Geant4 V10.0 Performances: A study of build options

A. Dotti

12th January 2013

Geant4/SFT Meeting - CERN

U.S. DEPARTMENT OF ENERGY
Office of Science

SLAC NATIONAL ACCELERATOR LABORATORY

Introduction

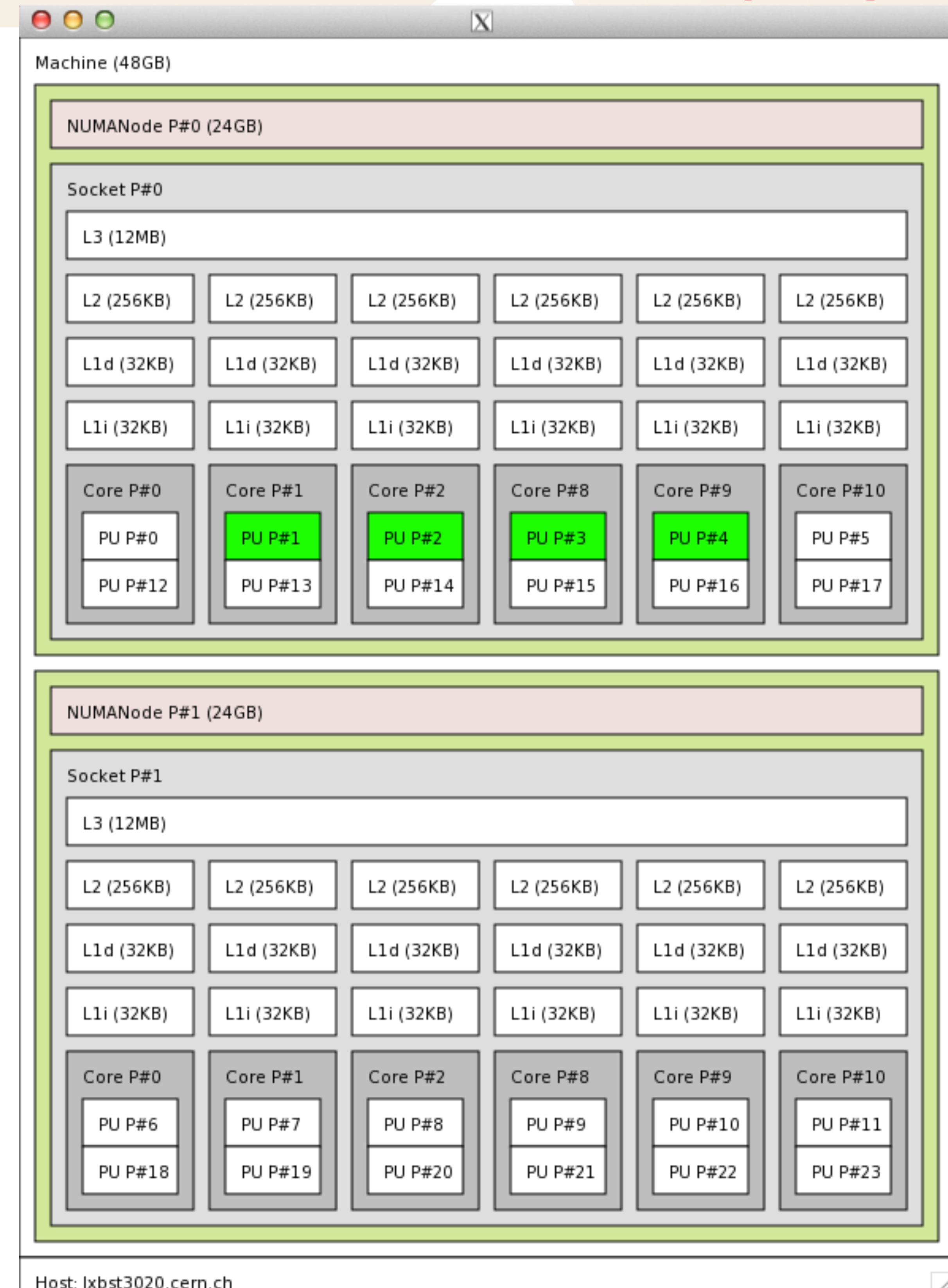
SLAC

- Some studies I've started after the very interesting findings from Vladimir Ivantchenko
  - http://indico.cern.ch/conferenceDisplay.py?confId=287282

- In particular I (we) were very interested to the Static/Shared libraries differences

Testing setup

- Using "performance" machine from Openlab
- Equipped with 2x Intel Xeon E-5650 @ 2.67 GHz
  - Westmere-EP architecture
  - 6 cores (+HT)
  - http://ark.intel.com/products/47922/Intel-Xeon-Processor-X5650-(12M-Cache-2_66-GHz-6_40-GTs-Intel-QPI)
- **Geant4 Version 10.0 codebase**
  - out-of-the-box, with cmake, 64bits
- **Running ParFullCMS application**
  - 6 threads locked on first cpu (hwloc V. 1.7.2)
  - GDML support from local Xerces-C (V. 3.1.1) installation
  - 50 GeV pions in random direction
  - FTFP_BERT physics list
  - B-Field on

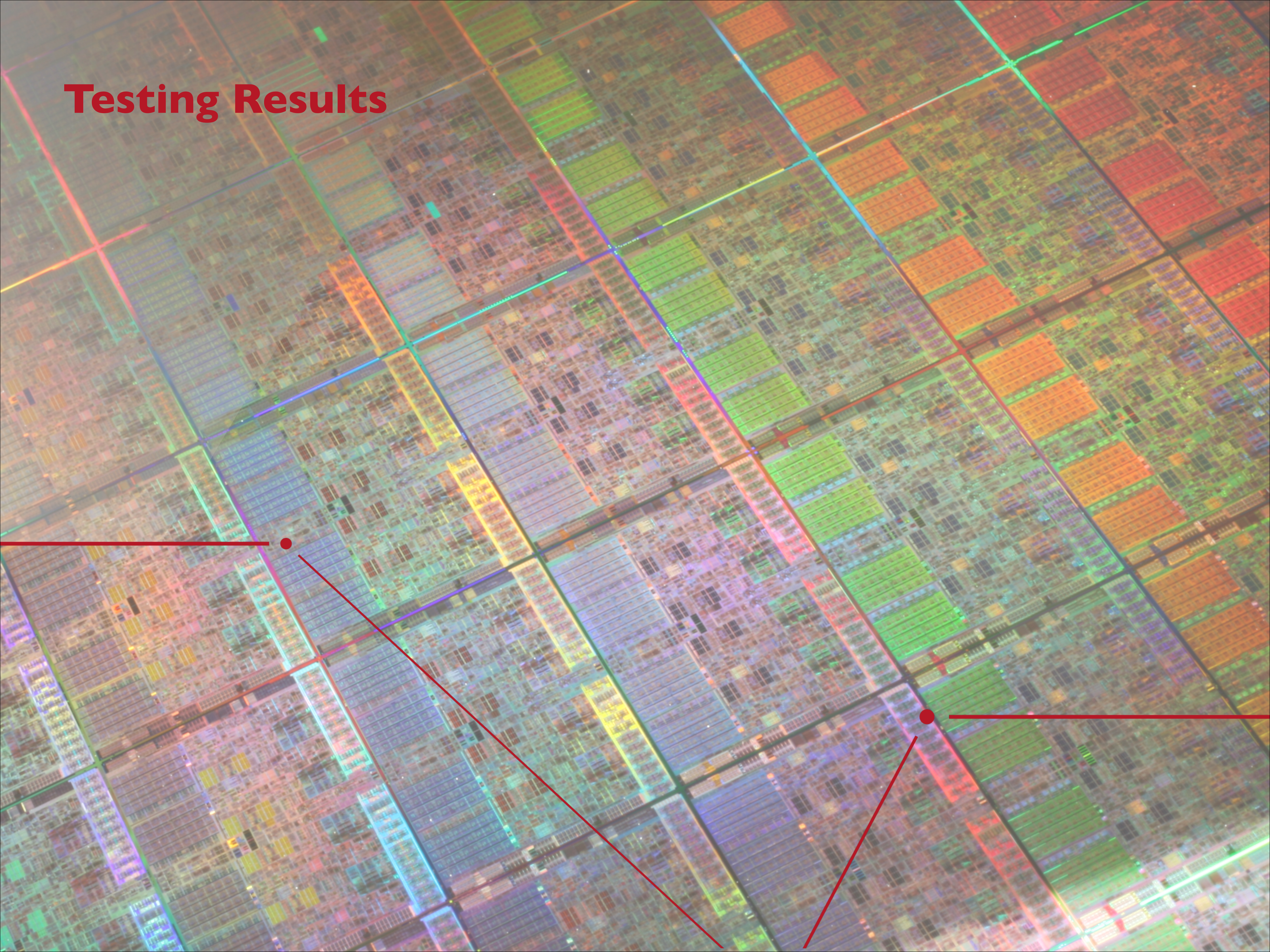


What to measure

SLAC

- measure time spent in event loop (e.g. exclude init, final)
  - 100 Events/thread
  - Report **throughput**: events/minute/thread
- Change RNG seed at each run
- Repeat test several times, **measure averages**
- Measurements are **not gaussian**: some events take longer than average (over fluctuation of hadronic component of showers)
- Estimate intrinsic precision is about 1% (worse w/o pinning)
- Whenever not specified using default compilation options: **-O2, tls-model=initial-exec -fPIC, Release** build type*

* See backup slides

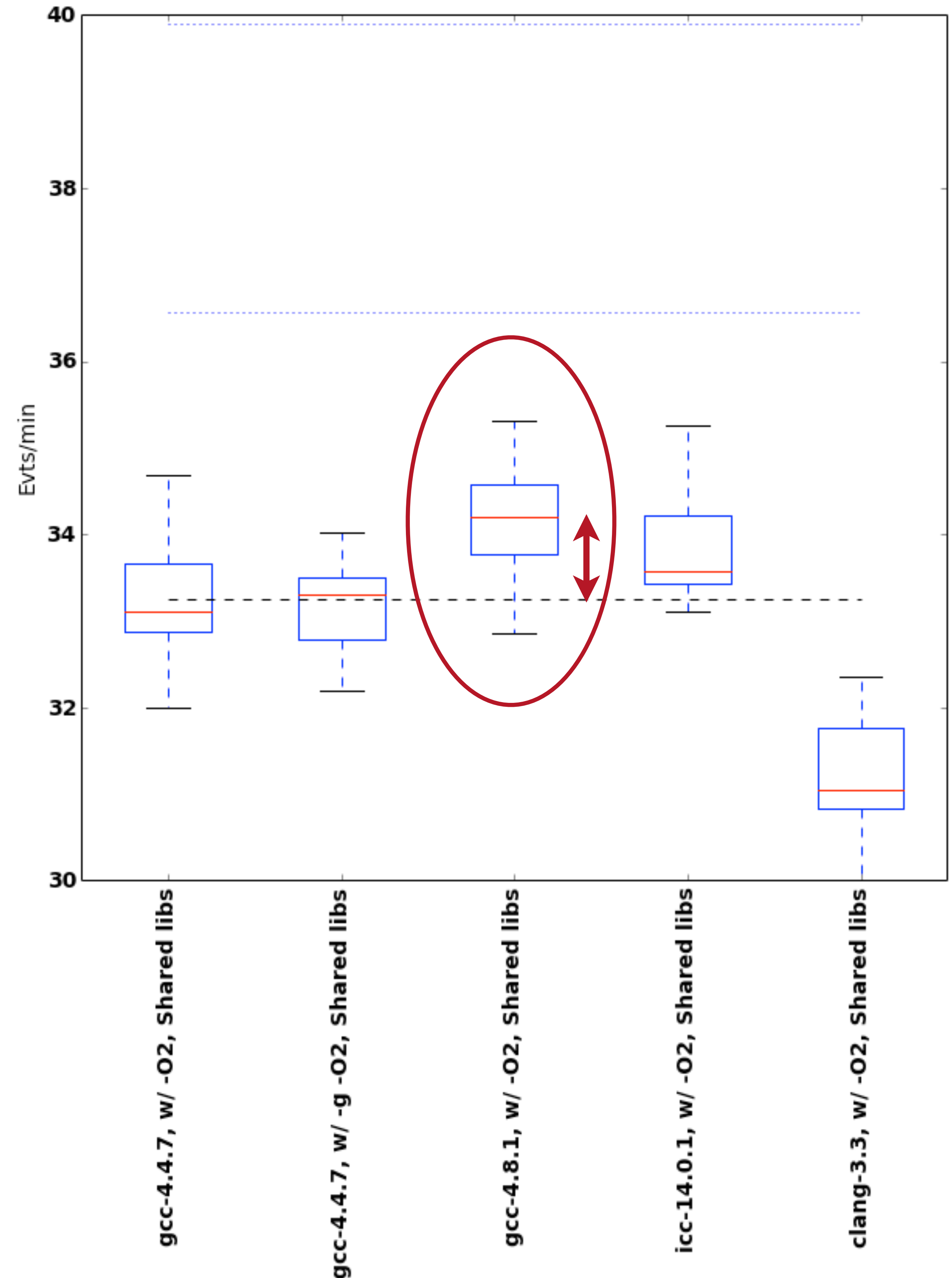

Testing Results

Compiler

SLAC

- GCC 4.4.7 baseline
- Via AFS:
  - GCC 4.8.1
  - CLANG 3.3
  - ICC 14.0.1

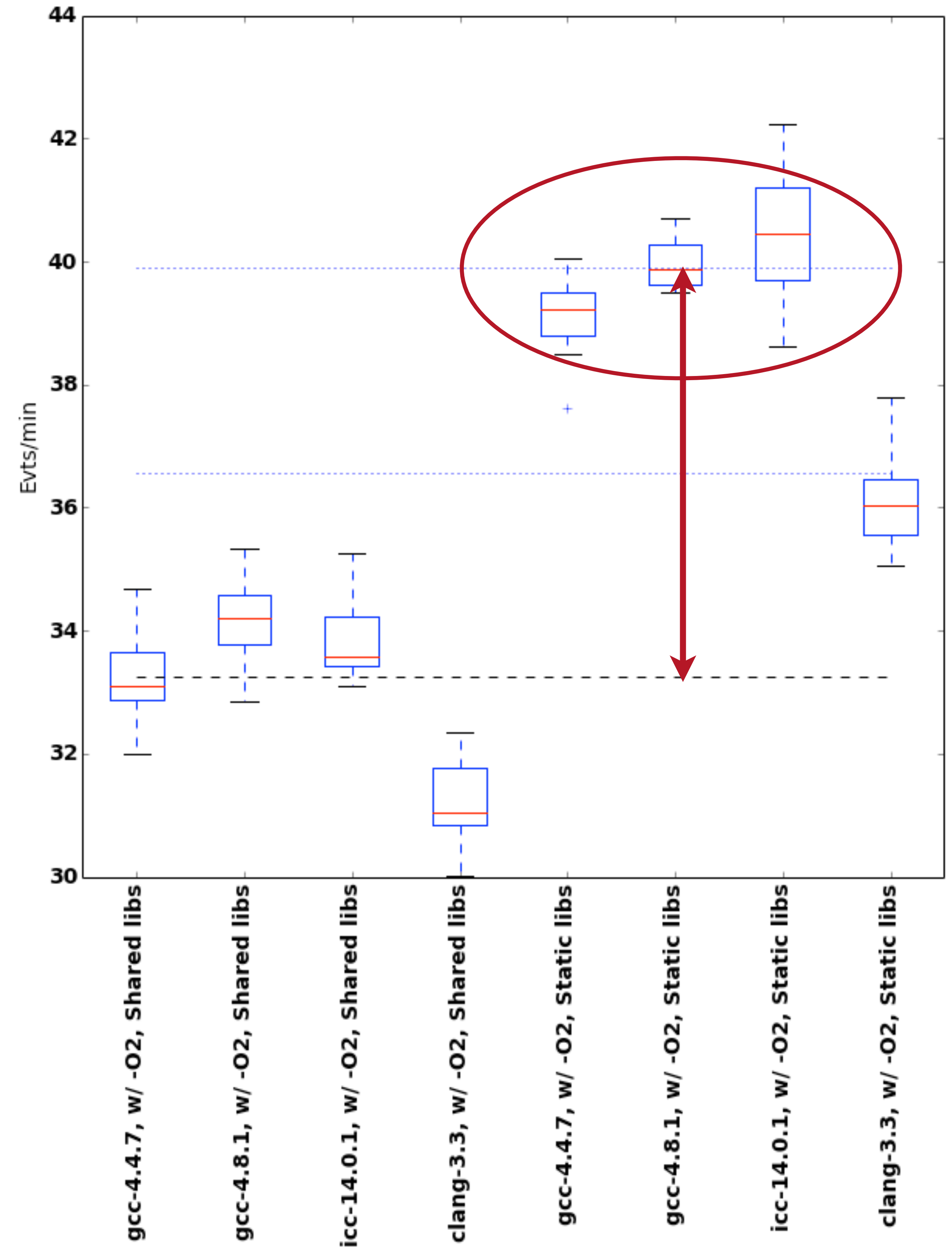
- **GCC 4.8.1**
  - **2.6% faster than baseline**
- CLANG 3.3
  - produces code substantially slower -6%
- Release or RelWithDebInfo are basically the same



Static Vs Shared

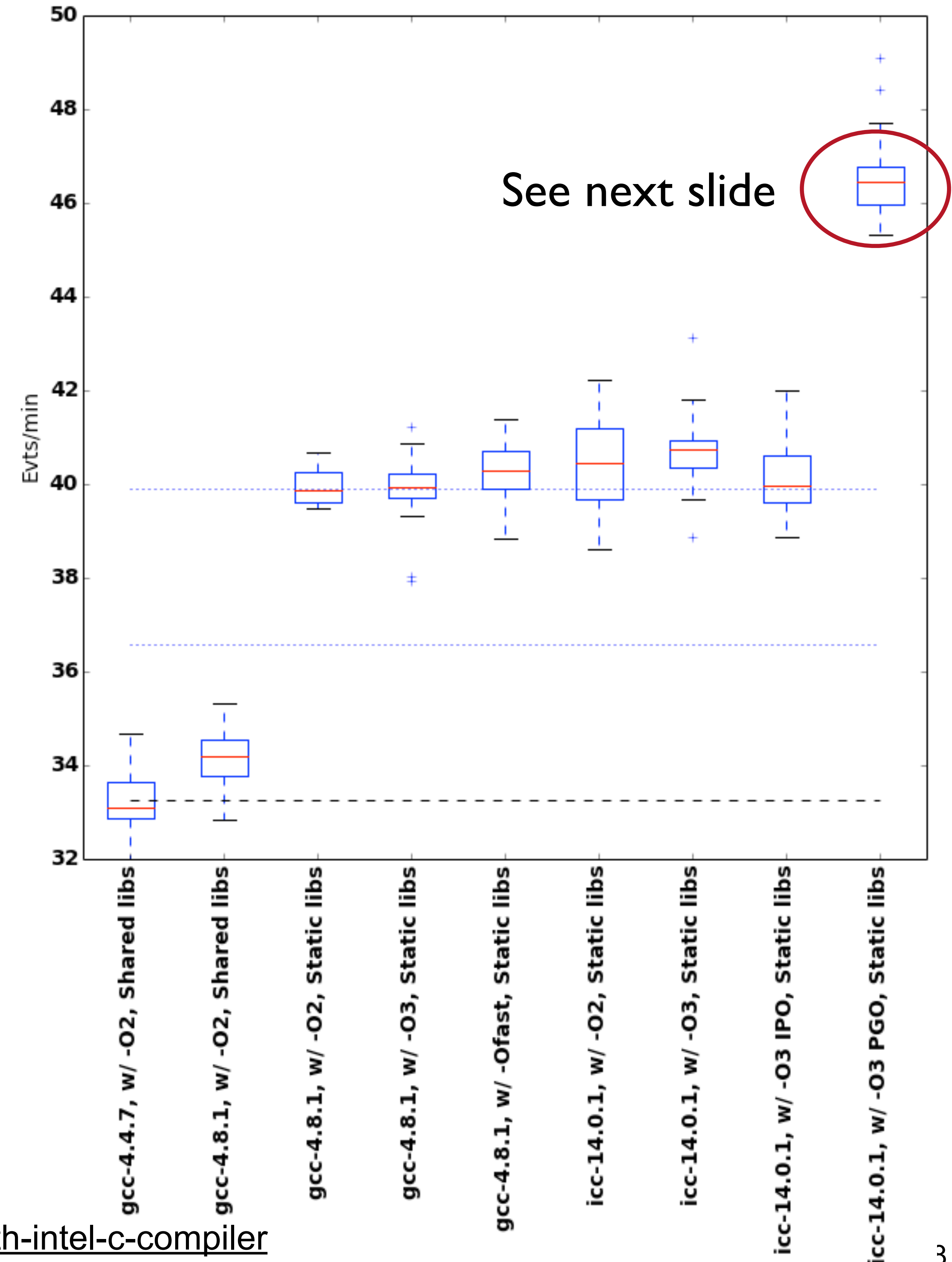
SLAC

- Confirmed Vladimir's finding
- **Static libraries builds are substantially faster: +20%**
- Hierarchy between compilers is preserved



Optimization levels

- Reminder: -O2 default options (note: meaning is different between compilers)
- -O3 does not bring improvements
- Neither does -Ofast (GCC for fast math)
  - Warning: may violate IEEE and ANSI standards
- neither does IPO
  - Exists also for GCC, not tested



http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
http://software.intel.com/en-us/articles/step-by-step-optimizing-with-intel-c-compiler

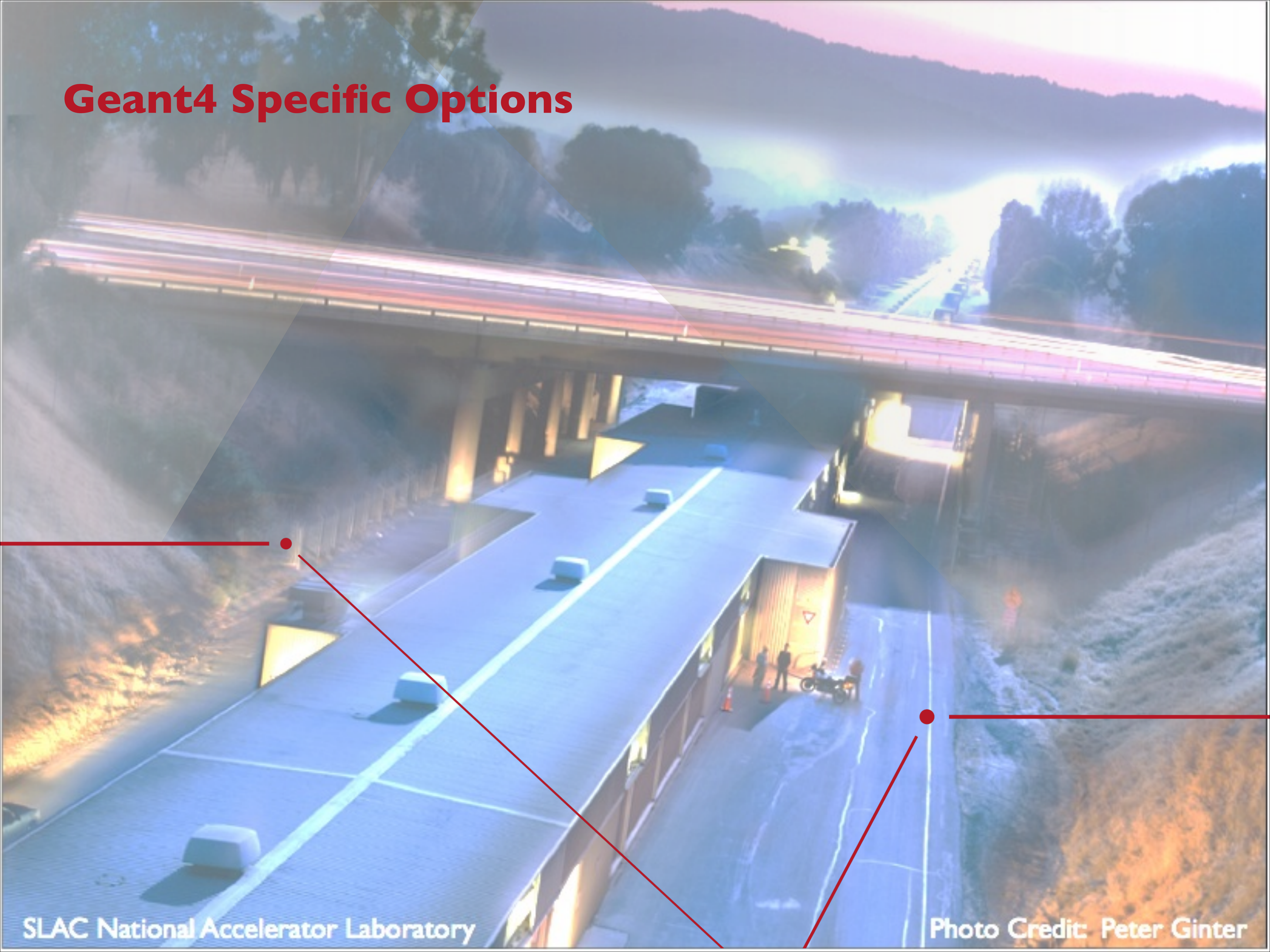
A note on PGO

SLAC

- **Profile Guided Optimization brings another 20%**
- How does it work:
  - Run once with instrumented code and produce profiles
  - Recompile code using results of previous step to push optimizations (e.g. improve branch predictions)
- Seems wonderful **but**:
  - Can produce wrong numeric results (not tested)
  - Need to know hw where process will run
    - Tried to compile with PGO on Nehalem CPU and run on Westmere, no gain (but no loss either...)
  - Clearly is application specific. What happens changing primary?

  - It's probably worth some further investigation
  - GCC also supports this, not tried

Geant4 Specific Options



TLS Model 1/2

SLAC

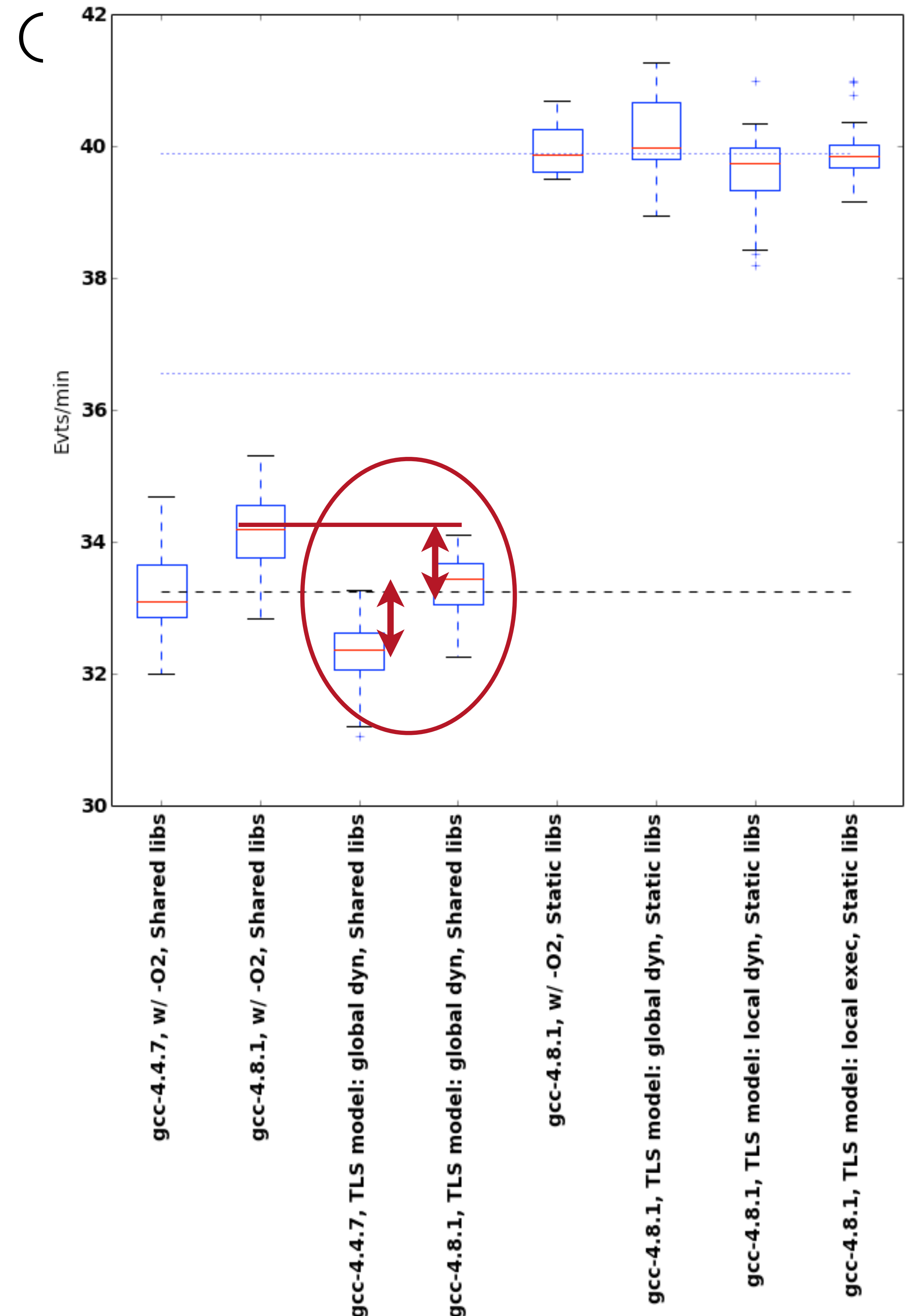
- For thread-local-storage four options exist. From most general to most optimized:
- **global-dynamic**
  - It allows TLS in dynamic loaded modules (e.g. dlopen)
  - Slow because it performs checks and creates TLS "blocks" at runtime (tls_get_addr function)
- **local-dynamic**
  - Optimized version of previous code, restrictions apply (intra-module access to TLS variables)
  - For Geant4 it produces broken code with Shared libraries
- **initial-exec**
  - Restrictive, optimized code. Default in Geant4
  - Avoid use of tls_get_addr assuming at program startup TLS space can be pre-prepared
  - dlopen ok only if all TLS static are uninitialized and fit in 512 bytes (!)
- **local-exec**
  - Most restrictive TLS code, fastest
  - All TLS variables defined in the executable
  - Geant4 does not compile with Shared libraries

http://software.intel.com/sites/products/documentation/doclib/iss/2013/compiler/cpp-lin/GUID-6ABFB39C-A478-4974-9A38-170AA13BC623.htm

TLS Model 2/2: Results

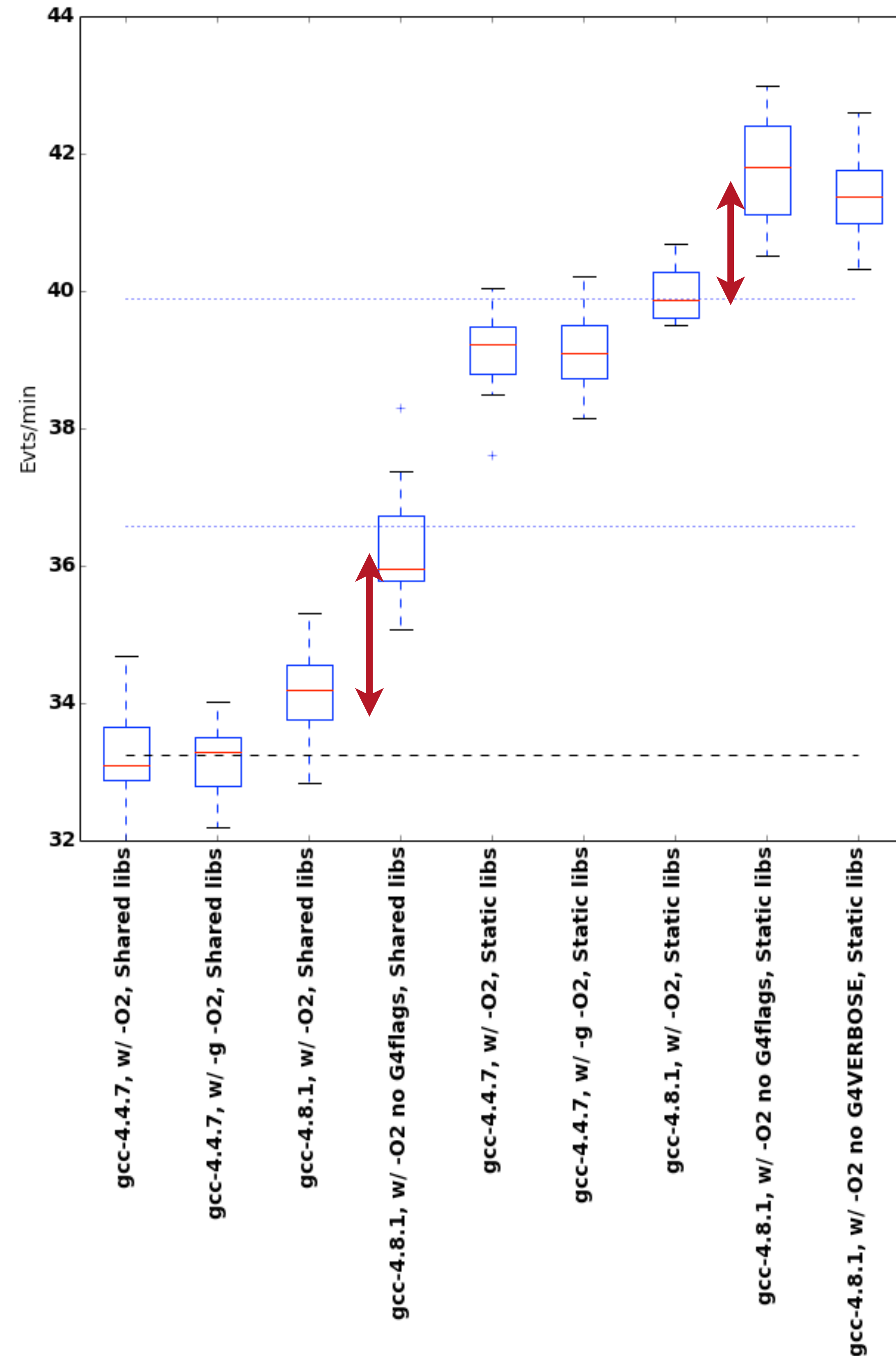
- With static libs there are no differences
  - **Warning**: compiler may have used always local-exec (optimization)
- For shared libraries global-dynamic gives **few percent CPU penalty** (not 20% reported two years ago, why this difference?)

- Side Note (for CMS): with dlopen can use **only** global-dynamic



Other Flags

- The following two flags are active: **G4VERBOSE** and **G4_STORE_TRAJECTORY**
- Can be switched off. Drawbacks:
  - Fewer run-time checks of simulation correctness and less info in case of error
  - No visualization of events
- **+5-6% CPU improvements**
  - Mainly from G4VERBOSE



Conclusions

SLAC

- With the following combinations:
  - Using static libraries (**main contribution**)
  - Using GCC 4.8.1
  - Turning off G4VERBOSE and G4_STORE_TRAJECTORY
- **25.7% additional throughput**

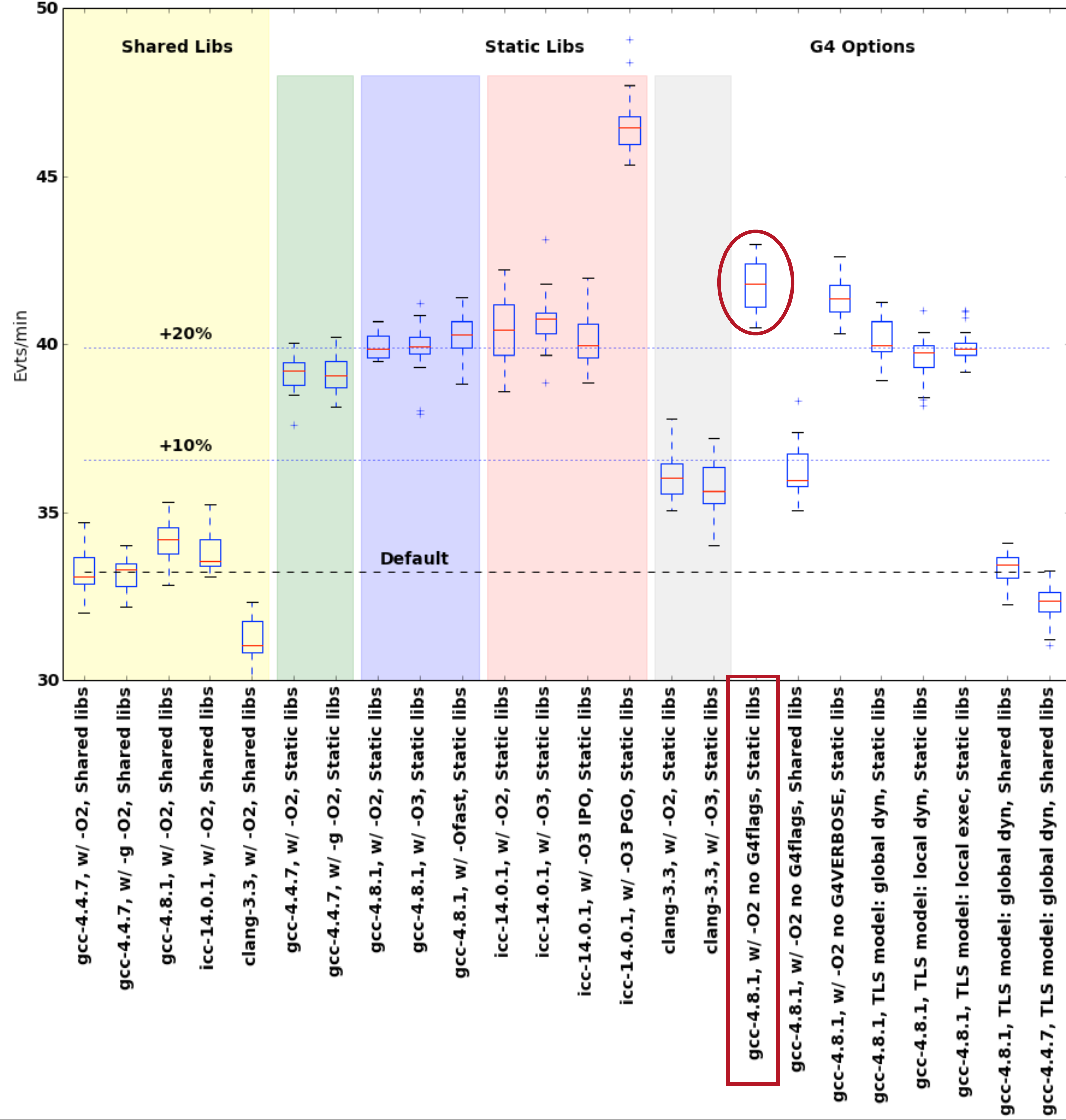
- Considering the possible large improvement user communities should consider the adoption of these options

- Note: In addition GPO provides additional large boost, but detailed studies on the validity of the results are needed. It may even require rethinking of how production are carried out (e.g. hw specific builds)

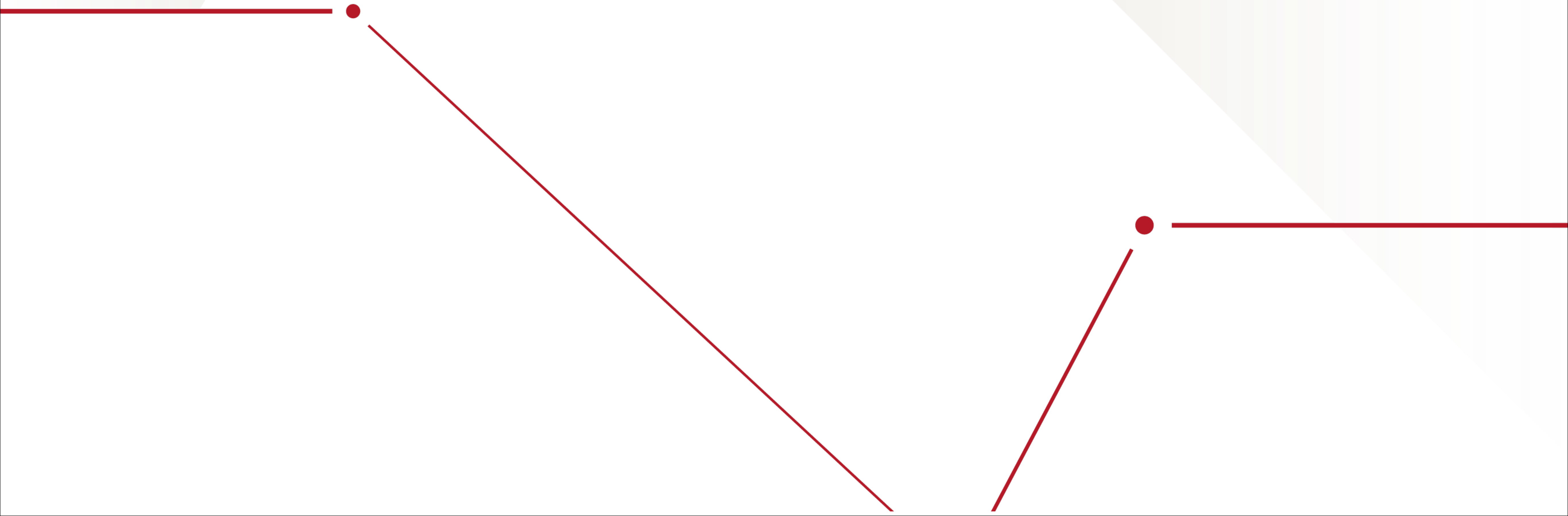
Overview

SLAC

- Compilers
- GCC 4.4.7
- GCC 4.8.1
- ICC 14.0.1
- CLANG 3.3
- G4 options



Backup



Compilation Option: Default

SLAC

- **-DG4_STORE_TRAJECTORY -DG4VERBOSE**
- -DGEANT4_DEVELOPER_RELEASE -W -Wall -pedantic -Wno-non-virtual-dtor -Wno-long-long -Wwrite-strings -Wpointer-arith -Woverloaded-virtual -Wno-variadic-macros -Wshadow -pipe **-ftls-model=initial-exec**
- **-pthread -std=c++98 -O2 -DNDEBUG -fPIC** [... module specific -D...]

- **Note on -fPIC:** this is mandatory for shared libraries in 64-bits. In 32-bits its use may introduce an overhead that is not present in 64-bits. When using static libraries we do not use -fPIC