(Why We Need) Large Code Base Change Ripple Management in C++

Niall Douglas

Contents:

1. What I am pitching: a new Boost library, and a possible motivating vision of a long-term future for C++ and Boost
2. Many (contentious) claims as to why … 🙂

What I am pitching: a low-level embedded graph database for Boost

The proposed embedded graph database

- All first tier content is a standard file which can be opened, mmapped etc
- Takes advantage of filing system specific features such as extents, metadata/data journaling, hole punching, copy on write, bitrot self healing etc
- Strong versioning and MVCC concurrency
- Per-graph content protection (e.g. parity healing)
- Content addressable with a per-graph hash of your choice

The proposed embedded graph database

- Per-graph optional ACID transactions
- Per-graph arbitrary indexers (e.g. Boost.Graph, SQLite3, ZIP etc)
- Network shardable to other copies with interrupted partial copy resumption
- Objects can be executable (in fact is self hosting)
- Uses an algorithm very close to git
- Designed to act like increasing mount points of ‘database-ness’ overlaid onto the filesystem

The proposed embedded graph database

- Performance is expected to be within two and five orders of magnitude slower than the big iron graphstores
- Write transaction performance shouldn’t be lower than ten per second hopefully
- All designed to work during very early process bootstrap i.e. before shared libraries are loaded
- There is nothing close in existing software - this design and its abilities are very unique

It is really more of a “generic data persistence library”

Want more detail?

There is a 25,000 word accompanying position white paper on ArXiv at http://arxiv.org/abs/1405.3323:

- What makes code changes ripple differently in C++ to other languages?
- Why hasn’t C++ replaced C in most newly written code?
- What does this mean for C++ a decade from now?
- What C++ 17 is doing about complexity management
- What C++ 17 is leaving well alone until later: Type Export
- Detail about the embedded graph database design
- Two example killer applications for such a graph database namely:
    a. An example C++ object components design (similar to Bandela’s)
    b. An example extensible Filesystem design

What does any of this have to do with the price of fish?

- What does this have to do with change ripple management?
- Or Boost?
- Or C++?
- Or anything?

Let the contention begin!

I am now going to articulate a (motivating) vision for a long term future goal of C++ and Boost which explains why we might absolutely need one of these databases soon

I will then make a series of supporting claims most of which will be contentious (and hence I place them last!)

What is this (motivating) vision of the future of C++ and Boost?

A post-C++ 17 goal:

I'd like to see a world where we can write C++ as if ***everything*** in the solution (including Python, Lua, PHP etc, including C++ in closely related processes) is header only, no matter the size of the program

(It is a natural result of a complete ABI management solution)

Example:

```
class Foo { virtual void boo(int a); };
```

A rule in the graph database says that when this type is changed, it should be reflected via `std::reflect` into a Python binding:

```
class Foo:
    @accepts(int)
    @returns(None)
    def boo(self, a)
```

Example:

Let’s break Foo’s ABI:

```
class Foo { virtual void boo (double); };
```

When you hit compile in C++ for each use of `Foo::boo()` in Python code you get:

```
TypeWarning:  'boo' method accepts (float), but was given
(int)
```

To clarify:

- C++ moves from a source file compilation model to a type graph compilation model (similar to exported templates)
- The type graphs are compiled a bit like GLSL shaders into many tiny C++ Modules i.e. bits of precompile all put into the graphstore
- Reflection (runtime) equals a graph query
- To bootstrap a C++ process equals visiting a graph query for all the matching tiny C++ Modules

Consequences:

- Instant notification of breakages from a code change no matter how far away
- Optimally minimal rebuild (and can dispense with external build tools)
- Optimal optimisation which can be pushed onto a batch pass/cloud compute

Consequences:

- Easy components via ripple propagation rules in the graphstore
- No longer pollutes all over the C symbol table
- Finally a real ABI management solution
- Other programming languages would be ***very*** interested in this

Why???

Claims (to which I shall return):

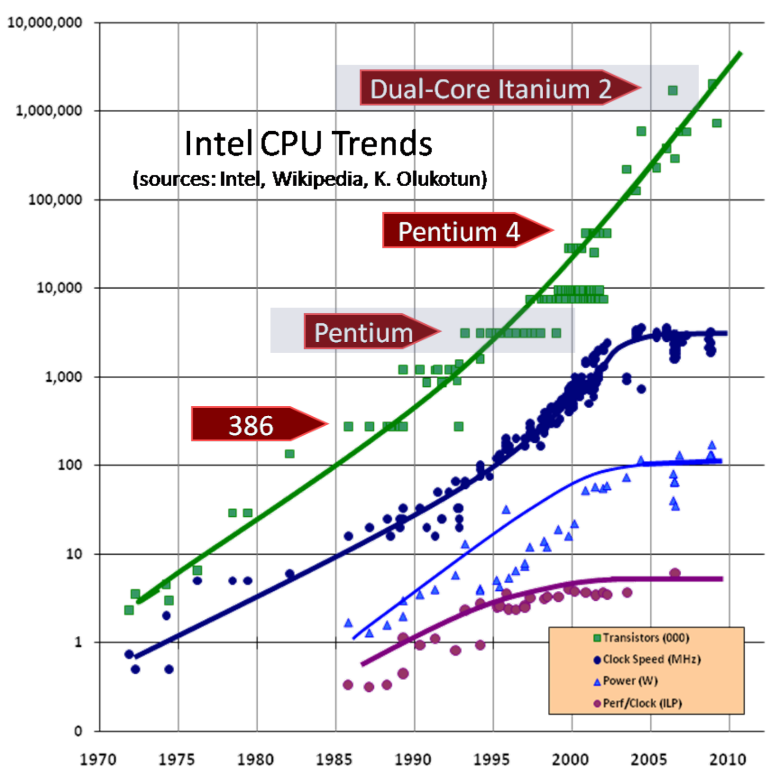
1. C++ is in relative decline
2. Boost is in both absolute and relative decline

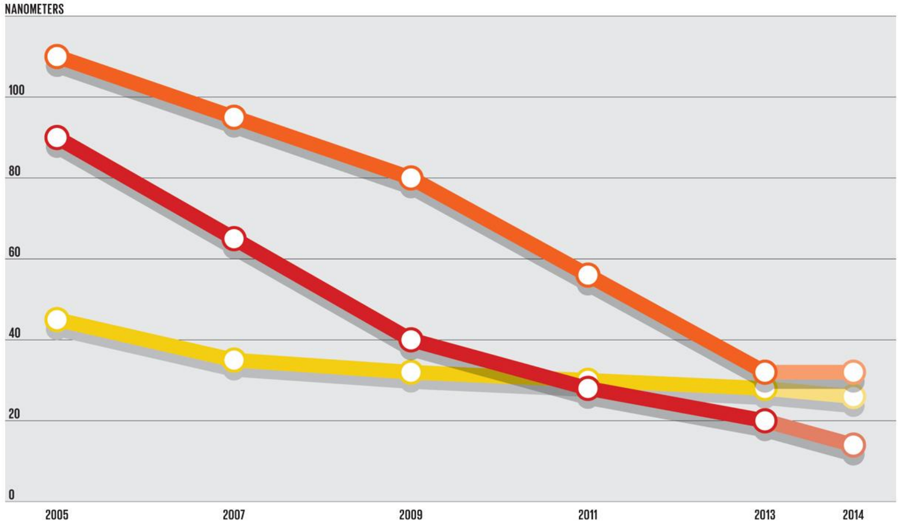
Therefore:

3. We need to return to becoming a better systems programming language
4. We need “signature projects” for C++ 11/14

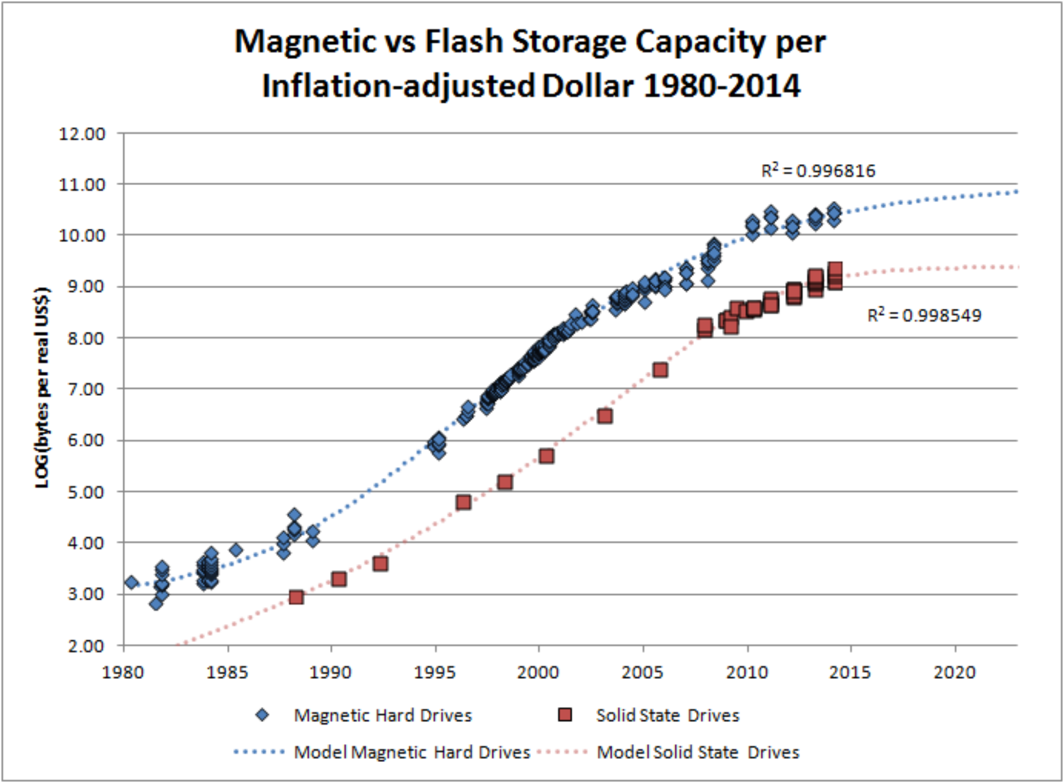
Why this instead of including more code per-compiland?

Where hardware is going soon





Where hardware is going soon



My best speculations on effects:

I therefore claim these likely outcomes in the future:

1. The cost of including ever more source code per-compiland stops being sunk by transistor density growth
2. Therefore build times start to rise and stop falling with time
3. Therefore C++ starts to look more like in the 1990s with small compiles and large links
   - ○ Except with all our fancy modern C++ techniques

My best speculations on effects:

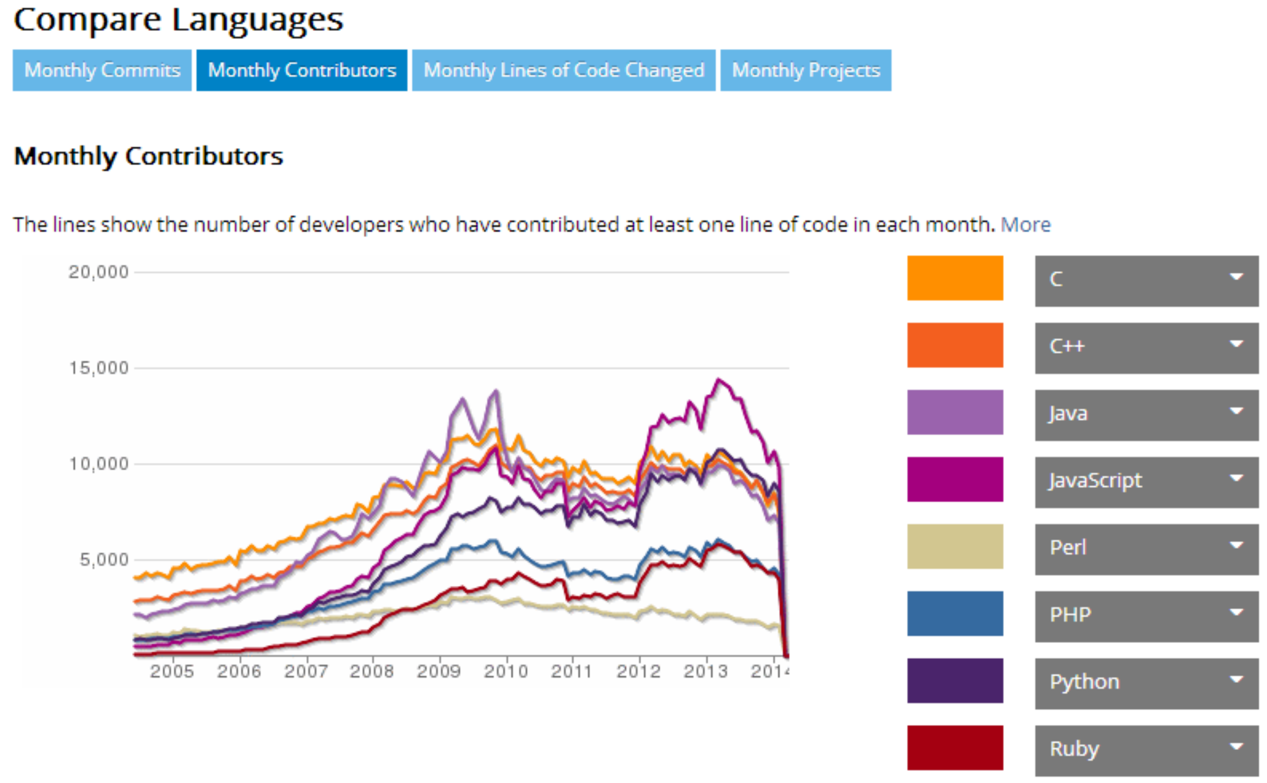
Also:

4. There will be a return to growth for systems programming languages as software is refactored to cope with linear growth CPU and RAM but still exponential growth storage
5. I claim this will happen around 2017-2020 if present trends continue and no surprises turn up

   (Mass production of Graphene or Phosphorine transistors won’t be ready by 2020 at present rates of R&D)

Will C++ be that majority choice of systems language?

The present structural revolution



Compare Languages

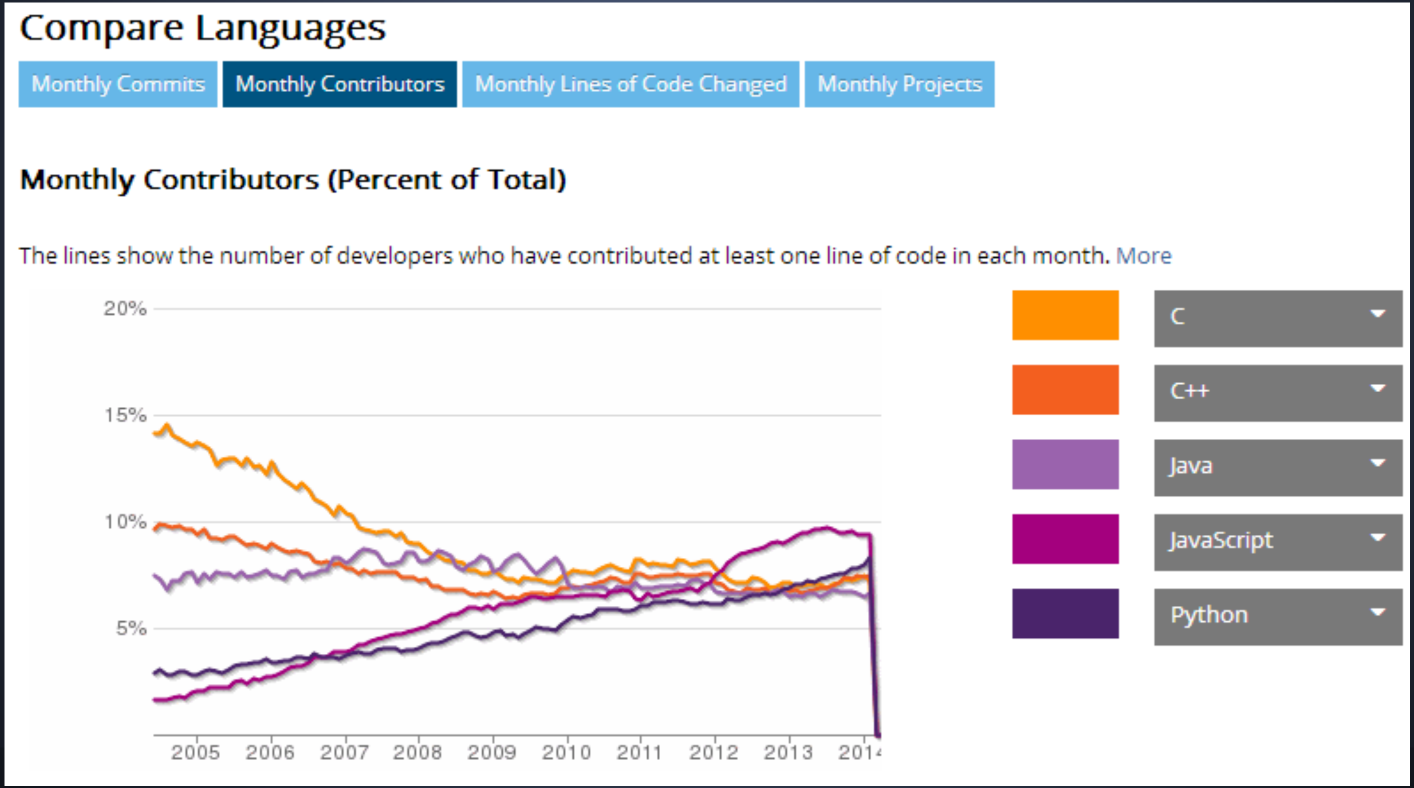
Monthly Contributors

The lines show the number of developers who have contributed at least one line of code in each month. More

Claim:
C++ has been in relative decline for a decade

(with a pause 2009-2011)

The present structural revolution

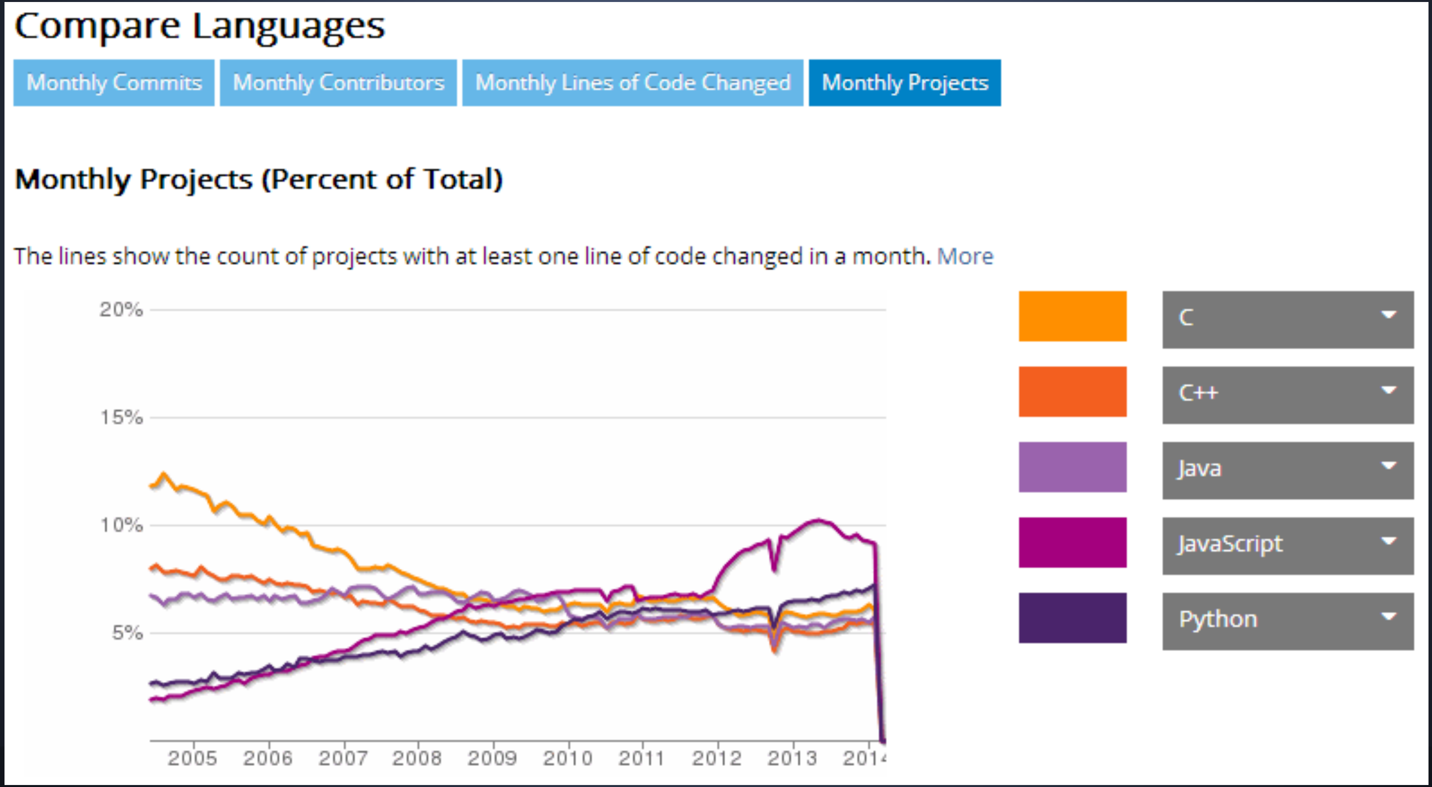


Compare Languages

Monthly Contributors (Percent of Total)

The lines show the number of developers who have contributed at least one line of code in each month. More

The present structural revolution



Compare Languages

Monthly Projects (Percent of Total)

The lines show the count of projects with at least one line of code changed in a month. More

Possible reasons why C++ is in relative decline

And why should we care?

Why C++ is in relative decline?

1. C++ is no longer a general purpose programming language - it’s a *niche specialist* language suited for:
   a. Low latency (async etc)
   b. Maximum performance (maths etc)
   c. Gluing application and service code written in other languages like Python or C# together

Note that C is good at all of the above too, and ***still*** remains more popular in open source than C++ for ***new*** code

Why C++ is in relative decline?

2. C++ has stopped trying to be the best systems programming language possible
    - ○ C++ 11/14 adds a ton of great stuff BUT …
        - ■ Did any of it persuade someone like Linus that C++ might be tolerable in the Linux kernel?
        - ■ Do the Python/Ruby/Lua/PHP interpreter guys look at C++ 11/14 and go “wow that transforms our use case for C++ over C”?

Why should we care?

If:

a. C++ remains best in class for high performance math

b. C++ remains very strong in low latency async

c. BUT C remains preferred to C++ as a systems programming language

Why should we care?

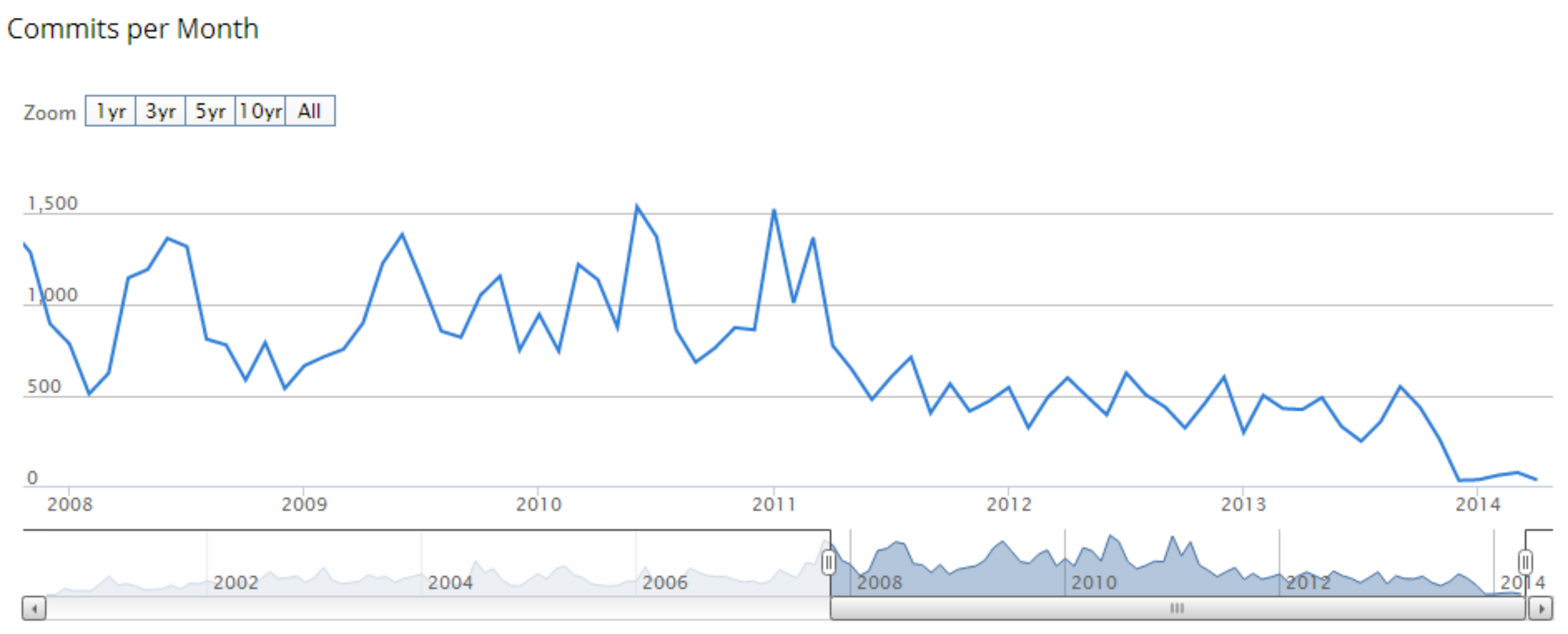
Then, assuming the previous claims are true, a reasonable prediction of the future is:

1. C (or some extension thereof) gets the majority of post-exponential hardware systems language growth
2. C++ becomes ever more like Haskell, with only a rarified programmer elite able to touch it

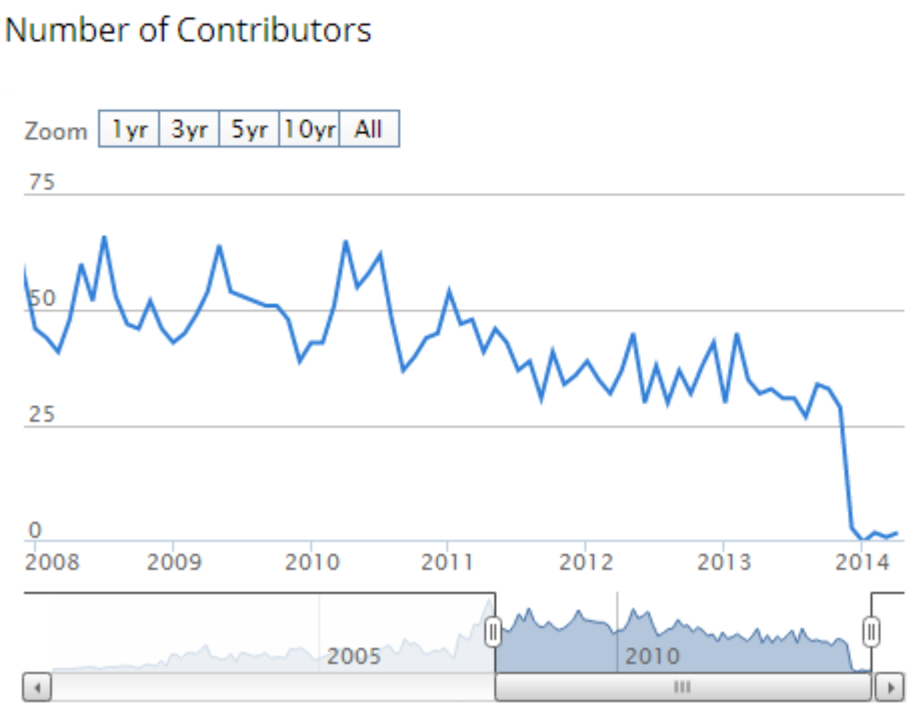
Is that what we want?

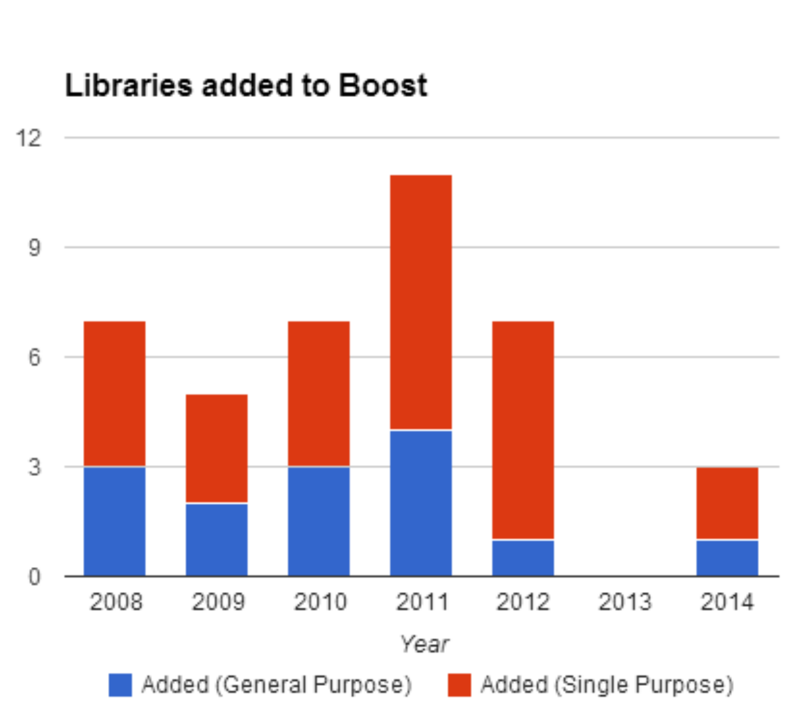
Claim:
Since 2011 Boost is in both an absolute & relative decline

The Decline of Boost

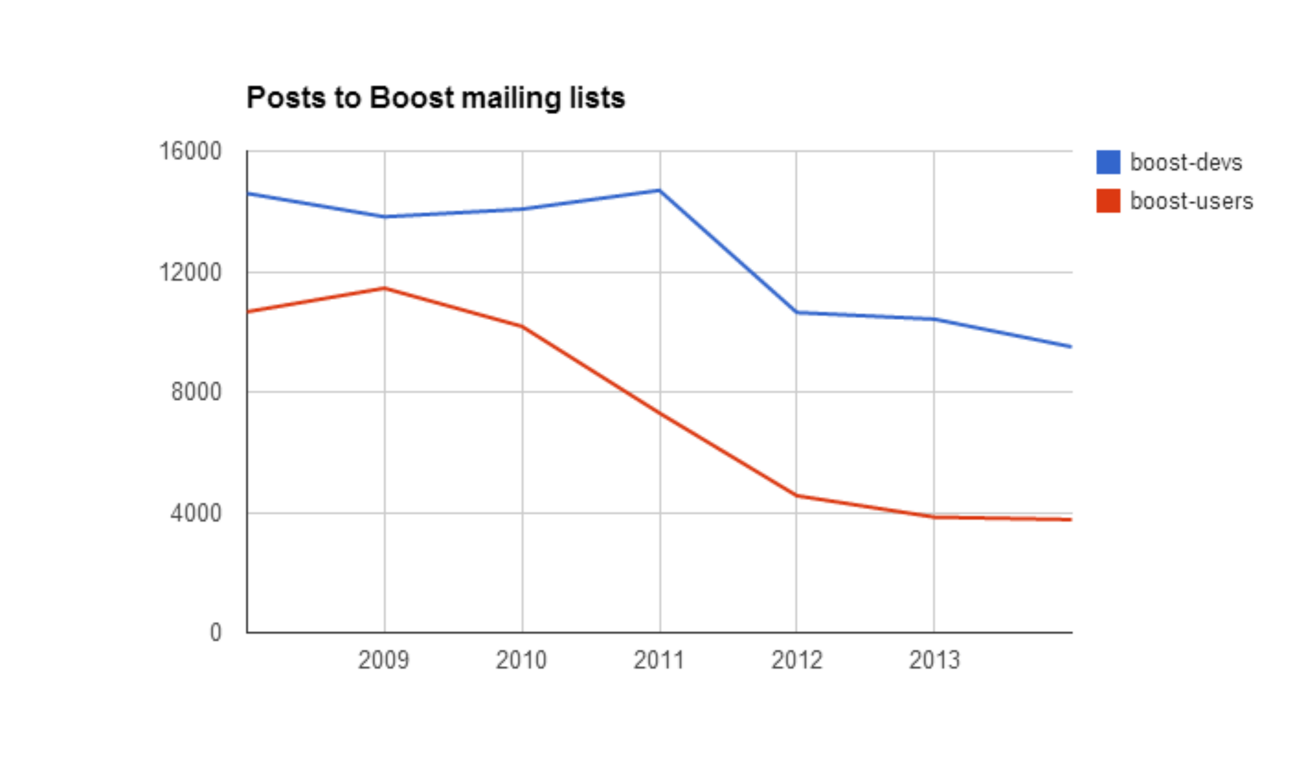


The Decline of Boost





The Decline of Boost



Possible reasons why Boost is in decline

And why should we care?

Why Boost is in decline?

1. The C++ 11/14 standard library now can do what people used to need Boost for

    - Unfixed bugs in Boost means that people simply switch to C++ 11/14 instead if they can
    - People think a particular Boost library needs all of Boost as a dependency - a real showstopper
    - Boost is seen as simply no longer relevant

Why Boost is in decline?

To quote a highly respected engineer from this very conference who said a few days ago:

“Boost used to be about all the stuff you really wanted in the standard. Now Boost looks like all the stuff that wasn’t good enough to get into the standard”

- somebody well known (not me)

Why Boost is in decline?

2. Boost has become two mutually incompatible sets of libraries:
   a. The C++ 11 STL emulation library for C++ 98
   b. A set of libraries which push the boundaries of C++, as Boost once used to in the 1990s
      - With the latter being suffocated of late

Why Boost is in decline?

3. Most of the interesting C++ 11 libraries on the internet appear to have no interest in joining Boost (with a few honourable exceptions)

I personally find that very scary

Why Boost is in decline?

4. Boost makes no attempt to preserve ABI stability, and therefore is not welcome in large stable code bases

- None of the improvements in C++ 11/14 do anything for change ripple management, so even mild ABI breakage is intolerable and therefore Boost is banned/pinned to some ancient version

Why should we care?

Simple answer:

How many of the changes to C++ 11/14 standard over C++ 98/03 originated in Boost?

Assertion:
C++ ought to return to trying to become a better systems programming language

Assertion:
We want Boost to continue to lead out the future of C++

(and therefore all systems programming)

So what?

What if one or all of the earlier assertions is false?

What if none of the issues described is a real problem?

What if all this is merely hand wavy nonsense?

An embedded graph database is still extremely useful:

- In 2014 it is still too hard for more than one process to write to many files concurrently
- In 2014 it is still too easy to lose data

Want more detail?

There is a 25,000 word accompanying position white paper on ArXiv at http://arxiv.org/abs/1405.3323:

- What makes code changes ripple differently in C++ to other languages?
- Why hasn’t C++ replaced C in most newly written code?
- What does this mean for C++ a decade from now?
- What C++ 17 is doing about complexity management
- What C++ 17 is leaving well alone until later: Type Export
- Detail about the embedded graph database design
- Two example killer applications for such a graph database namely:
    a. An example C++ object components design (similar to Bandela’s)
    b. An example extensible Filesystem design