Building the software

Overview

- This lecture will focus on a number of practical issues that come up when one is faced with when building a large set of software
- In particular I'd like to focus on some transitions that (for many of us!) have been happening recently, or are ongoing....
- Some of you work in the context of a large HEP collaborations and thus you may find the basic transition has been already been done by your software releases/tools group
  - But there are probably still things to investigate!
- Or perhaps you are at the other (standalone) end of the spectrum: “Here I am with my source tarball, now what?”

Overview

- In particular I intend to provide a bit of background information and describe some performance-related issues for the following topics:
  - Compilers
    - in particular the transition from gcc3 to gcc4
  - Shared libraries
  - Floating point math
  - The transition to 64bit

Personal Biases (Fair Warning)

- There is some bias towards Linux, gcc, about x86/x86_64, etc.
- Obviously there is also some bias towards experimental High Energy Physics (HEP) issues.
- And even within that I have some bias towards talking about LHC experiments and in particular my own experiment (CMS)...

Compilers

- The compiler is clearly one of the most important tools for achieving optimum code performance
- Unless we want to hand-code everything in assembly, we rely on it to take our code, written in a high-level language like C++, and produce the fastest code possible.
- Usually we also want it to accomplish that in the shortest time possible, to use as little memory as possible doing it, to produce the smallest code possible, etc.

GNU compiler collection (gcc)

- The workhorse open source compiler, used by most of us, most of the time, these days...
- Front ends for C, C++, Fortran (Ada, Objective-C(++), Java and others)
- Back ends for x86, x86_64 (Alpha, ARM, ia-64, PowerPC, Sparc and many others)
- Most software today is easily configured to build with gcc
- Although most of work on linux/x86(_64) today, or at most MacOSX/x86_64, at least in non-DAQ environments, the wide availability of gcc for different OS/CPU combinations once eased porting C/C++ from one to another.

LLVM/Clang Compiler

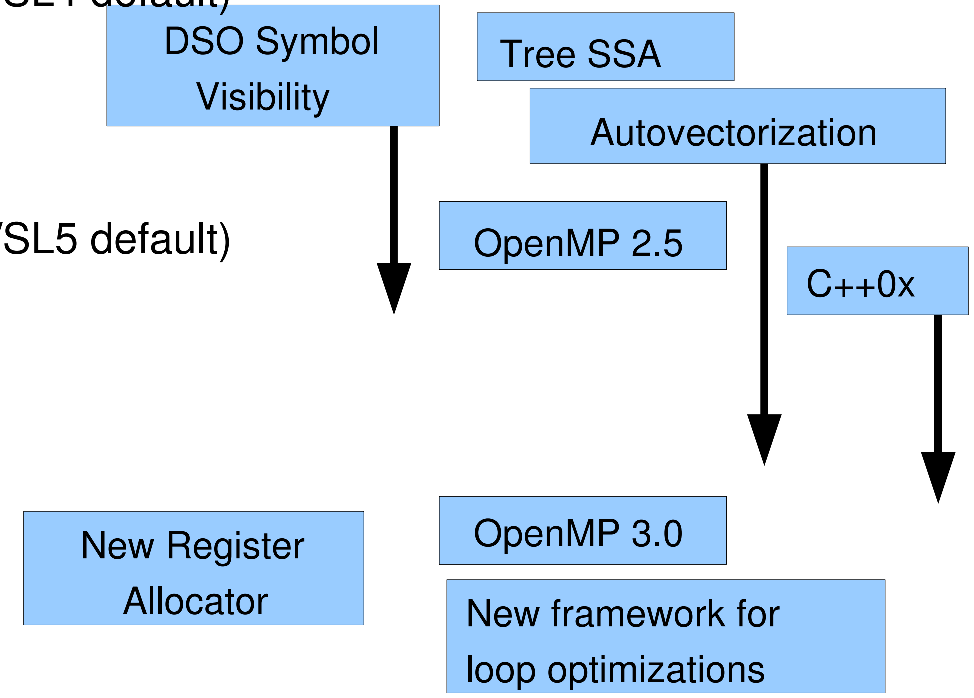
- Recent open source compiler project, aiming to build a set of modular compiler components
- The initial versions replace the optimizer and code generation of gcc, but still reuse the gcc front-end/parser (compatible compiler options!)
- A separate project (Clang) aims to replace gcc front-end for C/C++/Objective-C
- Targets both static compilation as well as just-in-time (JIT) compilation
- Sponsorship (in particular) by Apple

Intel Compiler (icc)

- Intel's showcase Fortran/C/C++ compiler(s)
- Arguably focused on demonstrating the best possible performance to be obtained from their processors
- Independent compiler (language syntax, code quality)
- Generates code for all of the Intel processors, plus in principle other x86/x86_64 compatible, i.e. AMD, processors
- Available for Linux/MacOSX/Windows, proprietary license
- The default behaviour for floating point may or may not be what is desired (see later slides about floating point)

GCC version timeline/features

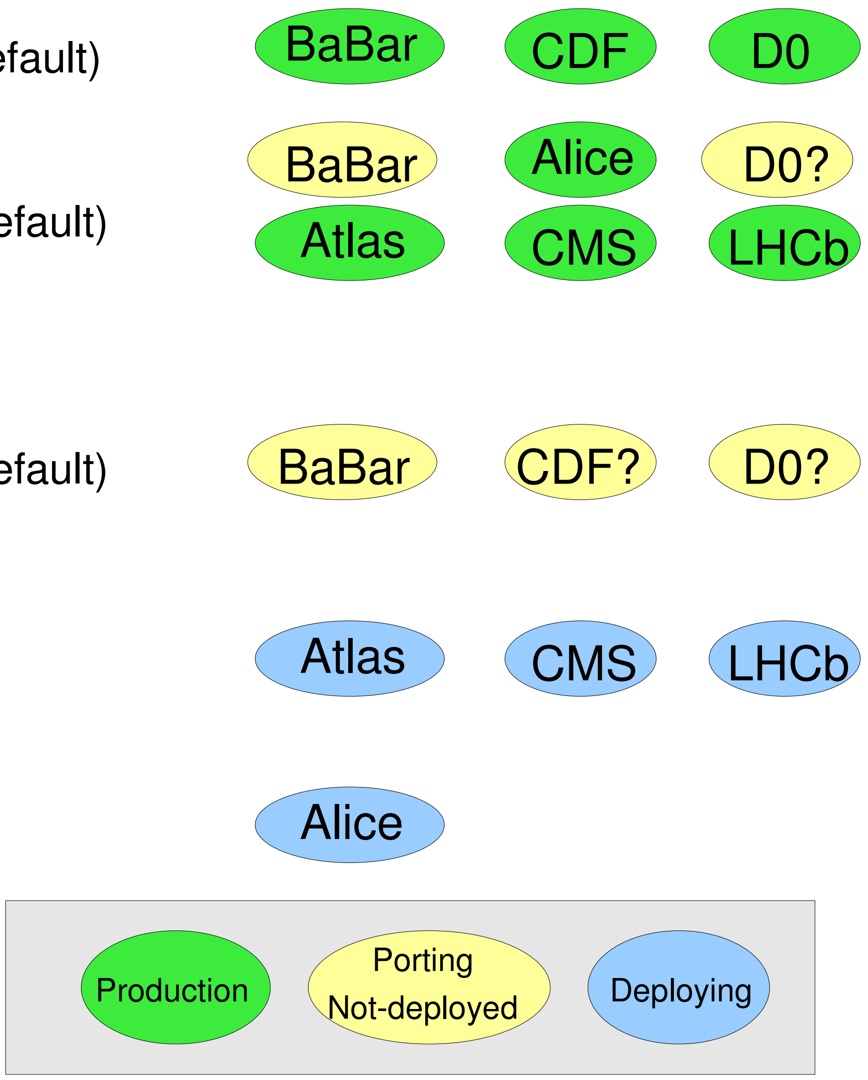
- GCC 3.2.0 - 14 Aug, 2002
  - GCC 3.2.3 - 22 Apr, 2003 (~RHEL3/SL3 default)
- GCC 3.4.0 - 18 Apr, 2004
  - GCC 3.4.6 - 06 Mar, 2006 (~RHEL4/SL4 default)
- GCC 4.0.0 - 20 Apr, 2005
- GCC 4.1.0 - 28 Feb, 2006
  - GCC 4.1.2 - 13 Feb, 2007 (~RHEL5/SL5 default)
- GCC 4.3.0 - 05 Mar, 2008
  - GCC 4.3.2 - 27 Aug, 2008
  - GCC 4.3.4 - 04 Aug, 2009
- GCC 4.4.0 - 21 Apr, 2009
  - GCC 4.4.1 - 22 Jul, 2009



Various banner improvements in recent gcc4.x compiler versions.
(See Release notes for full list, though!)

GCC versions and HEP

- GCC 3.2.0 - 14 Aug, 2002
  - GCC 3.2.3 - 22 Apr, 2003 (~RHEL3/SL3 default)
- GCC 3.4.0 - 18 Apr, 2004
  - GCC 3.4.6 - 06 Mar, 2006 (~RHEL4/SL4 default)
- GCC 4.0.0 - 20 Apr, 2005
- GCC 4.1.0 - 28 Feb, 2006
  - GCC 4.1.2 - 13 Feb, 2007 (~RHEL5/SL5 default)
- GCC 4.3.0 - 05 Mar, 2008
  - GCC 4.3.2 - 27 Aug, 2008
  - GCC 4.3.4 - 04 Aug, 2009
- GCC 4.4.0 - 21 Apr, 2009
  - GCC 4.4.1 - 22 Jul, 2009



Rough status of some large HEP experiments
(my personal understanding!)

gcc4x – Getting from here to there

- Porting your code forward to gcc4x (from gcc3x) is fairly straightforward:
  - There are often minor issues with missing system includes as other includes were cleaned up
  - Additional small cleanups related to namespace consistency, templates, duplicate parameters, etc.
  - Lots of documentation out there about the minor migration issues including dedicated “porting guides” for gcc4.3/gcc4.4
  - A slightly larger issue for some people is the transition from g77 to gfortran
- Regarding code from other people on which you depend:
  - You may find that those packages have been ported to gcc4x, but only in versions more recent than those you have
  - See Pere's talk on “Software Physical Design”: minimizing dependencies helps!

gcc4x – Getting from here to there

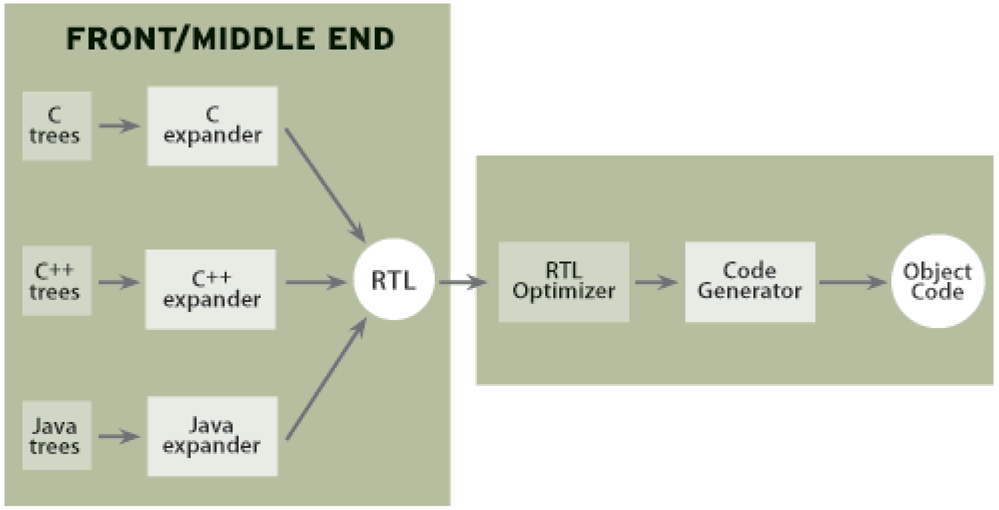
- A few technical issues to watch out for:
  - If you build the compiler yourself on some platform, as opposed to taking it from an operating system (OS) “default compiler” installation, in rare circumstances you can have problems with other tools like bintuils
  - If you are not rebuilding cleanly everything yourself, e.g. mixing gcc3x and gcc4x builds in your applications, note that you can wind up with multiple copies/versions of things like libstdc++.so
  - Some OS vendors (e.g. RH) ship more recent “preview” versions of their compilers, in addition to the default compiler. The preview version may have (as in the RH5 gcc4.3 example) the libstdc++.so downgraded for consistency to that of the default compiler.
  - In general the OS vendors don't ship the canonical, downloadable versions of many things, e.g. gcc, binutils, etc. but apply some number of patches on top.

gcc

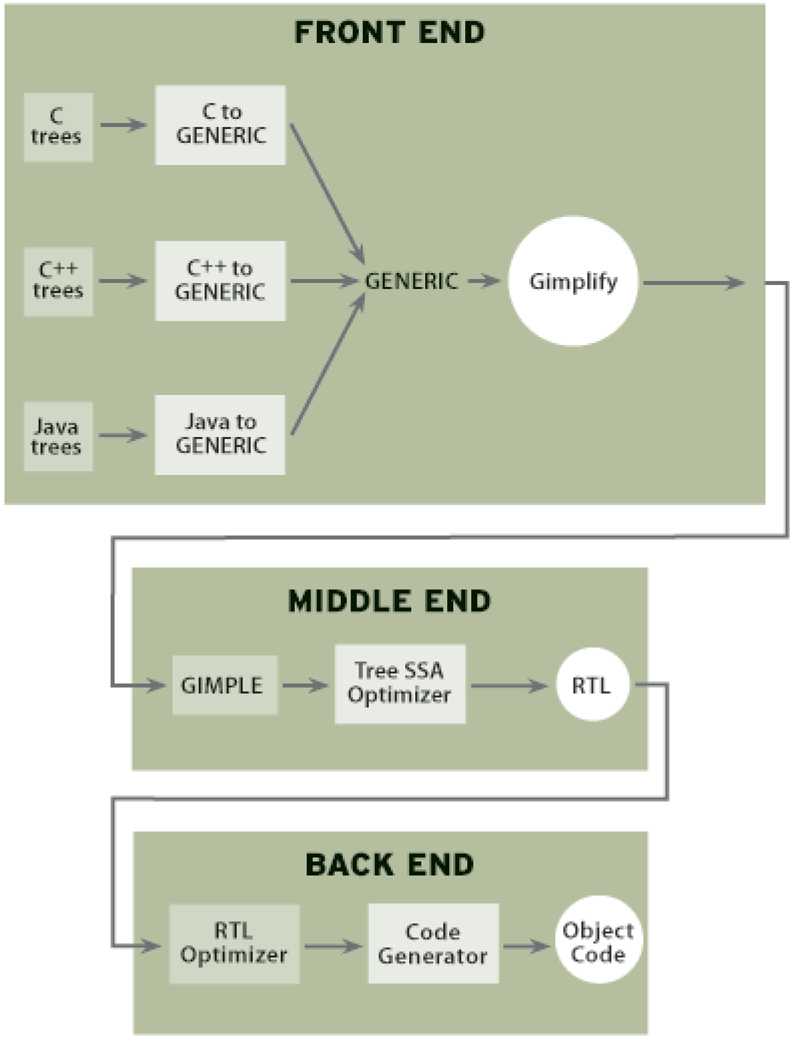
- This is an exciting time for many of us as the transition to gcc4x means that lots of new tools and features become available.
- I will present some background information on a few selected topics
- Others will cover (or have covered) Autovectorization, OpenMP, C++0X in much greater detail separate presentations

Compiler structure changes – gcc

gcc 3.x



gcc 4.x



Figures from:
http://www.redhat.com/magazine/002dec04/features/gcc/
"From Source to Binary: The Inner Workings of GCC"
D.Novillo, used with permission

Compiler optimizations

- A simple comparison of the evolution of the optimization options by version is here:

  http://cern.ch/elmer/gcc_opt_by_version.txt

- For 99% of purposes, you will probably stick with -O0, -O1, -O2, -Os, -O3, but it is interesting to see how things are evolving. In general you would probably only fine-tune the optimization options based on specific profiling results.

Compiler limitations

- The compiler should limit itself to optimizations that do not change the behaviour of the program
  - See however later notes on floating point
- It often also doesn't have enough information to make certain decisions, e.g. due to the scope of what it is looking at or due to lack of knowledge about inputs
  - While you (mostly) don't see this for the optimization process, you've certainly noticed it if you've spent time fixing compiler warnings

Aliasing/restrict

- A classic example where a compiler must make conservative choices is regarding possible aliasing of variables

```
void addme(int n, int* s, int* a, int* b) {
  for (i=0; i<n; i++) {
    s[i] = a[i] + b[i];
  }
}
```

- The compiler cannot *assume* that s*, a* and b* all point to independent storage.
- __restrict__ keyword can be used

Compiler limitations

- In general the compiler may not be able perform many optimizations in the presence of code constructs that may have side effects:
  - Objects within loops
  - Function calls within loops
- It can't make assumptions about code it cannot see

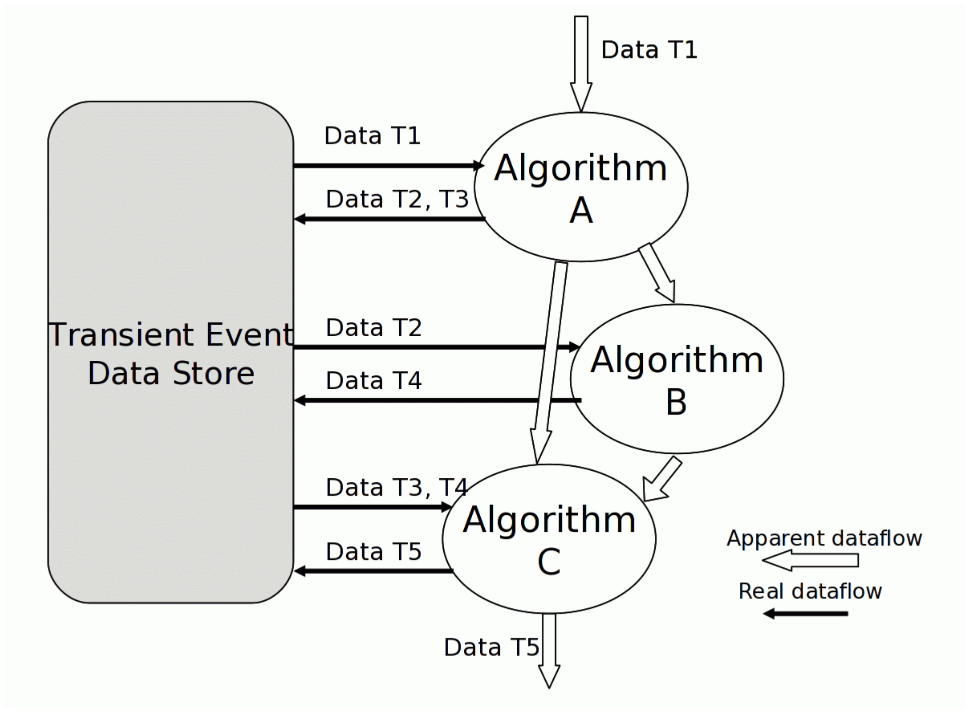
Code (physical) packaging

- The last generation(s) of experiments often used archive (.a) libraries for code and linked some set of static binaries for various purposes
  - The question is “which static binaries?”
- The current generation of experiments has moved almost entirely to using dynamic shared (.so) libraries as a technology
- In the following slides, I will discuss two use cases where they are advantageous, but also some of the resulting problems (plus a few solutions)

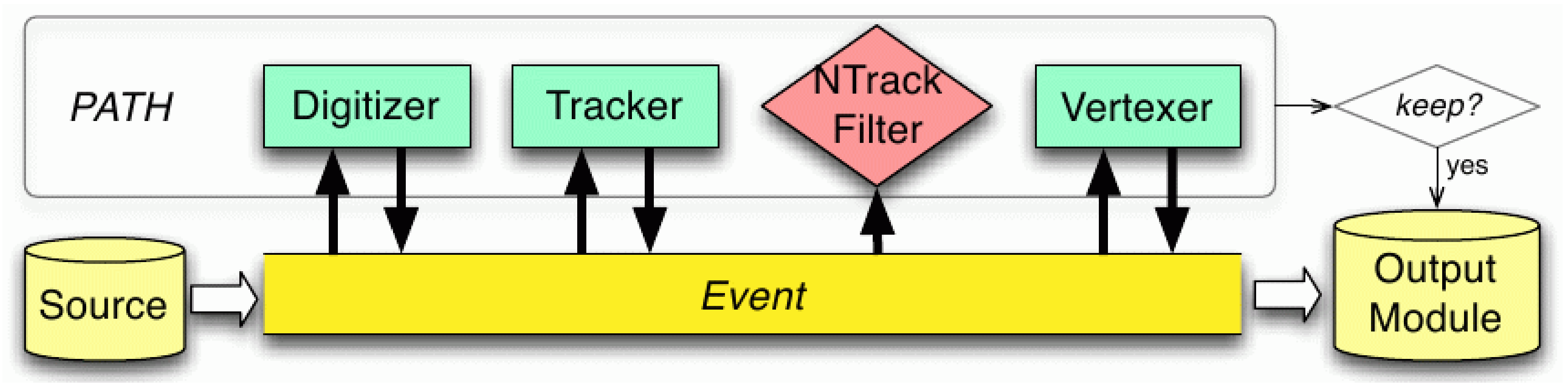
HEP Event Processing Frameworks

- Modern HEP event processing frameworks are usually designed such that:
  - A number of independent “modules” or “algorithms” are run one after another
  - The “modules”/”algorithms” use as input data taken from the “event”
  - The “modules”/”algorithms” may produce and add data to the “event” or they may do other things (e.g. stop some set of modules from

HEP Event Processing Frameworks

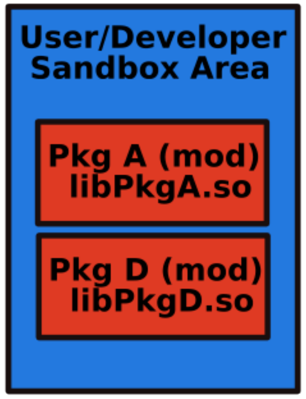


Highly configurable frameworks allow for a specific choice of modules (sequences, etc.) to be run via simple edits to a python config script, without needing to relink a full binary. A standard stub application reads the config script and loads the necessary shared libraries as dynamic “plugins”

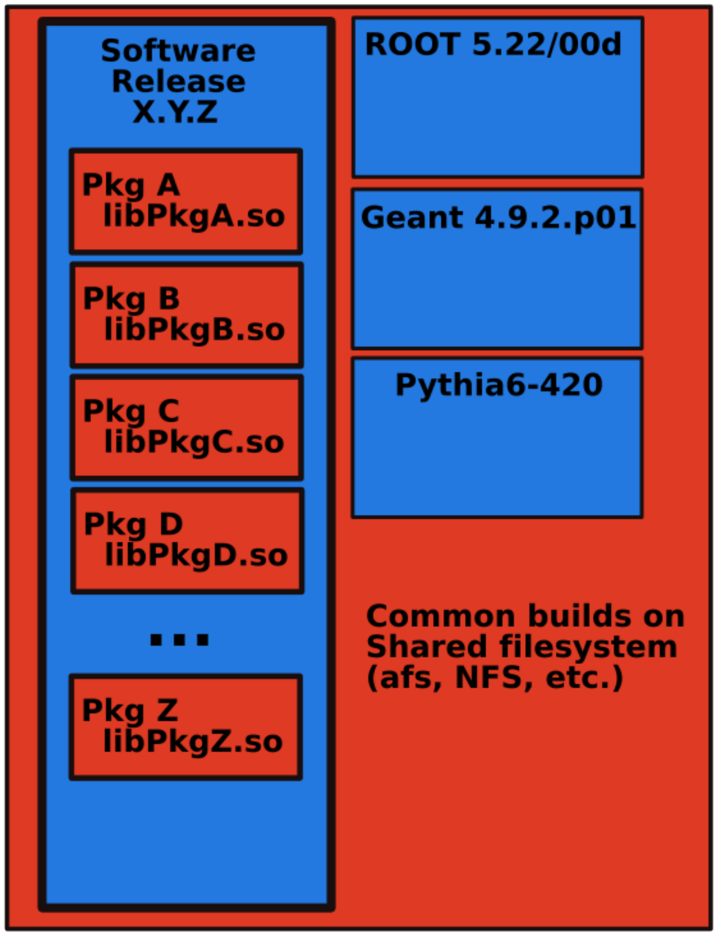


Software Development Model

Most modern experiments support in some way a "base release"/user sandbox style of development, e.g. via SCRAM or CMT or SoftRelTools



The terminology and technical details vary from system to system, but the basic idea is that a user/developer wants to modify and rebuild only a subset of the relevant code and libraries.



Shared library/plugin advantages

- Much more convenient for rapid turnaround testing:
  - Many application changes can be made by a simple python edit alone.
  - Even if code has been changed, it is often only necessary to rebuild one or more shared libraries (fast) and not relink one or more static binaries
- There are also advantages for release management and bookkeeping
  - No need to keep both (archive) libraries, (numerous) largish binaries and application config files around. These are replaced by just shared libraries and the config files.

Shared library/plugin advantages

- The fact that multiple users might be reusing the same shared libraries in different applications on any given machine permits the OS to share a single in-memory copy between them. If each user has a custom static binary for each possible small variation on the application configuration, no sharing is possible.
- The possibility of replacing a .so used by other people with another (compatible) one, e.g. with some bug fix, is also possible, but less commonly done.

Shared libraries – downsides

- Size/bloat
  - What is loaded in memory is often far more than what is actually needed. In practice, there are often also fine-grained redundancies between .so's
- Run time performance - Poor code locality
- Startup time
  - The dynamic linker is still needed, but at run-time
- Virtual Memory Fragmentation (See Lassi's talk)
- All of these are worsened by having too many .so's and each may or may not be a relevant effect for you

Shared libraries – size/bloat

- Three things in particular contribute to the larger than necessary in-memory size of the code:
  - Redundant copies of the same template function, instantiated over and over again in each .so
  - Redundant (out-of-line) copies of inline functions (i.e. those functions you have as implementations in the .h)
  - Code which is co-resident in the same .so, but not needed in this particular application
- In practice you care mostly if the relevant and the irrelevant are mingled together on memory pages

Shared libraries – size/bloat

- What can be done to reduce the code size/bloat?
  - Combine .so's which are always loaded together
  - Look for and remove various types of bloat, e.g.:
    - Non-templated parts of implementations in templates or obviously non-inlinable functions can be moved out-of-line.
    - Use of exceptions (if truly exceptional) can often be moved out-of-line and/or out of template functions.
  - In short, look to see what is redundant/repeated and what contributes to the size! “Why do I need N copies of that?”
  - gcc4.x on average appears to produce smaller code, -Os can be used in some cases, see also next slides on symbols

Shared libraries– Startup time

- Even though a full link of a static binary is avoided through the use of shared libraries and/or plugins, nothing comes for free.
- Some of that work still needs to be done by the dynamic linker at run-time, according to the actual load addresses of the .so's.
- For full description of this process and options for improving performance, see U.Drepper, “How To Write Shared Libraries”
- In the following slides I will just give an overview!

Shared libraries– Startup time

- Each shared library exports a (sometimes quite large) dynamic symbol table
- The symbol lookup for (undefined) named symbols involves (hashed) string comparisons with things found in that symbol table
- The strings used in the comparisons are the mangled C++ names (which have the disadvantage of being both long and often having longish common initial character sequences)
- Many of the exported symbols don't actually need to be... The default is to export all globally visible symbols.

Symbol visibility

- The ELF format used for shared libraries permits one to control the visibility of the symbols in the shared library
- By default all globally visible symbols are exported
- There are several methods to control the symbol visibility:
  - gcc compiler switches
    - -fvisibility-inlines-hidden
    - -fvisibility=default, -fvisibility=hidden
  - Fine-grained use of __attribute__((visibility ("hidden"))) in source code
  - Linker export maps (wildcarded lists of symbols and the desired visibility for those symbols)

Notes on symbol visibility

- Most noticeable effects:
  - Faster startup, mostly due to fewer string comparisons
  - Smaller (loaded) library size, from reduced dynamic symbol table
- Difficulties:
  - -fvisibility=hidden is in practice difficult to use, as the various type_info in particular often needs to remain visible
  - Consistent choices across all relevant shared libraries needed
- Additional improvements were also made in the hash table used for the dynsym table (via –hash-style=gnu, available in the binutils shipped with RHEL5/SL5). And removing spuriously linked libraries (explicitly or with –as-needed) of course always helps.

Integer calculations

Fairly straightforward, all integers in some given range are representable

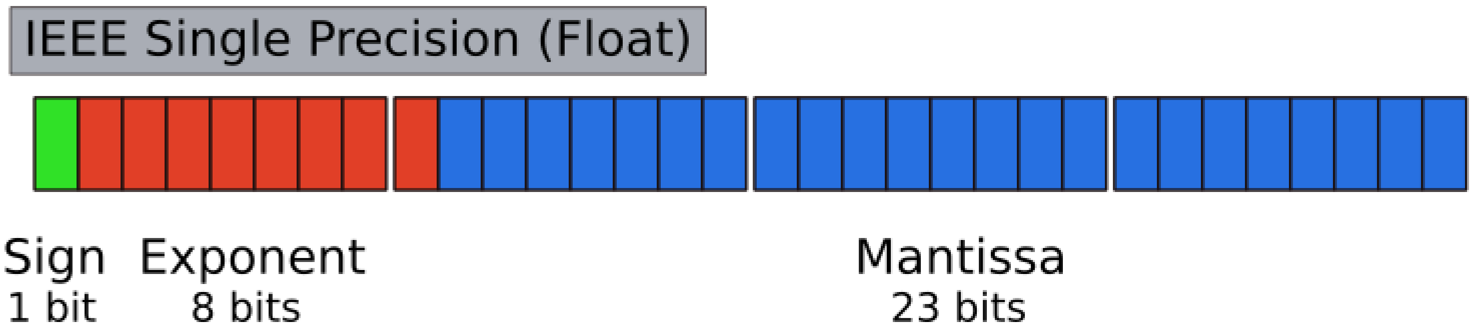
Math with integers is easy to understand and easily reproducible. The main "gotcha" is overflows.

- (signed) int
  - 11111111 11111111 11111111 11111110 == -2
  - 11111111 11111111 11111111 11111111 == -1
  - 00000000 00000000 00000000 00000000 == 0
  - 00000000 00000000 00000000 00000001 == 1
  - 00000000 00000000 00000000 00000010 == 2
  - 01111111 11111111 11111111 11111111 == 2147483647
  - 10000000 00000000 00000000 00000000 == -2147483648
- unsigned int
  - 00000000 00000000 00000000 00000000 == 0
  - 00000000 00000000 00000000 00000001 == 1
  - 00000000 00000000 00000000 00000010 == 2
  - 11111111 11111111 11111111 11111111 == 4294967295
  - 00000000 00000000 00000000 00000000 == 0

Floating point calculations

- Floating point calculations are a more complicated problem
- The problem stems from the fact that the number of real numbers on any given interval is not finite and countable, as for the integers.
- Using a finite number of bits, we thus cannot hope to represent all real numbers exactly for computations, not even by restricting the range
  - 32 bits => 2^32 = 4294967296 numbers max
- Indeed for floating point numbers we need to make restrictions on both the range and the precision of the representation

IEEE-754 representation



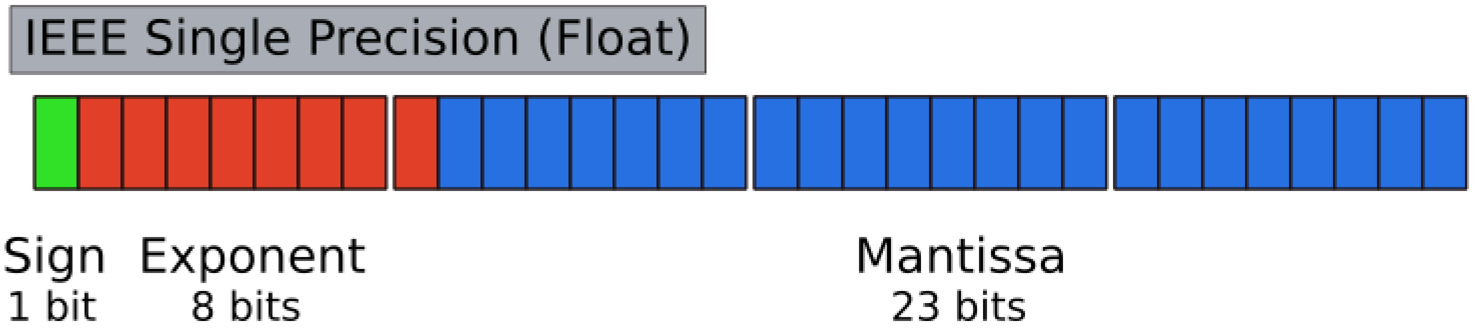
Tricks in the representation

Valid (binary) exponents are -126 to 127
The exponent is "biased", add 127 to real exponent (1-254)
Biased exponent values 0 and 255 are special

The mantissa is "normalized". The leading digit is always 1, e.g. 1.xyz, in binary.
Since the leading digit is always 1, it is assumed and treated as a "hidden bit".

(Also double precision with 1 bit sign, 11 bit expt and 52 bit mantissa)

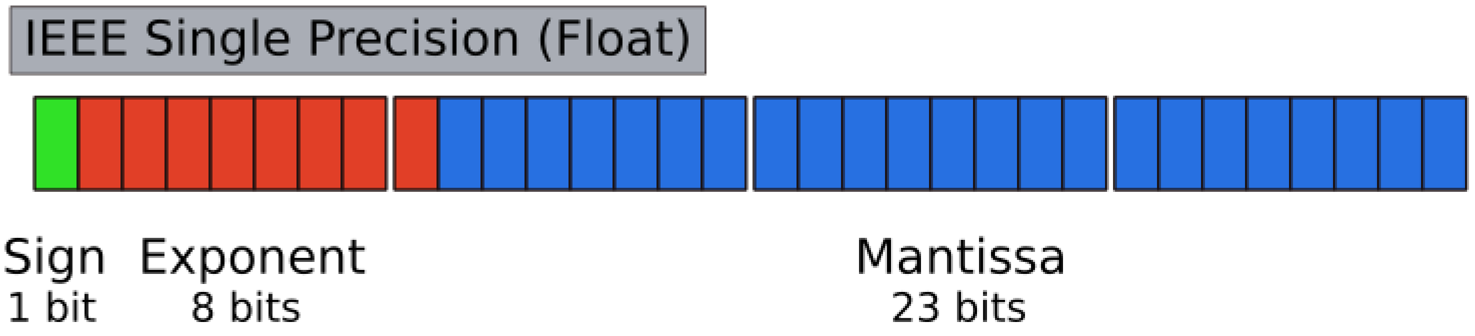
IEEE-754 representation



Special Values in the representation

- Biased exponent = 0, mantissa = 0 represents (signed) 0
- Biased exponent = 0, mantissa != 0 represents "denormal numbers": gradual underflow
- Biased exponent = 255, mantissa = 0 represents (signed) infinity
- Biased exponent = 255, mantissa != 0 represents NaN (Not a Number)

IEEE-754 operations



Operations

Rounding! (Upcoming slides)

Invalid operations like INF/INF, 0/0, INF-INF, 0*INF, sqrt(negative), etc. produce NaN

Five exceptions are specified for errors:

1) Invalid operation - a NaN was produced
2) Divison by zero
3) Overflow
4) Underflow
5) Inexact - rounding

Edge cases – what could go wrong?

First Ariane 5 flight, destroyed just after launch as it veered out-of-control, along with a $400M scientific mission (Cluster).

The subsequent investigation identified the lowest level problem as a software bug coming an arithmetic overflow when converting a 64bit floating point to a 16bit integer, triggering a hardware exception.
Numerous higher level problems also contributed, of course.

Rounding

- Since most real numbers don't correspond exactly to floating point numbers, a mapping (“rounding”) process is required. Non-repeating in decimal doesn't imply the same in binary:
  - 00111111 00000000 00000000 00000000 == 0.5
  - 00111101 11001100 11001100 11001101 == 0.1
- Usual rules of associativity don't (always) apply due to limited precision and rounding
  - (1.0e10 – 1.0e10) + 3.14e-2
  - 1.0e10 – (1.0e10 + 3.14e-2)
- “x=y/c” and “a=1.0/c; x=a*y;” not necessarily numerically the same

Rounding and compiler optimizations

- In general the goal for compiler optimizations should be to modify the code in a way that one obtains the same results
- Due to the issues mentioned on the previous slide and the possibility that intermediate values could generate infinities, underflows or NaN's, what may initially appear to be a simple algebraic transformation of the code may actually change the results.

Implementations – x87 and sse

- Note that there are two floating point units available in the x86 processors (x87 and sse)
- x87 floating point unit
  - Available from i386 onwards, default for gcc/32bit, deprecated for x86_64
  - Registers by default are “double extended” (80bit) format
- sse floating point unit
  - Supports both scalar and vector single and double precision operations (see earlier talk by Sverre)
  - Default for x86_64

Floating Point Rules

- Understand the IEEE standard and what the compiler is doing, e.g. if by default or by options it is making optimizations that affect floating point results
  - Reciprocals instead of division, associativity
- Understand the ranges of your numbers. What precision do you really need?
- Sum from smallest to largest (see also Kahan summation)
- Watch for places where cancellations might occur
- Do not mix single and double precision: especially bad for SSE
- Globally applied solutions and choices are probably not possible (and probably not needed)

32bit vs 64bit

- Another transition that is still ongoing for the large HEP experiments is the transition from 32bit to 64bit. For example:
  - BaBar/CDF/D0 are 32bit only
  - The LHC experiments have (mostly) been using 32bit applications, although all have 64bit builds available and/or deployed in parallel
- One limiting factor has been that WLCG grid sites in the past have deployed a mix of 32bit and 64bit OS on batch worker nodes, even if the CPU's were 64bit-capable.
- As part of the recent/ongoing WLCG transition to SLC5, sites were requested to deploy uniformly the 64bit version of the OS.

64bit history

- Early 64bit processors – Cray, Alpha (DEC), MIPS, Nintendo 64, PowerPC, (now CBE), etc.
- Intel/HP attempted ia64/Itanium
- But in the 21st century we are mostly interested in commodity hardware for bulk, batch-oriented HEP computing
- In practice the 64bit commodity CPU which interests most of us at this point is AMD's AMD64 (and Intel's EM64T implementation), generically x86_64

64bit performance advantages

- Larger addressible memory space (>4GB)
  - In HEP, at least, this is not relevant for most bulk batch-oriented data processing applications (MC simulation, reconstruction, etc.)
  - Some specialized applications may benefit, e.g. tracking alignment, standalone fitting, perhaps also interactive analysis
- Increased number, and size, of general purpose registers
- Increased number of SSE registers
- Instruction pointer relative addressing: reduced cost of position independent code (shared libraries)

64bit applications – issues

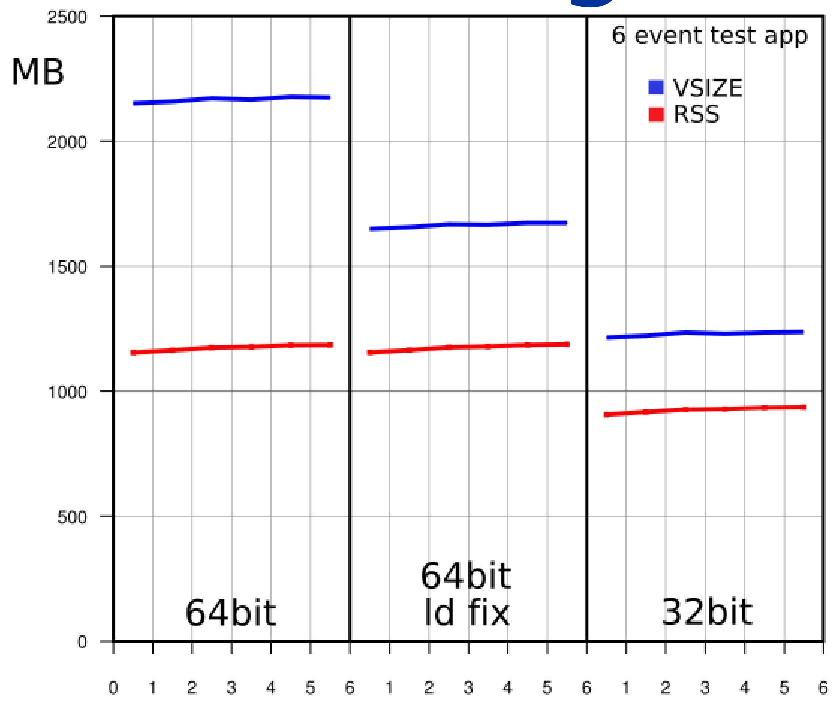
- There are numerous webpages out there about porting applications to 64bit and YMMV in any case, but examples of problems include:
  - Assumptions about the size of pointers and/or types and their interchangeability (lots of flora and fauna here)
  - Use of variable size types (e.g. size_t) for persistent/stored classes
  - Memory use issues (and confusion, see next slides)
  - Floating point math changes – The use of the sse fpmath (see earlier slide) is default for x86_64/gcc. This may or may not be a change for you, depending on the options you use for your 32bit software builds

64bit – memory accounting

- The first thing to note when looking into the memory use of your 64bit applications is that using the VSIZE, even as a rough metric, is becoming meaningless.
- Unfortunately it is still true that a number of tools (shells, queue systems, etc.) as well as people's naive expectations, use VSIZE for resource accounting.
- The combination of these two things has given 64bit a worse reputation than it deserves in some quarters.
- The next slide gives two specific 64bit examples, but “How much memory is my job using?” also appears as we move to multicore (also for RSS, etc.).

64bit – memory accounting

- Although the VSIZE doubles for the 64bit applications, the RSS increase is much more modest (25-30%).
- Most of the VSIZE/RSS difference between 32bit and 64bit comes from a default 1MB alignment of data/text pages, imposed by ld for 64bit, visible with pmap as memory segments with no permissions.
- A similar issue happens for the mapping of the locale file: for 32bit it is mapped into memory in 2MB slices, for 64bit the entire file is memory mapped (50MB+)



- CMS example: with a custom linker script, ~500MB can be removed by relinking the numerous and very small libs (see plot). Another ~100MB would be possible from linking external libs in the same way. (Workaround to keep systems using VSIZE happy for now...)

64bit memory – the increase

- There are some real increases in the memory use, however:
  - Pointer sizes increase from 4 bytes to 8 bytes, with corresponding increases in data structures
  - Alignment padding may increase the size of data structures
  - The actual code (text/data) size itself also increases by a small amount from 32bit to 64bit (YMMV, CMS saw ~5% increase)
  - The heap allocation overhead/alignment cost for 64bit is twice that of 32bit. For 5-10M live allocations in the heap (observed in HEP applications!) this itself implies an extra 40-80MB for our applications, plus (one copy of) the 64bit pointers themselves imply an extra 20-40MB.

64bit – Getting from here to there

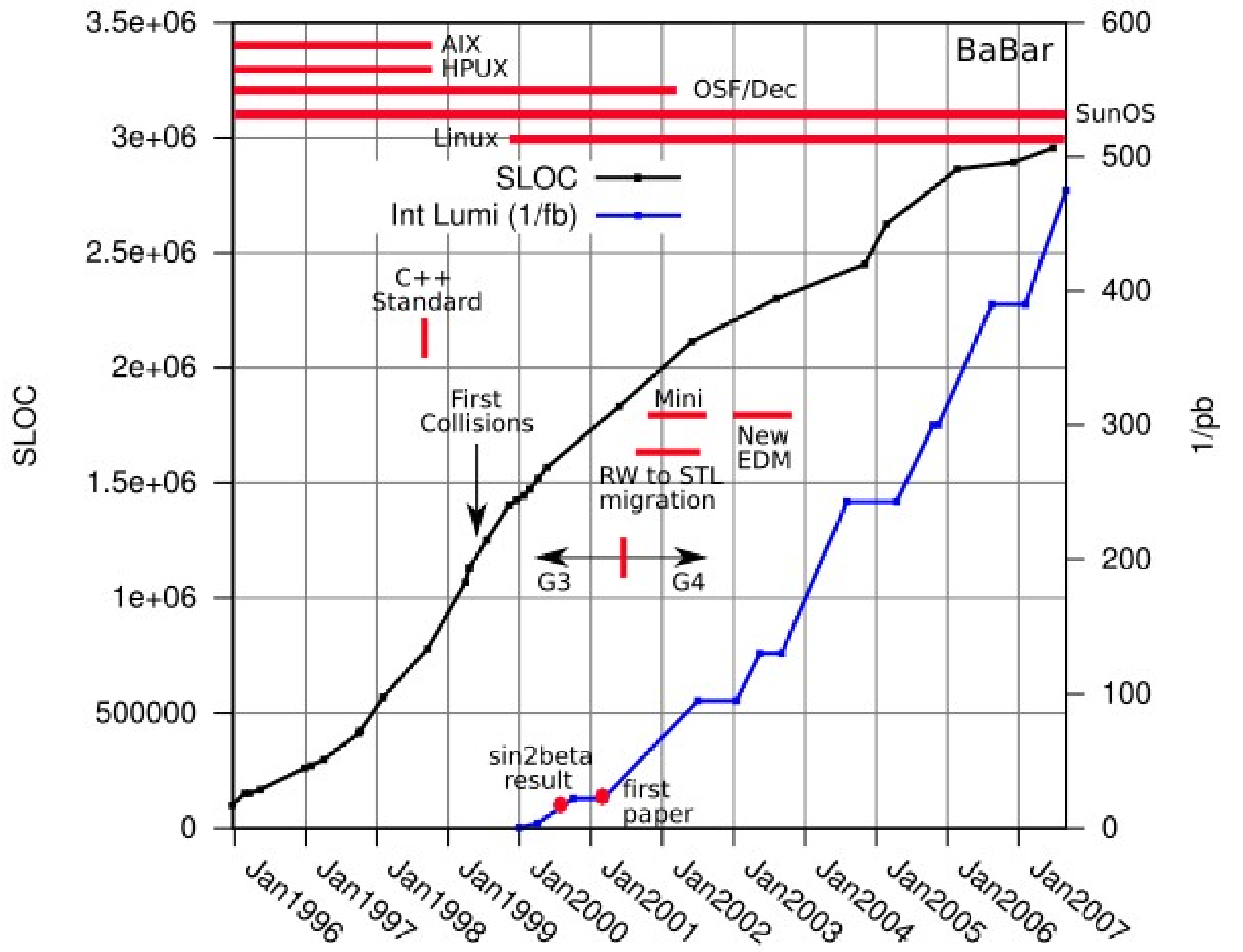
- Many of the memory (footprint) issues are in the end the same ones we see for 32bit, so can be improved by the same techniques
- At least within WLCG we are in the process of providing uniform 64bit computing, enabling the experiments to transition fully should they like
- A full transition to 64bit (only) by an experiment may mean cutting off support for very old hardware (e.g. some laptops)

Code Evolution and Maintainance

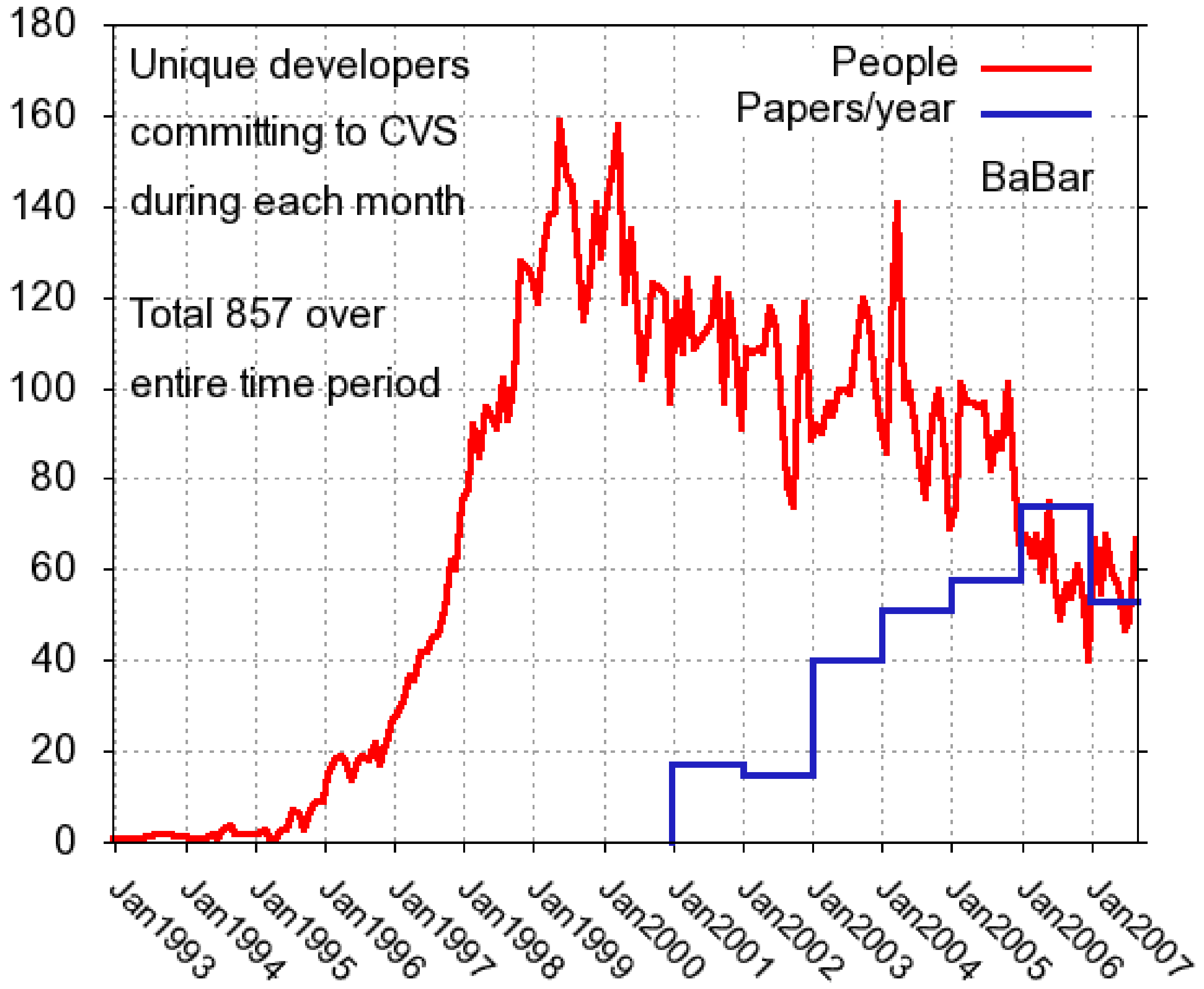
- Certain types of changes made for improved performance, e.g.
  - Changes to use multithreading in applications
  - Widespread changes to floating point behaviour
  - Etc.

  may impact the software development model.
- Making such changes requires validation at the time they are deployed and in fact continued validation may be needed as the software evolves, simply because they can break the simple model used by developers

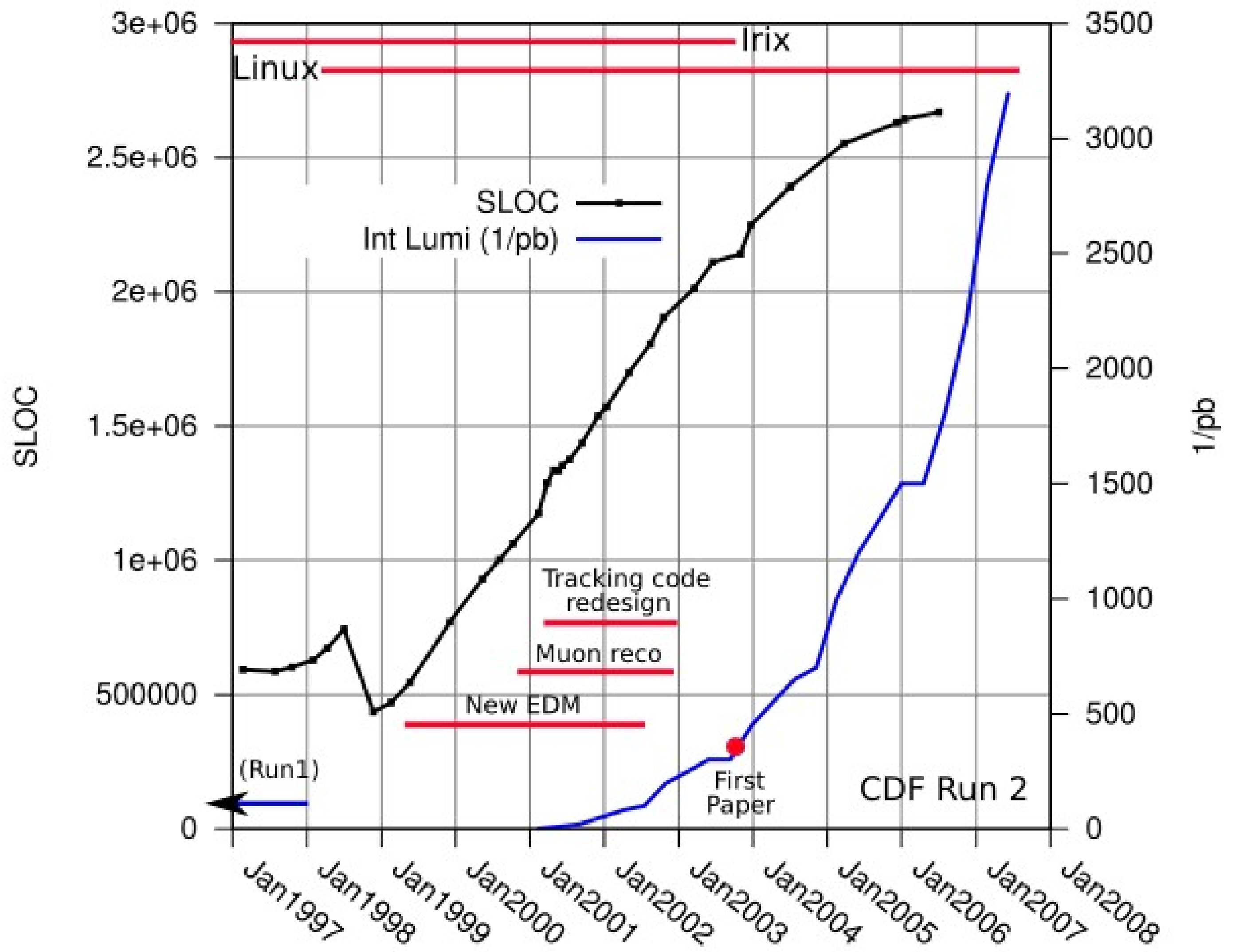
Code evolution – BaBar



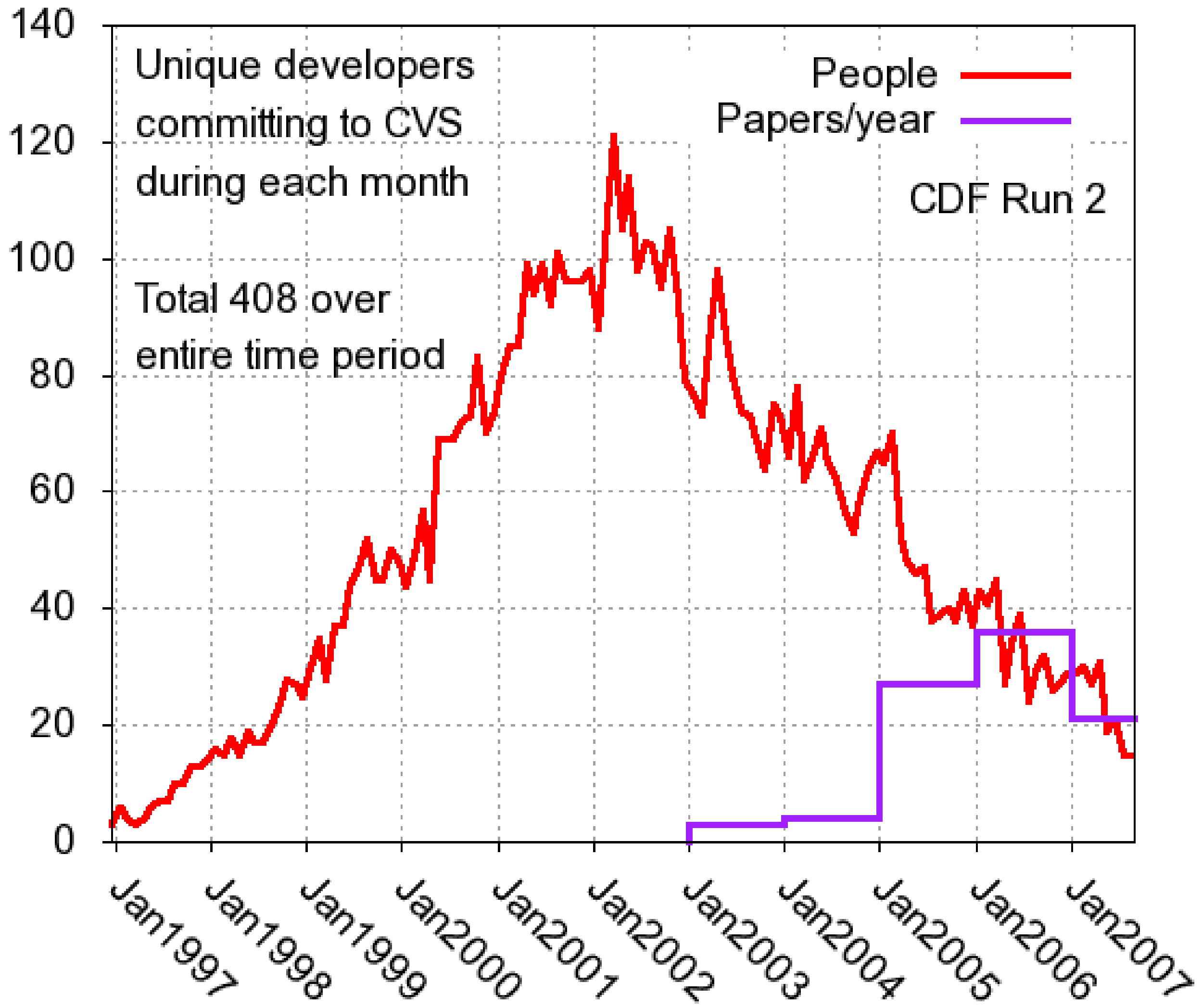
Code evolution – BaBar



Code Evolution – CDF Run II



Code Evolution – CDF Run II



Summary

- A number of interesting transitions are underway, bringing new opportunities and challenges!

References

- gcc release notes:
  - http://gcc.gnu.org/gcc-4.1/
  - http://gcc.gnu.org/gcc-4.2/
  - http://gcc.gnu.org/gcc-4.3/
  - http://gcc.gnu.org/gcc-4.4/
- gcc optimization options:
  - http://gcc.gnu.org/onlinedocs/gcc-4.4.1/gcc/Optimize-Options.html
- Many interesting articles at gcc summits
- LLVM: http://www.llvm.org/
- Intel compilers: http://software.intel.com/en-us/intel-compilers/

References

- Ulrich Drepper, “How to Write Shared Libraries”, http://people.redhat.com/drepper/dsohowto.pdf (along with most other notes)
- Ian Lance Taylor, “Linkers”, 20 blog entries in http://www.airs.com/blog/archives/38 and follow-ons
- David Goldberg, “What Every Computer Scientist Should Know About Floating-Point Arithmetic”
- David Monniaux, “The Pitfalls of Verifying Floating-Point Computations”, http://hal.archives-ouvertes.fr/docs/00/28/14/29/PDF/floating-point-article.pdf