt conversion between types
- accommodates any type in a single container
- generic solution for the first half of the problem
- “syntactic sugar” – template without template syntax

```
class any
{
 public:
  template<typename T>
  any(const T& value):content(new holder<T>(value)) { }
};

any a = "123";
any b = 123;
```

boost::any – under the hood

```
class any {
  template<typename T>
  any(const T& value):content(new holder<T>(value)){ }
  // ...
  struct placeholder
  {
    virtual ~placeholder(){ }
    // ...
    virtual const std::type_info & type() const = 0;
    // ...
  };

  template<typename T>
  struct holder : public placeholder
  {
    holder(const T& value):held(value){ }
    // ...
    T held;
  };
};
```

boost::any – summary

> generic container for values of different types

> simple to understand and use

> useful when sender and receiver (or caller and callee) know exactly what to expect but the “middle man” does not

> dynamic receiving

> static giving (stricter than language)

> heap alloc overhead

> virtual call overhead

how expensive is boost::any ?

```
const size_t n = 100000000;
volatile size_t x;

void anyfunc(const boost::any& value) {
  x = boost::any_cast<size_t>(value);
}

void holdany(const boost::spirit::hold_any& value) {
  x = boost::spirit::any_cast<size_t>(value);
}

void voidptrinnerfunc(void * val) {
  x = *(static_cast<size_t*>(val));
}

template<typename T> void voidptr(T t) {
  voidptrinnerfunc(static_cast<void*>(&t));
}

for (size_t i = 0; i < n; ++i) anyfunc( i );
for (size_t i = 0; i < n; ++i) holdany(hold_any(i));
for (size_t i = 0; i < n; ++i) voidptr(i);
```

boost::any is expensive

METHOD	g++	g++ -O1	g++ -O2	g++ -O3
boost::any	7.29s	3.64s	3.63s	3.21
boost::spirit::hold_any	2.93s	0.93s	0.66s	0.40
void*	0.49s	0.32s	0.05s	0.05

http://felipedelamuerte.wordpress.com/2012/04/06/why-you-shouldnt-use-boostany-especially-not-in-time-critical-code/

boost::spirit::hold_any

```
template <typename T>
struct get_table
{
  typedef mpl::bool_<(sizeof(T) <= sizeof(void*))> is_small;

  // ...
};

template <typename T>
explicit basic_hold_any(T const& x):
table(spirit::detail::get_table<T>::template get<Char>()), object(0)
{
   if (spirit::detail::get_table<T>::is_small::value)
      new (&object) T(x);
   else
      object = new T(x);
}
```

boost::variant

- "safe, generic, stack-based discriminated union container, offering a simple solution for manipulating an object from a heterogeneous set of types in a uniform manner"
- can hold any type specified at compile time
- default construction (first type must be default-constructible)

```
boost::variant<int, std::string> v;

v = "hello";

std::cout << v << std::endl; // "hello"

std::string& str = boost::get<std::string>(v);

str += " world! "; // "hello world!"
```

boost: variant visitors welcome

> supports compile-time checked visitation

```
// conversion via visitation
struct str_int_converter : public static_visitor<int>
{
    int operator()(int i) const
    {
        return i;
    }

    int operator()(const std::string & str) const
    {
        return NumberParser::parse(str);
    }
};


variant<int, std::string> u("123");
int i = apply_visitor(str_int_converter(), u); // i == 123
```

boost::variant construction

```
variant()
{
  new( storage_.address() ) internal_T0();
  indicate_which(0); // index of the first bounded type
}

template <typename T>
variant(const T& operand)
{
   convert_construct(operand, 1L);
}

template <typename T> void convert_construct(
          T& operand
        , int
        , mpl::false_ = mpl::false_() //true for another variant
        )
{
  indicate_which(initializer::initialize(
    storage_.address(), operand));
}
```

boost::variant behavior

- provides “never empty” guarantee
- assignment from one type to another may incur dynamic allocation
- it is a variant storage, but not a convenient conversion facility

boost::type_erasure - valued conversion

```
int i = 123;

any<...> s(i);

std::string str = s.to_string();
```

boost::type_erasure – conversion scaffolding

```
template<class F, class T>
struct to_string
{
    // conversion function
    static T apply(const F& from, T& to)
    {
        return to = NumberFormatter::format(from);
    }
};

namespace boost {
namespace type_erasure {
    template<class F, class T, class Base> // binding
    struct concept_interface<::to_string<F, T>, Base, F> : Base
    {
        typedef typename rebind_any<Base, T>::type IntType;
        T to_string(IntType arg = IntType())
        {
            return call(::to_string<C, T>(), *this, arg);
        }
    };
}
```

boost::type_erasure – valued conversion

```
typedef any<to_string<_self, std::string>, _self&> stringable;
int i = 123;
stringable s(i);
std::string str = s.to_string();
```

boost::type_erasure – summary

- extends any/variant by introducing a mechanism for "attaching" operations to types at compile time
- addresses the limitations of virtual functions
- uses heap allocation to store values
- complex framework
- slippery scenarios can easily take a naïve user into UB-land
- like any and variant, it is not an out-of-the-box value conversion facility
- could be a good foundation to build on

QVariant – design guidelines

Design Constraints:

- not template class.

- must work for both POD and non-POD data types.

- must be able to store custom user types.

- not using C++ RTTI.

QVariant – internals (storage)

```
struct Private {
  union Data {
    char c;
    uchar uc;
    short s;
    signed char sc;
    ushort us;
    int i;
    uint u;
    long l;
    ulong ul;
    bool b;
    double d;
    float f;
    qreal real;
    qlonglong ll;
    qulonglong ull;
    QObject *o;
    void *ptr;
    PrivateShared *shared;
} data;
    uint type : 30;
    uint is_shared : 1;
    uint is_null : 1;
};
```

QVariant – internals (operations)

```
// ...
typedef void (*f_construct)(Private *, const void *);
    typedef void (*f_clear)(Private *);
    typedef bool (*f_null)(const Private *);
    typedef bool (*f_compare)(const Private *, const Private *);
    typedef bool (*f_convert)(const QVariant::Private *d, int t,
void *, bool *);
    typedef bool (*f_canConvert)(const QVariant::Private *d, int t);
    typedef void (*f_debugStream)(QDebug, const QVariant &);
    struct Handler {
    f_construct construct;
    f_clear clear;
    f_null isNull;
    f_compare compare;
    f_convert convert;
    f_canConvert canConvert;
    f_debugStream debugStream;
};
```

QVariant – type registry

```
qRegisterMetaType<MyStruct>("MyStruct");
QMetaType::registerType("MyStruct", ctr, dtr);
//Internally stored in a QVector<QCustomTypeInfo>
```

Facebook folly::dynamic

- “runtime dynamically typed value for C++, similar to the way languages with runtime type systems work”
- holds types from predefined set of types
- supports “objects” and “arrays” (JSON was the motivation)
- union-based, implementation reminiscent of boost::variant
- runtime operation checking
- linux-only

```
dynamic twelve = 12; // dynamic holding an int
dynamic str = "string"; // FBString
dynamic nul = nullptr;
dynamic boolean = false;
```

folly::dynamic – usage

> clean, intuitive and reasonably predictable interface

```
std::string str("123");
dynamic var(str);
int i = var.asInt(); // 123
double d = var.asDouble(); // 123.0
var = i; // 123
std::string s = var.asString().c_str(); // FBString!
var = d; // 123
s = var.asString().c_str(); // "123.0"
```

folly::dynamic – storage

> somewhat similar to boost::variant and QVariant

```
union Data
{
  explicit Data() : nul(nullptr) {}

  void* nul;
  Array array;
  bool boolean;
  double doubl;
  int64_t integer;
  fbstring string;

  typename std::aligned_storage<
    sizeof(std::unordered_map<int,int>),
    alignof(std::unordered_map<int,int>)
  >::type objectBuffer;
} u_;
```

folly::dynamic – get stored type

```
template<class T>
T dynamic::asImpl() const
{
  switch (type())
  {
  case INT64:     return to<T>(*get_nothrow<int64_t>());
  case DOUBLE:    return to<T>(*get_nothrow<double>());
  case BOOL:      return to<T>(*get_nothrow<bool>());
  case STRING:    return to<T>(*get_nothrow<fbstring>());
  default:
    throw TypeError("int/double/bool/string", type());
  }
}

inline fbstring dynamic::asString() const
{
  return asImpl<fbstring>();
}
```

folly::dynamic – get stored type

```
template<> struct dynamic::GetAddrImpl<bool> {
  static bool* get(Data& d) { return &d.boolean; }
};

template<> struct dynamic::GetAddrImpl<int64_t> {
  static int64_t* get(Data& d) { return &d.integer; }
};

template<class T>
T* dynamic::getAddress() {
  return GetAddrImpl<T>::get(u_);
}

template<class T>
T* dynamic::get_nothrow() {
  if (type_ != TypeInfo<T>::type) {
    return nullptr;
  }
  return getAddress<T>();
}
```

folly::dynamic – conversion

- to<>() functions in “folly/Conv.h”
- custom algorithm for integer conversions
- V8 double-conversion for floating point conversions

folly::dynamic summary

- built around Facebook's JSON needs
- uses C++11 and boost extensively
- performs very well (excellent design/performance balance)
- good example of user-friendly interface
- not portable (optimized for Linux/g++)
- holding only predefined types

Poco::Dynamic::Var (ex DynamicAny)

- boost::any + value conversion
- aims to be a general-purpose dynamic typing facility
- balance between performance and ease of use
- transparent conversions
- conversion is checked (e.g. no signedness loss or narrowing conversion loss of precision)
- any type, extensible for UDT through VarHolder specialization
- optional small object optimization (experimental)

Poco::Dynamic::Var – under the hood

```
namespace Poco {
namespace Dynamic {

class Var
{
public:
    // ...
    template <typename T>
    Var(const T& val):
        _pHolder(new VarHolderImpl<T>(val))
    {
    }

    // ...
private:
    VarHolder* _pHolder;
};

} }
```

* Design based on boost::any

Poco::Dynamic::Var – Small Object Optimization

```
template<typename ValueType>
void construct(const ValueType& value) {
 if (sizeof(Holder<ValueType>) <= Placeholder<ValueType>::Size::value)
 {
   new
  (reinterpret_cast<ValueHolder*>(_valueHolder.holder))
   Holder<ValueType>(value);
   _valueHolder.setLocal(true);
 }
 else {
   _valueHolder.pHolder = new Holder<ValueType>(value);
   _valueHolder.setLocal(false);
 }
}

void construct(const Any& other) {
  if(!other.empty())
    other.content()->clone(&_valueHolder);
  else
    _valueHolder.erase();
}

void destruct() {
  content()->~ValueHolder();
}

Placeholder<ValueHolder> _valueHolder;
```

Poco::Dynamic::Var – under the hood

```
namespace Poco {
namespace Dynamic {

class VarHolder {
public:
    virtual ~VarHolder();
    virtual void convert(int& val) const;
    // ...
protected:
    VarHolder();
    // ...
};

template <typename T> // for end-user extensions
class VarHolderImpl: public VarHolder {
    //...
};

template <> // native and frequent types specializations courtesy of POCO
class VarHolderImpl<int>: public VarHolder {
    //...
};

//...
}}
```

Poco::Dynamic::Var – checked narrowing

```
template <typename F, typename T>
void convertToSmallerUnsigned(const F& from, T& to) const
    /// This function is meant for converting unsigned integral data types,
    /// from larger to smaller type. Since lower limit is always 0 for unsigned
    /// types, only the upper limit is checked, thus saving some cycles
    /// compared to the signed version of the function. If the value to be
    /// converted does not exceed the maximum value for the target type,
    /// the conversion is performed.
{
    poco_static_assert (std::numeric_limits<F>::is_specialized);
    poco_static_assert (std::numeric_limits<T>::is_specialized);
    poco_static_assert (!std::numeric_limits<F>::is_signed);
    poco_static_assert (!std::numeric_limits<T>::is_signed);

    checkUpperLimit<F,T>(from);
    to = static_cast<T>(from);
}
```

Poco::Dynamic::Var – to/from string

```
template <typename T>
class VarHolderImpl<std::basic_string<T> >: public VarHolder
{
public:
// ...
     void convert(Int16& val) const
     {
          int v = NumberParser::parse(_val); // uses internal conversion routine
          convertToSmaller(v, val);
     }

     void convert(double& val) const
     {
          int v = NumberParser::parseFloat(_val); // uses V8 double-conversion
     }
};

template <>
class VarHolderImpl<double>: public VarHolder
{
public:
//...
     void convert(std::string& val) const
     {
          val = NumberFormatter::format(_val);
     }
};
```

Poco::Dynamic::Var – int to string

```
template <typename T>
bool intToStr(T value, char* result)
{
     uint32_t const size = digits10(value);
     //Impl::Ptr ptr(result, size);
     char* ptr = result;
     T tmpVal;
     do
     {
          tmpVal = value;
          value /= 10;
          *ptr++ = "9876543210123456789"[9 + (tmpVal - value * 10)];
     } while (value);

     if (tmpVal < 0) *ptr++ = '-';

     *ptr-- = '\0';

     char* ptrr = result;
     char tmp;
     while (ptrr < ptr)
     {
          tmp = *ptr;
          *ptr-- = *ptrr;
          *ptrr++ = tmp;
     }

     return true;
}
```

Being too clever

Pay the price

folly::dynamic – int to string

> current code

```
inline uint32_t uint64ToBufferUnsafe(uint64_t v, char *const buffer)
{
  auto const result = digits10(v);
  // WARNING: using size_t or pointer arithmetic for pos slows down
  // the loop below 20x. This is because several 32-bit ops can be
  // done in parallel, but only fewer 64-bit ones.
  uint32_t pos = result - 1;
  while (v >= 10) {
   auto const q = v / 10;
   auto const r = static_cast<uint32_t>(v % 10);
   buffer[pos--] = '0' + r;
   v = q;
  }
  // Last digit is trivial to handle
  buffer[pos] = static_cast<uint32_t>(v) + '0';
  return result;
}
```

folly::dynamic – # of digits

```
#define LIKELY(x)   (__builtin_expect((x), 1))*

inline uint32_t digits10(uint64_t v) {
  uint32_t result = 1;
  for (;;) {
    if (LIKELY(v < 10)) return result;
    if (LIKELY(v < 100)) return result + 1;
    if (LIKELY(v < 1000)) return result + 2;
    if (LIKELY(v < 10000)) return result + 3;
    // Skip ahead by 4 orders of magnitude
    v /= 10000U;
    result += 4;
  }
}
```

*Since "if" jumps can invalidate what's in the CPU's instruction pipeline, **__builtin_expect** allows gcc to try to assemble the code so the likely scenario involves fewer jumps than the alternate.

"Three Optimization Tips for C++"

- by Andrei Alexandrescu
- count digits by replacing division with comparison, favoring small numbers

```
uint32_t digits10(uint64_t v) {
  if (v < P01) return 1;
  if (v < P02) return 2;
  if (v < P03) return 3;
  if (v < P12) {
    if (v < P08) {
      if (v < P06) {
        if (v < P04) return 4;
        return 5 + (v >= P05);
      }
      return 7 + (v >= P07);
    }
    if (v < P10) { return 9 + (v >= P09); }
    return 11 + (v >= P11);
  }
  return 12 + digits10(v / P12);
}
```

“Three Optimization Tips for C++”

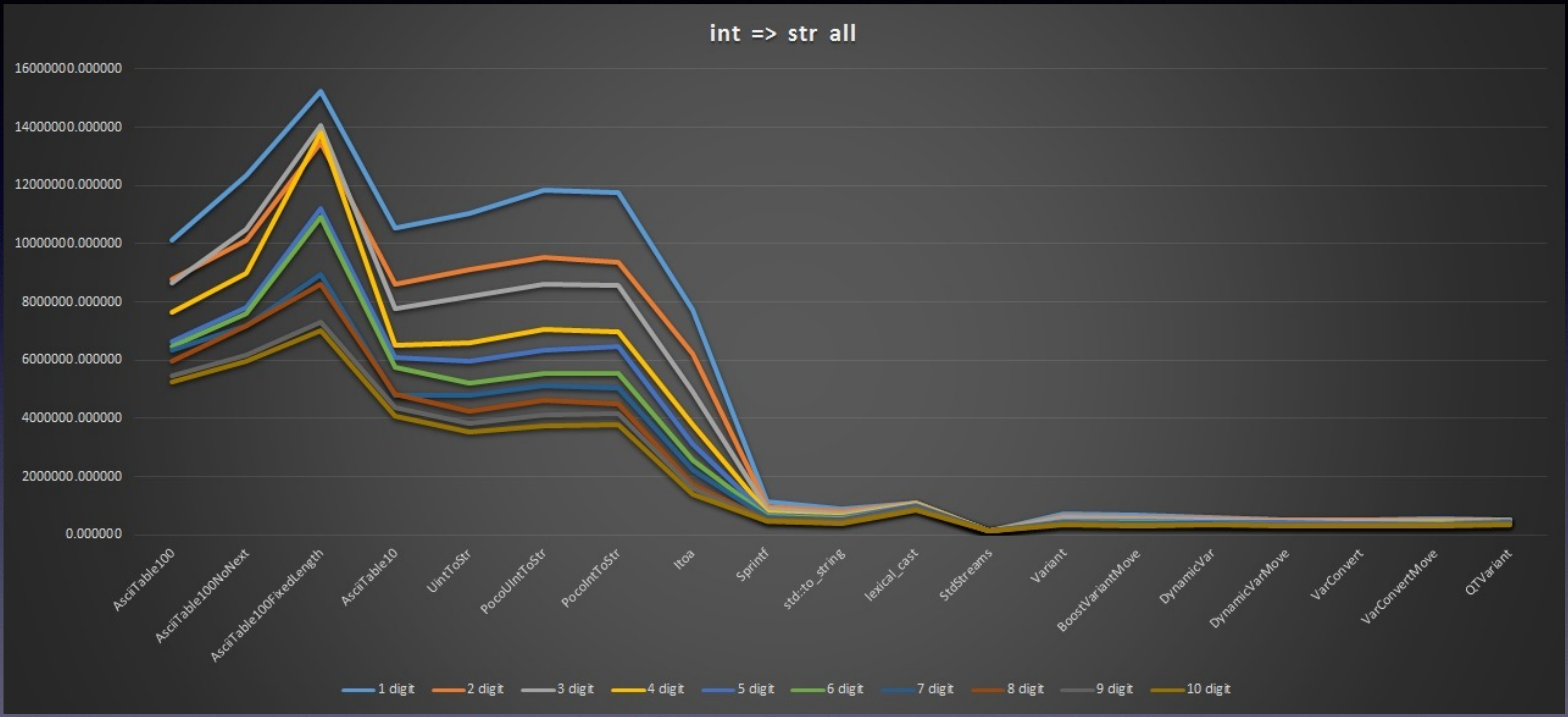
```
unsigned u64ToAsciiTable(uint64_t value, char* dst) {
  static const char digits[201] =
    "0001020304050607080910111213141516171819"
    "2021222324252627282930313233343536373839"
    "4041424344454647484950515253545556575859"
    "6061626364656667686970717273747576777879"
    "8081828384858687888990919293949596979899";
  uint32_t const length = digits10(value);
  uint32_t next = length - 1;
  while (value >= 100) {
    auto const i = (value % 100) * 2;
    value /= 100;
    dst[next] = digits[i + 1];
    dst[next - 1] = digits[i];
    next -= 2;
  }
  // Handle last 1-2 digits
  if (value < 10) {
    dst[next] = '0' + uint32_t(value);
  } else {
    auto i = uint32_t(value) * 2;
    dst[next] = digits[i + 1];
    dst[next - 1] = digits[i];
  }
  return length;
}
```

Use pointer as counter?

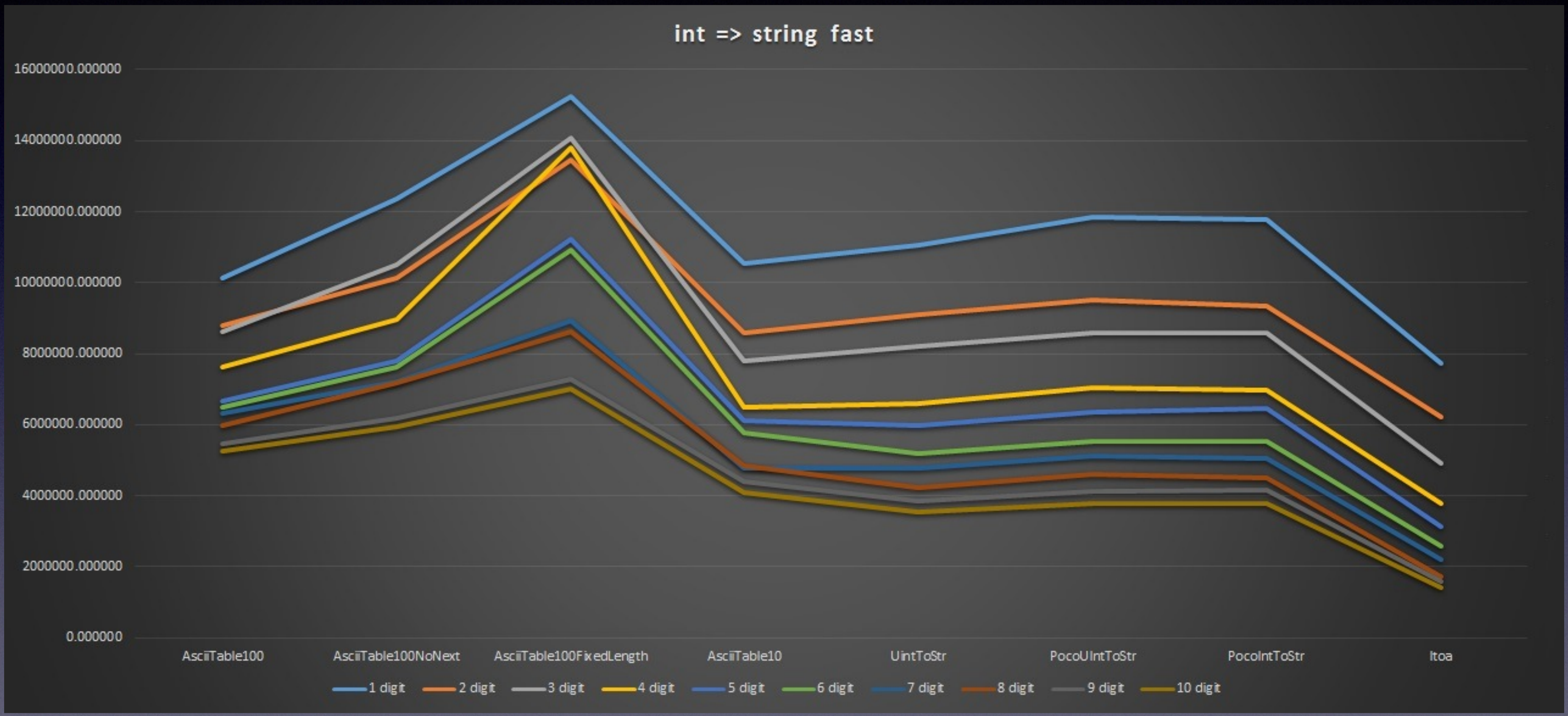
```
unsigned u64ToAsciiTable(uint64_t value, char* dst) {
  // ... char table buffer
  uint32_t const length = digits10(value);
  dst += length;

    while (value >= 100) {
        auto const i = (value % 100) * 2;
        value /= 100;
        *--dst = digits[i + 1];
        *--dst = digits[i];
    }

    if (value < 10) {
        dst[0] = '0' + uint32_t(value);
    }
    else {
        auto i = uint32_t(value) * 2;
        *--dst = digits[i + 1];
        *--dst = digits[i];
    }  return length;
}
```

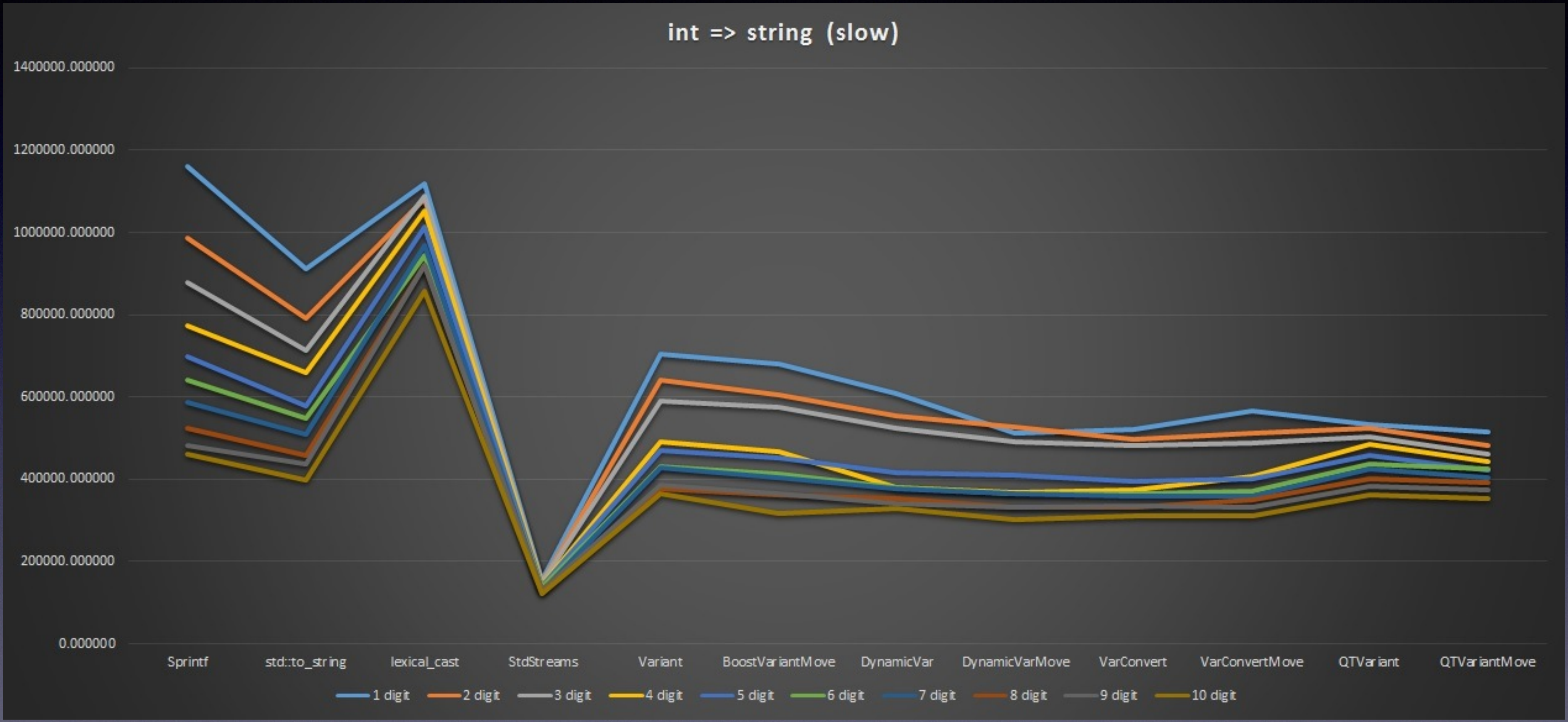
Use fixed length?

```
unsigned u64ToAsciiTable(uint64_t value, char* dst) {
    // ... char table buffer
    uint32_t const length = std::numeric_limits<uint64_t>::digits10;

    dst += length; length = 0;
    while (value >= 100) {
        auto const i = (value % 100) * 2;
        value /= 100;
        *--dst = digits[i + 1];
        *--dst = digits[i];
        length += 2;
    }

    if (value < 10)
    {
        dst[0] = '0' + uint32_t(value);
        ++length;
    }
    else {
        auto i = uint32_t(value) * 2;
        *--dst = digits[i + 1];
        *--dst = digits[i];
        length += 2;
    }
    return length;
}
```

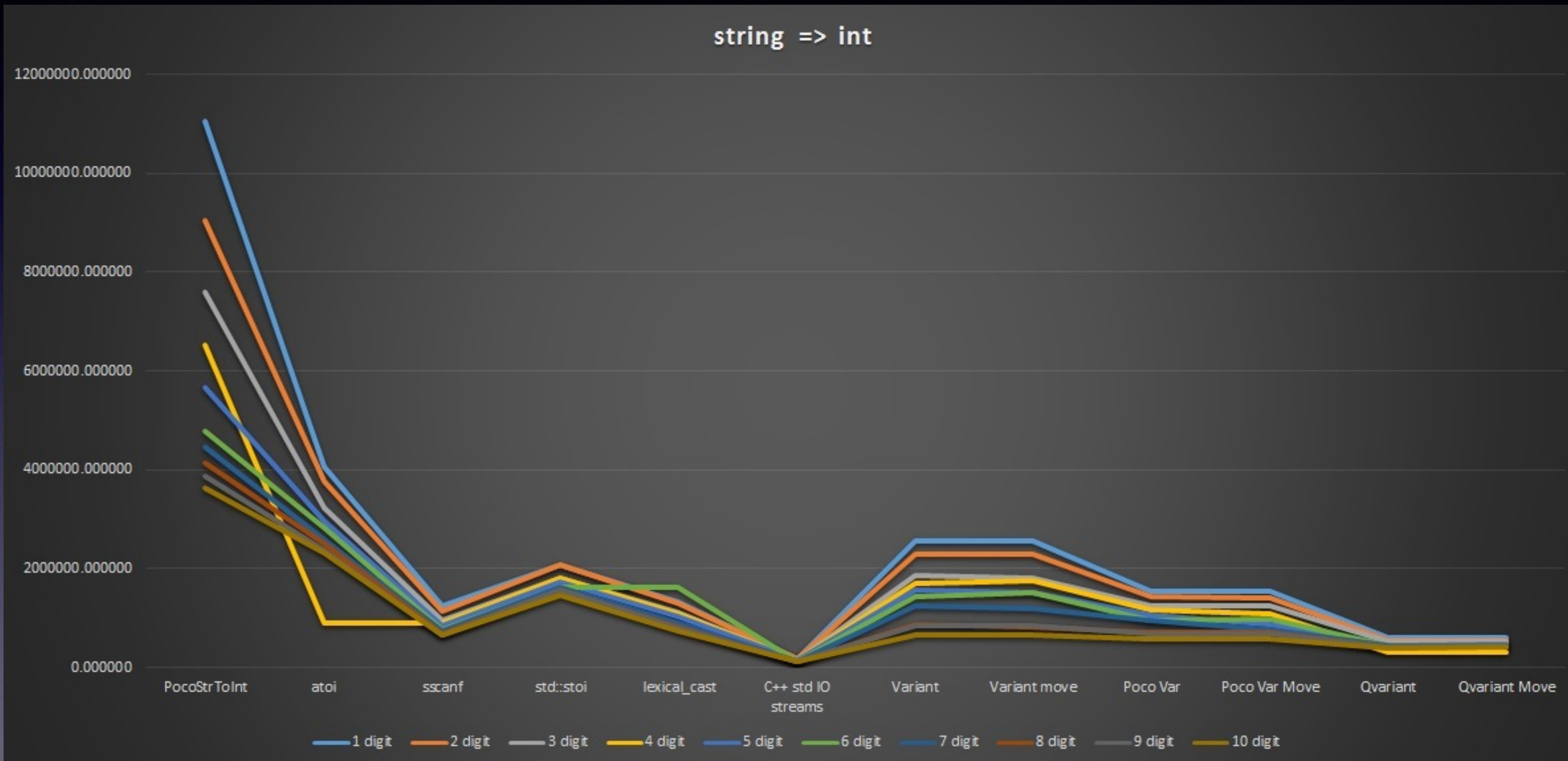


int => str all
16000000.000000
14000000.000000
12000000.000000
10000000.000000
8000000.000000
6000000.000000
4000000.000000
2000000.000000
0.000000
AsciiTable100
AsciiTable100NoNext
AsciiTable100FixedLength
AsciiTable10
UIntToStr
PocoUIntToStr
PocoIntToStr
Itoa
Sprintf
std::to_string
lexical_cast
StdStreams
Variant
BoostVariantMove
DynamicVar
DynamicVarMove
VarConvert
VarConvertMove
QTVariant
1 digit
2 digit
3 digit
4 digit
5 digit
6 digit
7 digit
8 digit
9 digit
10 digit


int => string fast
16000000.000000
14000000.000000
12000000.000000
10000000.000000
8000000.000000
6000000.000000
4000000.000000
2000000.000000
0.000000
AsciiTable100
AsciiTable100NoNext
AsciiTable100FixedLength
AsciiTable10
UintToStr
PocoUIntToStr
PocoIntToStr
Itoa
1 digit
2 digit
3 digit
4 digit
5 digit
6 digit
7 digit
8 digit
9 digit
10 digit



int => string (slow)
1400000.000000
1200000.000000
1000000.000000
800000.000000
600000.000000
400000.000000
200000.000000
0.000000
Sprintf
std::to_string
lexical_cast
StdStreams
Variant
BoostVariantMove
DynamicVar
DynamicVarMove
VarConvert
VarConvertMove
QTVariant
QTVariantMove
1 digit
2 digit
3 digit
4 digit
5 digit
6 digit
7 digit
8 digit
9 digit
10 digit



string => int
12000000.000000
10000000.000000
8000000.000000
6000000.000000
4000000.000000
2000000.000000
0.000000
PocoStrToInt
atoi
sscanf
std::stoi
lexical_cast
C++ std IO streams
Variant
Variant move
Poco Var
Poco Var Move
Qvariant
Qvariant Move
1 digit
2 digit
3 digit
4 digit
5 digit
6 digit
7 digit
8 digit
9 digit
10 digit

“We all float down here”

```
const double val = 0.1;
char result[64] = {0};
std::cout << “\t\t\t .0123456789ABCDEF" << std::endl;

doubleConversion(result, 64, val, -17, 17, 17);
std::cout << "doubleConversion:\t" << result << std::endl;

sprintf(result, "%1.17f", val);
std::cout << "sprintf:\t\t" << result << std::endl;

std::cout << "std iostream:\t\t”
   << std::setprecision(17) << val << std::endl;
```

```
                             .0123456789ABCDEF
double-conversion:          0.1
sprintf:                    0.10000000000000001
std iostream:               0.10000000000000001
Press any key to continue . . .
```

Floating point conundrum

`0.1 == 0.09999999999999999777955395074968691915273666381835937 5`

What `0.1` equals to depends on the number of significant digits:

- 15 digits, `0.1 == 0.0999999999999999`(7779…)
- 16 digits; `0.1 == 0.1000000000000000`(0779…)
- 17 digits; `0.1 == 0.10000000000000001`

A Brief History of Floating Point Conversion

Coonen 1980:

It is possible to place specific bounds on how much error could be allowed in input and output while maintaining idempotence.

White and Steele 1971 - 1990 (Dragon):

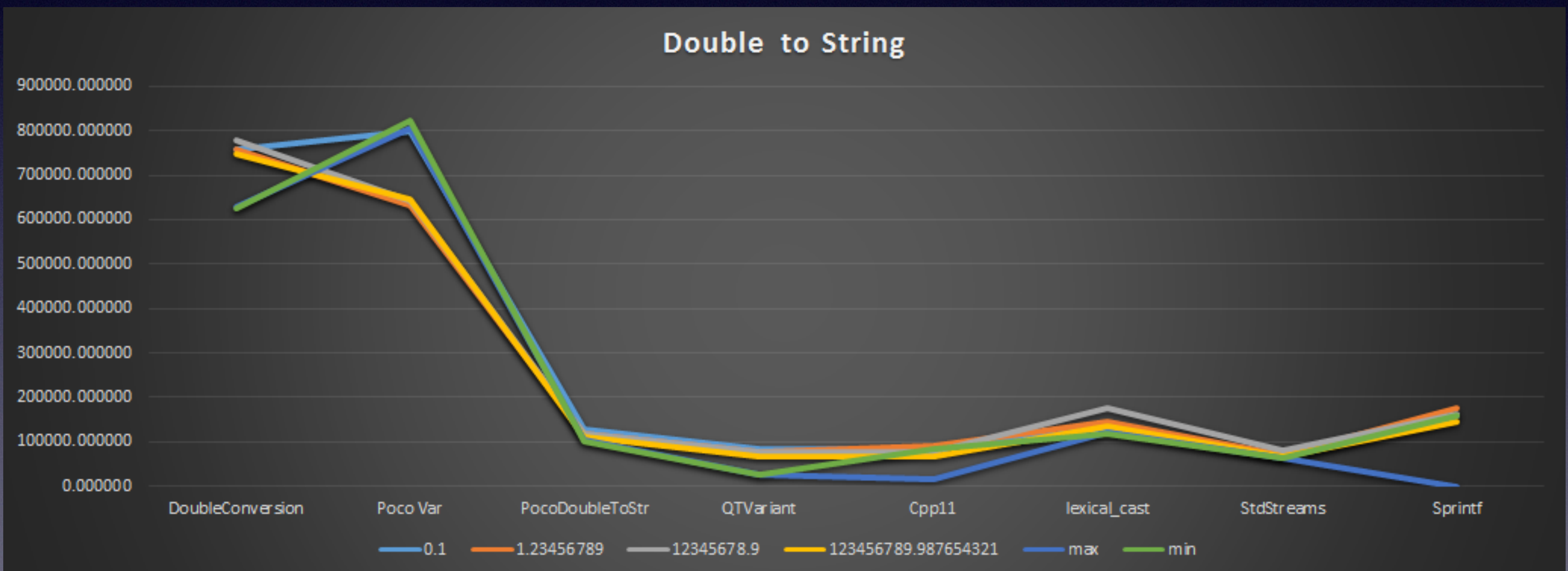
Convert floating-point numbers to decimal with the fewest digits needed to preserve idempotence.

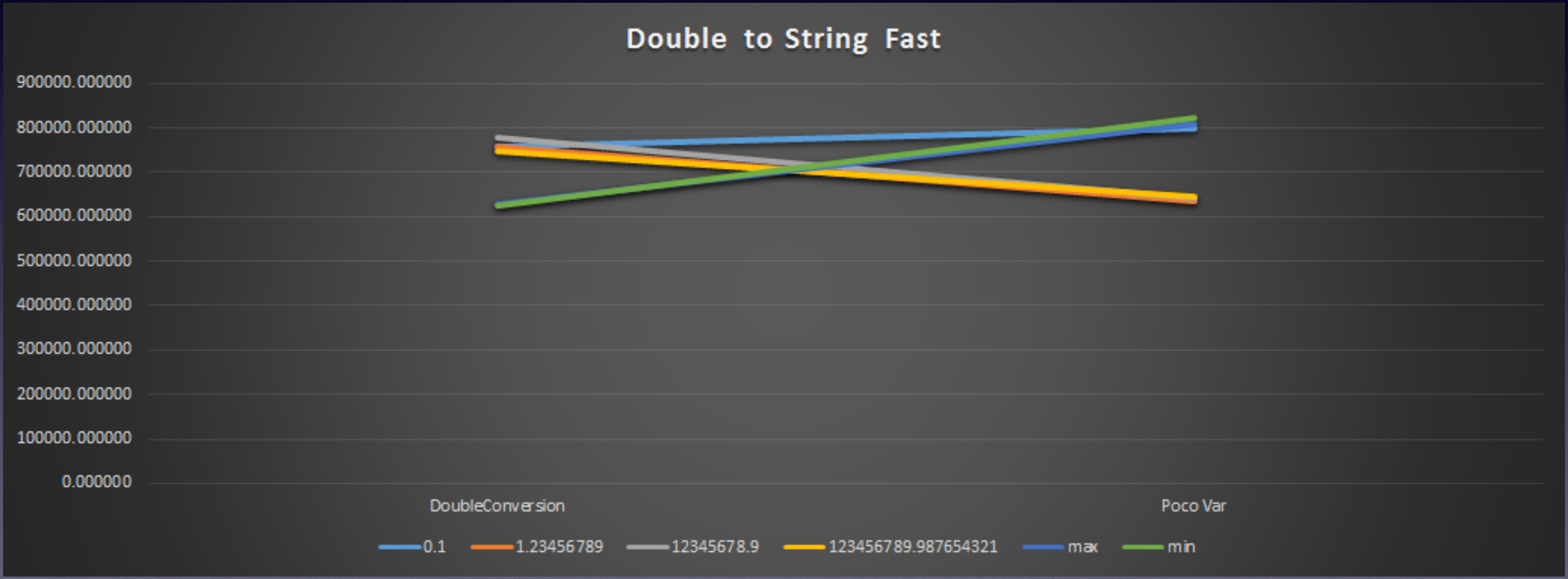
Loitsch 2007 (Grisu):

Printing Floating-Point Numbers Quickly and Accurately with Integers.

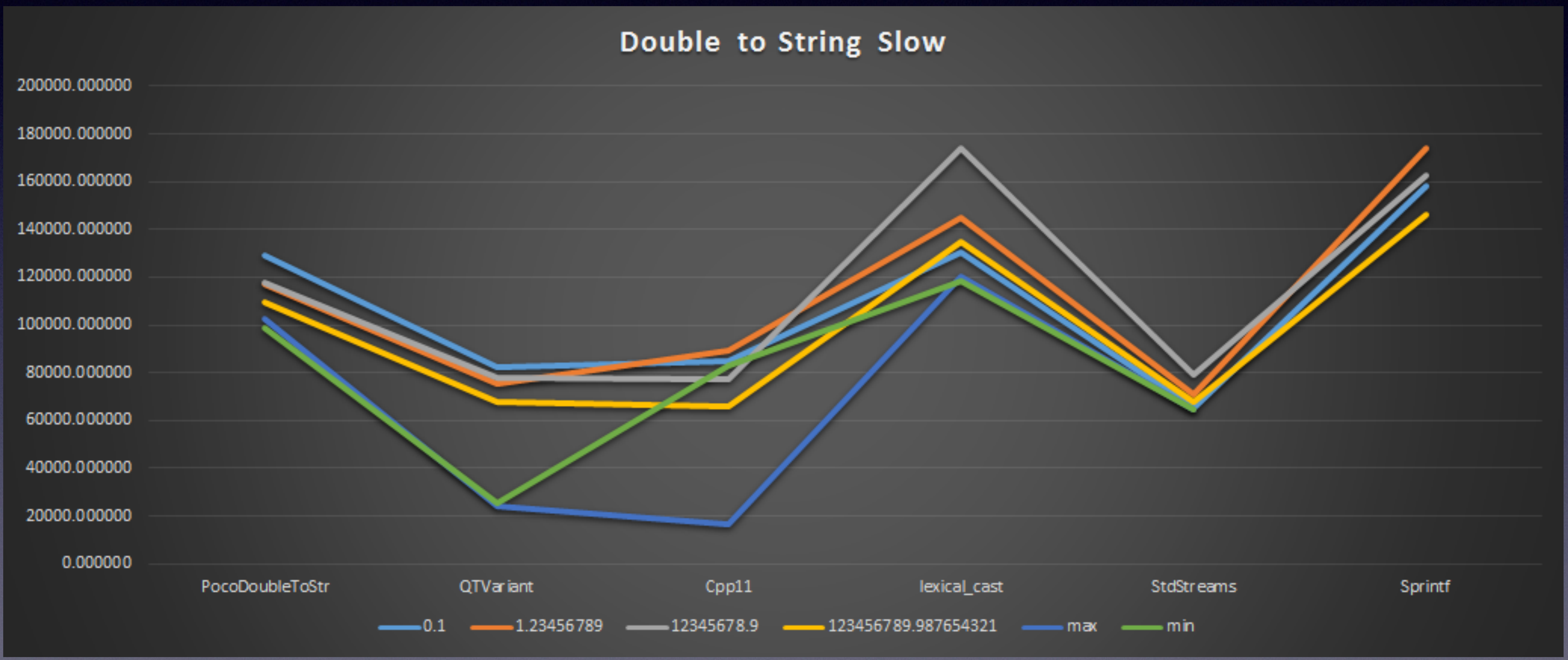
“Even simple floating-point output is complicated.”

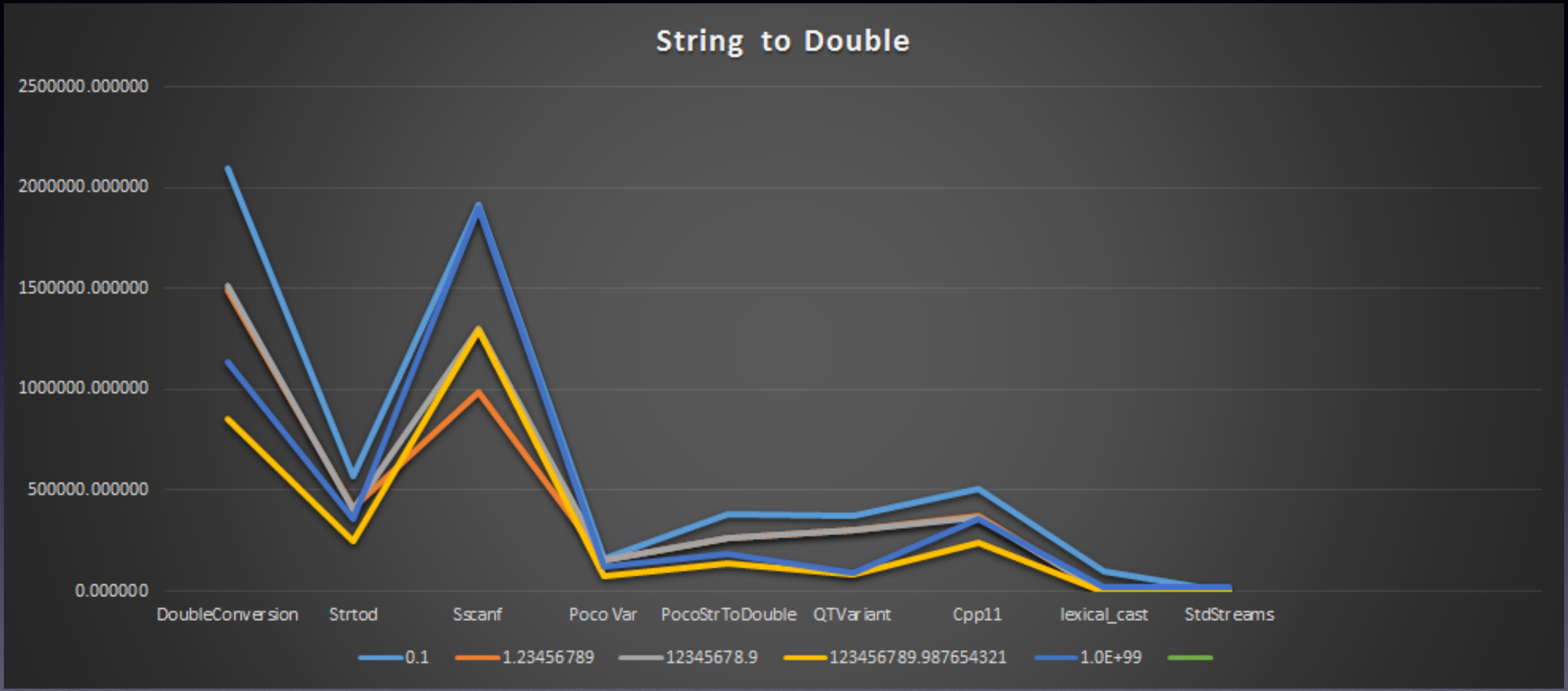
-Andrew Koenig, 2014

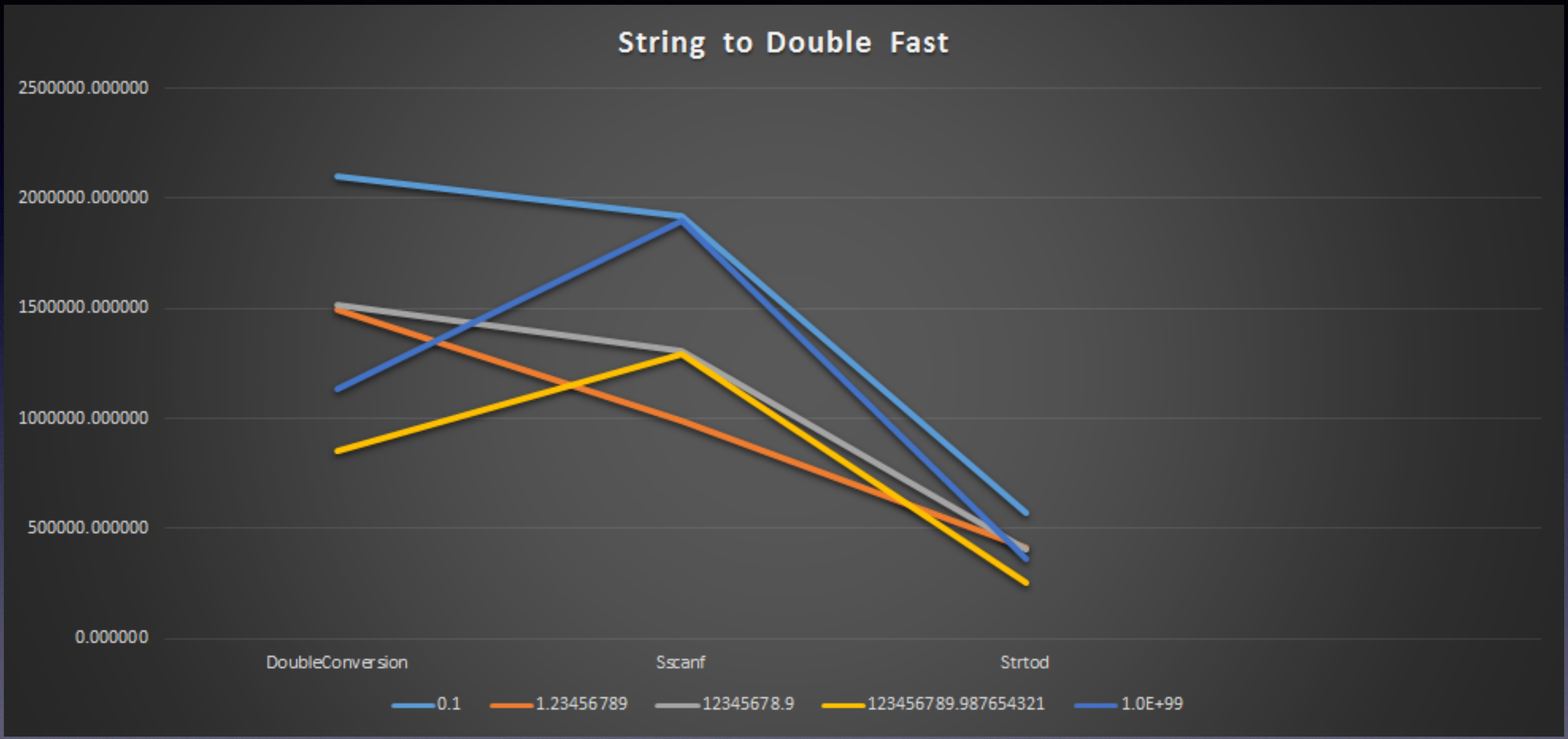

Double to String
900000.000000
800000.000000
700000.000000
600000.000000
500000.000000
400000.000000
300000.000000
200000.000000
100000.000000
0.000000
DoubleConversion
Poco Var
PocoDoubleToStr
QTVariant
Cpp11
lexical_cast
StdStreams
Sprintf
0.1
1.23456789
12345678.9
123456789.987654321
max
min

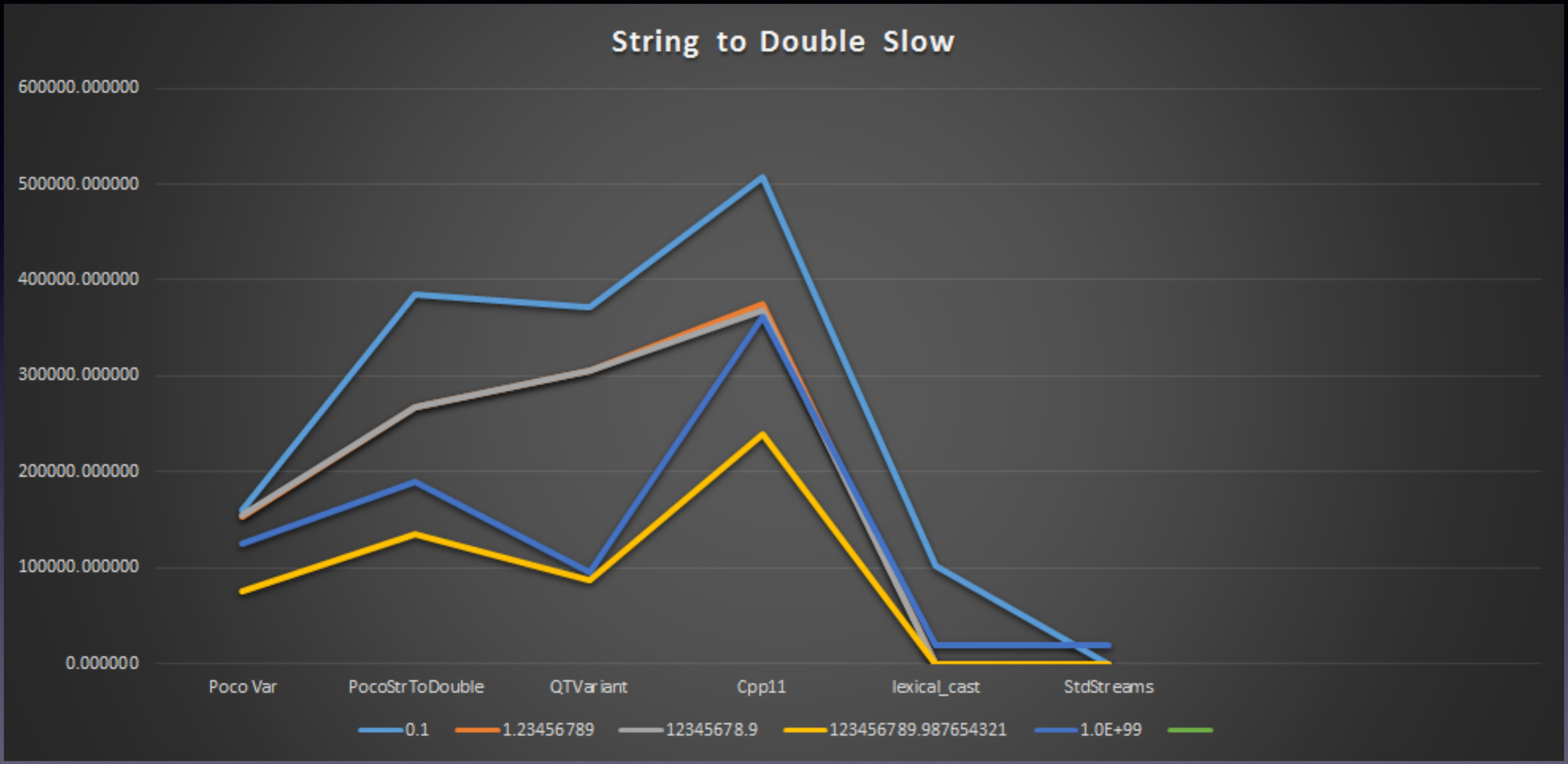


Double to String Fast
900000.000000
800000.000000
700000.000000
600000.000000
500000.000000
400000.000000
300000.000000
200000.000000
100000.000000
0.000000
DoubleConversion
Poco Var
0.1
1.23456789
12345678.9
123456789.987654321
max
min


Double to String Slow
200000.000000
180000.000000
160000.000000
140000.000000
120000.000000
100000.000000
80000.000000
60000.000000
40000.000000
20000.000000
0.000000
PocoDoubleToStr
QTVariant
Cpp11
lexical_cast
StdStreams
Sprintf
0.1
1.23456789
12345678.9
123456789.987654321
max
min


String to Double
2500000.000000
2000000.000000
1500000.000000
1000000.000000
500000.000000
0.000000
DoubleConversion
Strtod
Sscanf
Poco Var
PocoStrToDouble
QTVariant
Cpp11
lexical_cast
StdStreams
0.1
1.23456789
12345678.9
123456789.987654321
1.0E+99


String to Double Fast
2500000.000000
2000000.000000
1500000.000000
1000000.000000
500000.000000
0.000000
DoubleConversion
Sscanf
Strtod
0.1
1.23456789
12345678.9
123456789.987654321
1.0E+99


String to Double Slow
600000.000000
500000.000000
400000.000000
300000.000000
200000.000000
100000.000000
0.000000
Poco Var
PocoStrToDouble
QTVariant
Cpp11
lexical_cast
StdStreams
0.1
1.23456789
12345678.9
123456789.987654321
1.0E+99

Overview

	printf/scanf	itoa / atoi	C++11	C++ IOStream
int => str	Slow	Med	Slow	Slow
str => int	Slow	Med	Med	Slow
float => str	Med	N/A	Slow	Slow
str => float	Fast	N/A	Med	Slow

Celero

- Goal: eliminate all of the noise and overhead, and measure just the code under test
- Establish baseline and measure relative performance
- Multi-platform (Windows, Linux, Mac)
- Easy to build and use
- https://github.com/DigitalInBlue/Celero

Stabilizer

- a compiler and runtime system that enables statistically rigorous performance evaluation
- eliminates measurement bias by comprehensively and repeatedly randomizing the placement of functions, stack frames, and heap objects in memory
- random placement makes anomalous layouts unlikely and independent of the environment
- re-randomization ensures they are short-lived when they do occur
- http://plasma.cs.umass.edu/emery/stabilizer

Conclusion

- memory allocation is performance killer
- conversion speed depends heavily on what is converted (e.g. [u]int or double)
- floating point problem is hard but Grisu (double-conversion) was a significant step forward
- there’s lots of dynamic conversion code out there but no “silver bullet”
- landscape shaping forces:
  - performance
  - accuracy (floating point)
  - convenience