U.S. DEPARTMENT OF
ENERGY

UNIVERSITY OF
CALIFORNIA

BERKELEY LAB

LAWRENCE BERKELEY NATIONAL LABORATORY

U.S. DEPARTMENT OF ENERGY

Benchmarking C++ Code

Bryce Adelstein Lelbach aka wash <balelbach@lbl.gov>

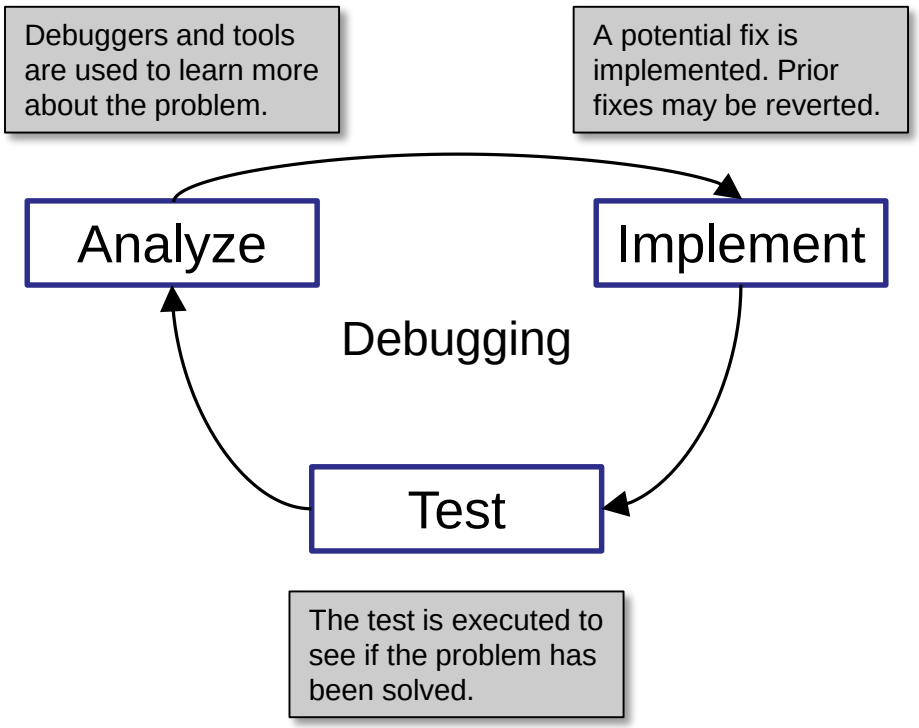
Computer Architecture Group, Computing Research Division

CppCon 2015

The Problem with Performance

Problem: **Code seg faults**

- We solve this type of problem with an iterative workflow.
- We know when we're done; we can easily get a *"yes/no"* answer during the testing phase.
  - Usually, there's no random error when testing for this type of problem (excluding race conditions).

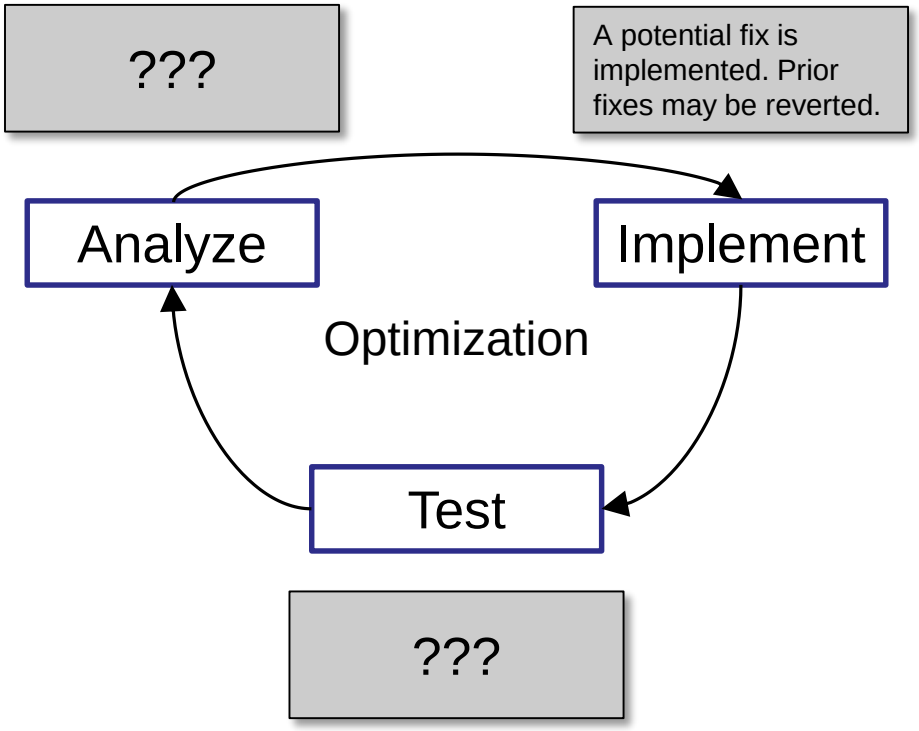


Copyright (C) 2015 Bryce Adelstein lelbach

The Problem with Performance

Problem: **Code is slow**

- Producing a *"yes/no"* answer during the testing phase is more difficult.
  - Performance is not a Boolean quantity.
    - It is often unclear when the problem is "fixed".
    - You never really finish optimizing.
  - Performance data is subject to **random error** due to natural variability.



Copyright (C) 2015 Bryce Adelstein lelbach

What is Performance?

How do we define performance, anyways?

- Not “fast”, but “fast enough”.
- Real-world metrics:
  - Ex: simulation-years/day
- Roofline:
  - Ex: FLOP/s
- Deadline:
  - Ex: takes 50 milliseconds

You need to be able to come up with meaningful definitions for performance.

Copyright (C) 2015 Bryce Adelstein lelbach

BERKELEY LAB

Sources of Error

Observational Error: The difference between what you measure and the true result.

- Random Error: Errors caused by natural variance.
- Systemic Error: Errors caused by an inaccuracy – usually constant or proportional to the true result.

Observational error is unavoidable. Meaningful performance analysis **must** account for error.

- E.g. statistical testing approach

Copyright (C) 2015 Bryce Adelstein lelbach

Variance

Computers can reproduce answers, not performance.

- Hardware jitter
  - Instruction pipelines: The pipeline fill level has an effect on the execution time for one instruction.
  - Difference in CPU/memory bus clock cycles: The CPU clock cycle is different from the memory bus clock speed. Your CPU sometimes has to wait for the synchronization of memory accesses.
  - CPU frequency scaling and power management: These features cause heterogeneities in processing power.
  - Shared hardware caches: Caches shared between multiple cores/threads are subject to variance due to concurrent use.

Source: http://www.chronox.de/jent/doc/CPU-Jitter-NPTRNG.html

Copyright (C) 2015 Bryce Adelstein lelbach

Variance

Computers can reproduce answers, not performance.

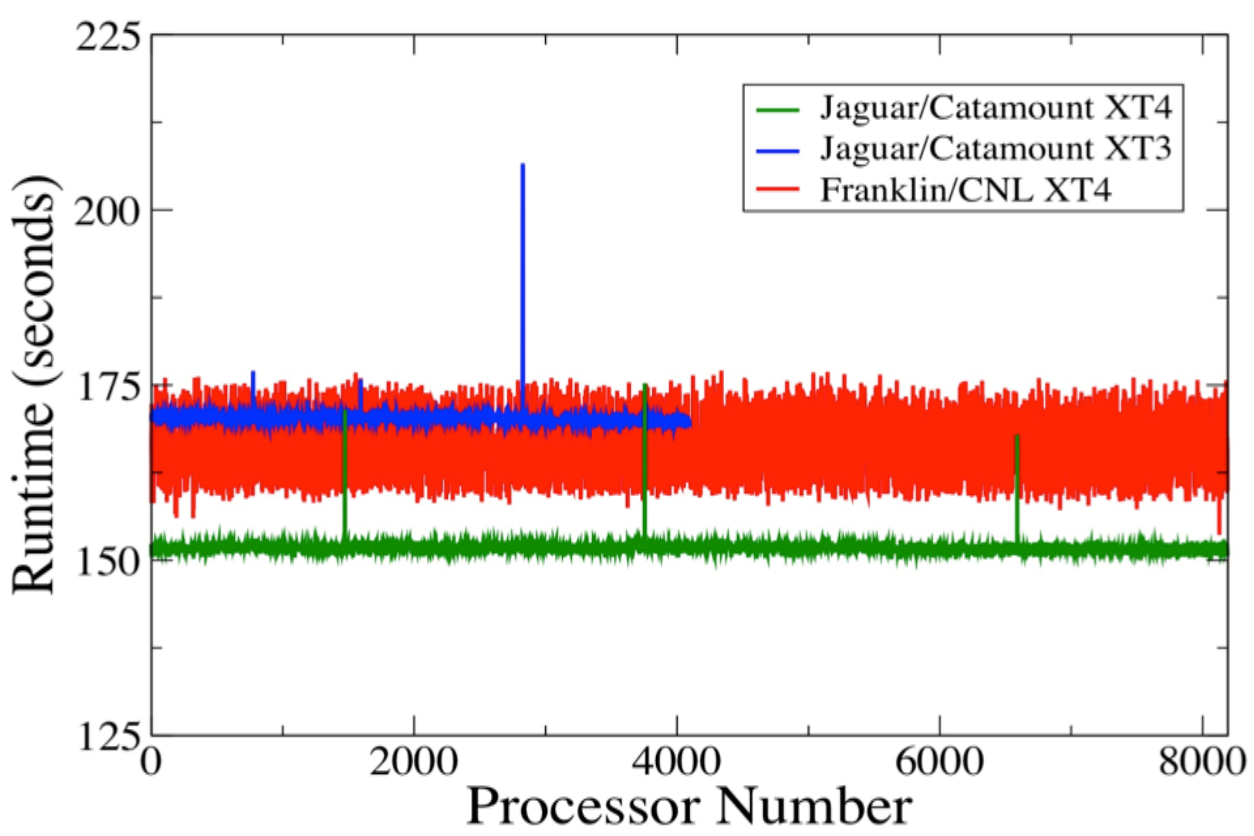
- Larger memory segments may have variance in access times due to physical distance from the CPU.
- Additionally, OS activities can cause non-determinism.
  - Some hardware interrupts require OS handling immediately after delivery.
  - Migration of non-pinned processes can affect the performance of CPU heuristics.

Observer Effect: all forms of instrumentation change the results.

Source: http://www.chronox.de/jent/doc/CPU-Jitter-NPTRNG.html

Copyright (C) 2015 Bryce Adelstein lelbach

Variance



Source: Cy Chan, John Bachan

Copyright (C) 2015 Bryce Adelstein lelbach

BERKELEY LAB

Statistical Best Practices

BERKELEY LAB

Statistical Best Practices

Statistics: A great way to lie to yourself.

Copyright (C) 2015 Bryce Adelstein lelbach

BERKELEY LAB

Statistical Best Practices

Statistics: A way to extract conclusions from your data

Copyright (C) 2015 Bryce Adelstein lelbach

BERKELEY LAB

Statistical Best Practices

Statistics: The science of data…

collection

analysis

interpretation

presentation

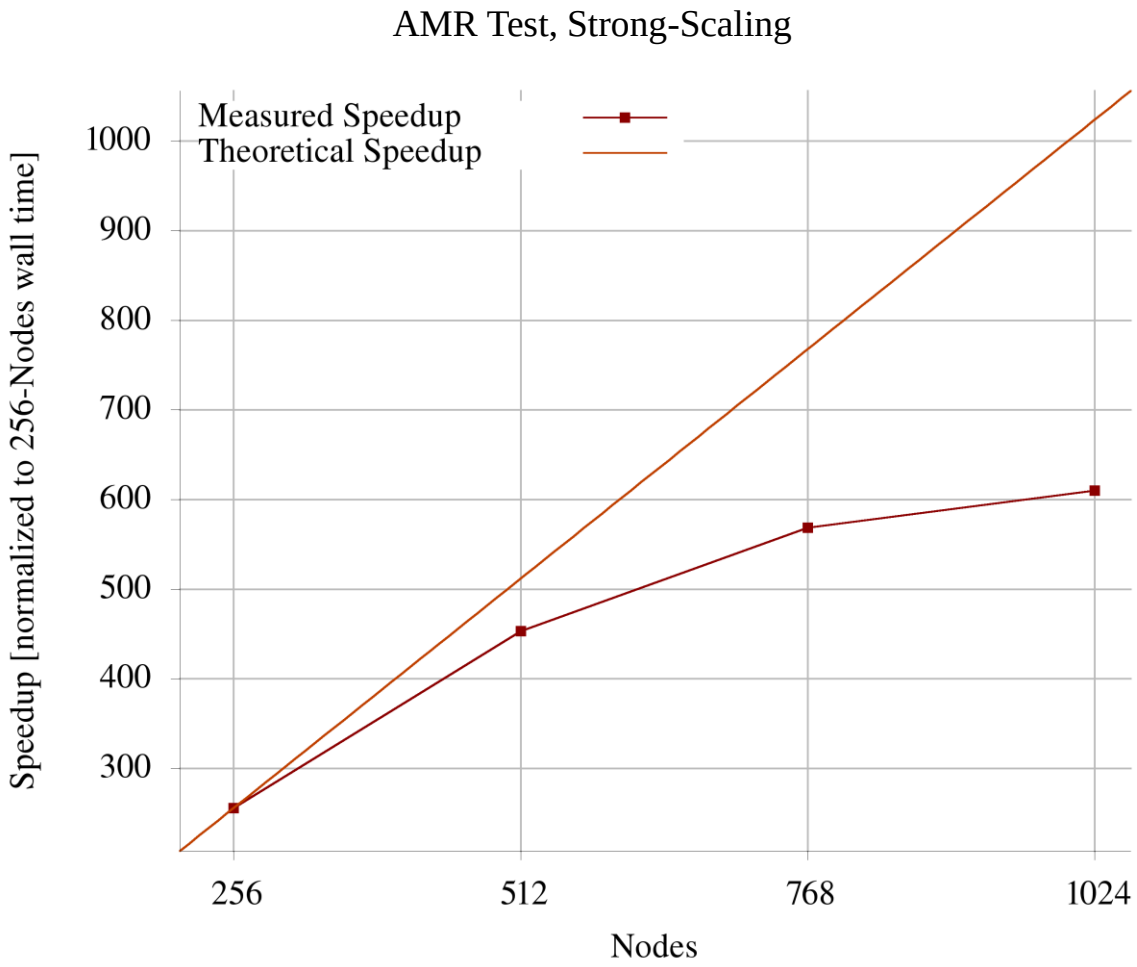
Copyright (C) 2015 Bryce Adelstein lelbach

Case Study: CFD AMR Scaling

Statistical Best Practices

BERKELEY LAB

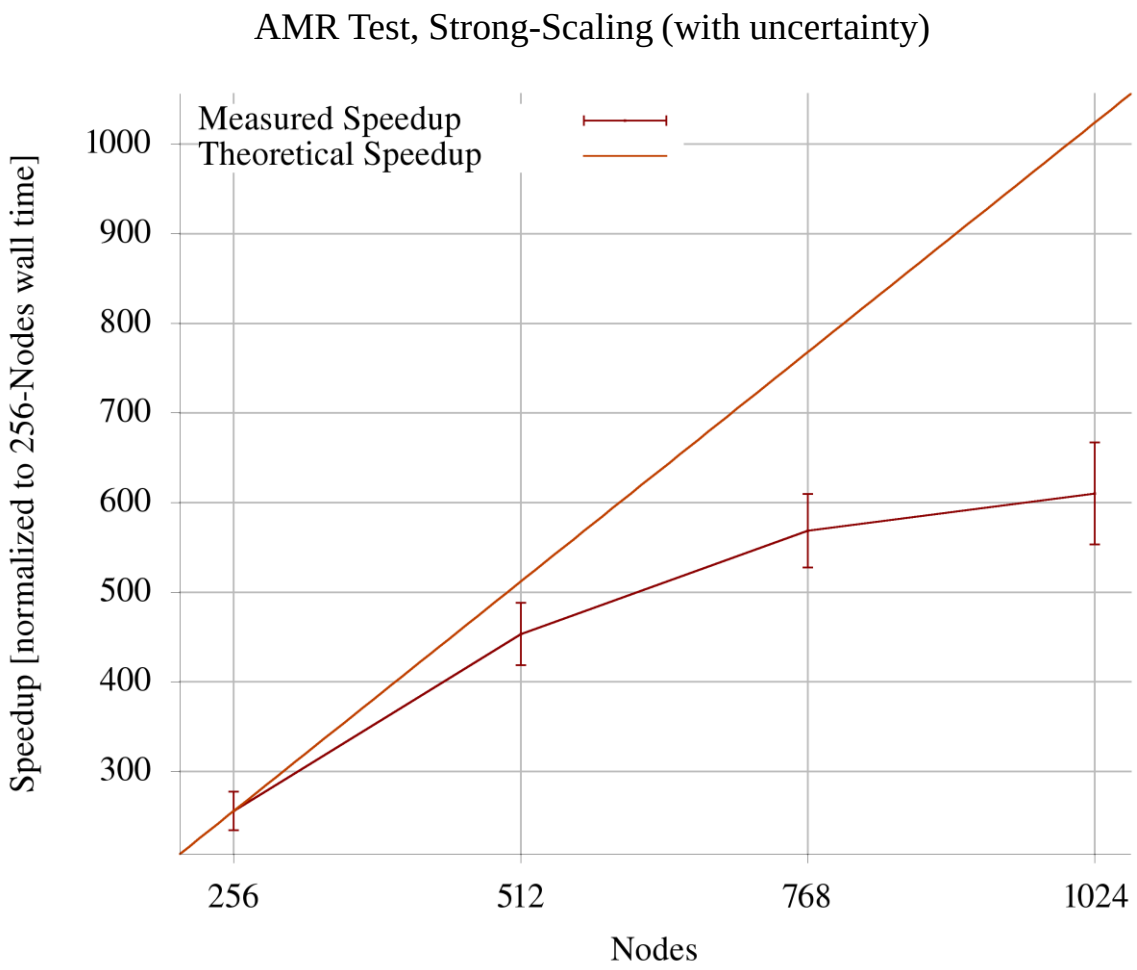
Case Study: CFD AMR Scaling



Copyright (C) 2015 Bryce Adelstein lelbach

BERKELEY LAB

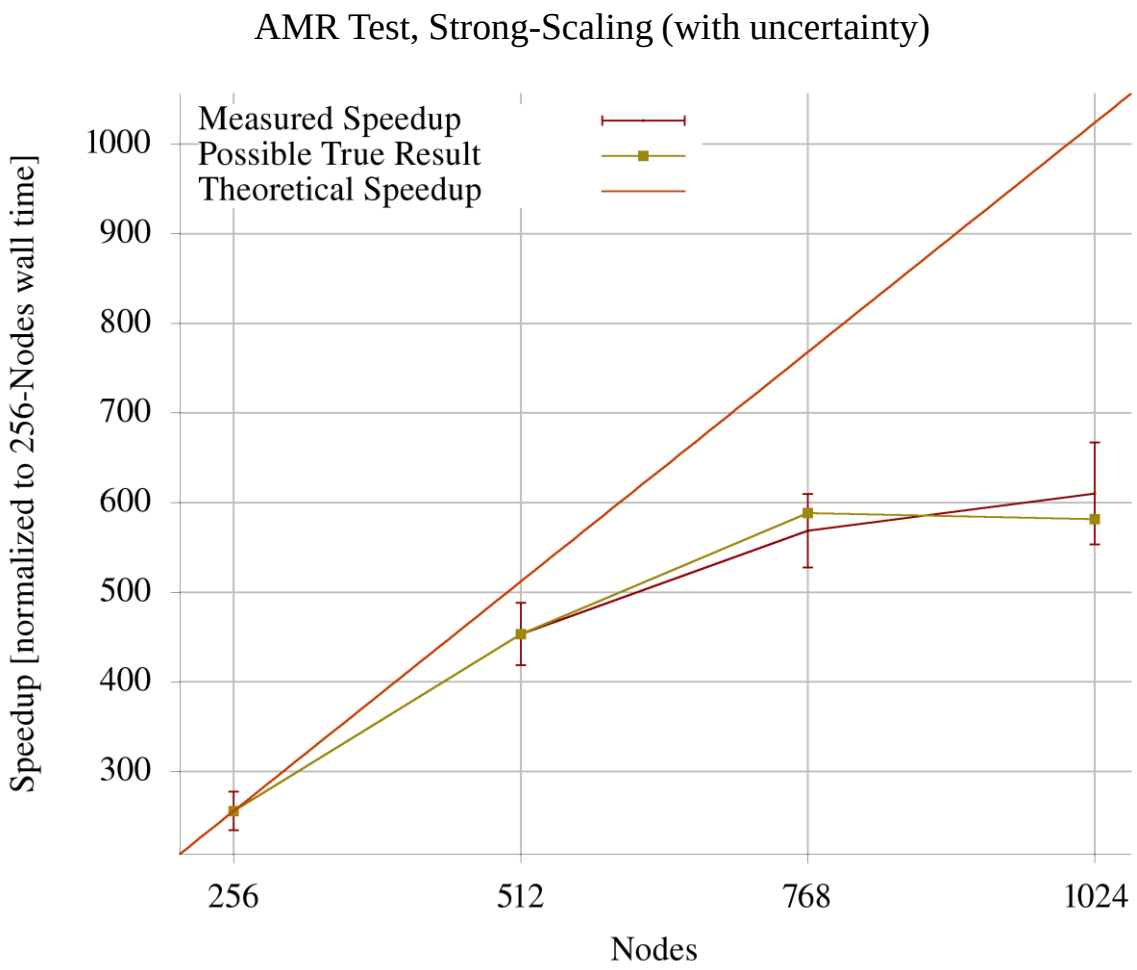
Case Study: CFD AMR Scaling



Copyright (C) 2015 Bryce Adelstein lelbach

BERKELEY LAB

Case Study: CFD AMR Scaling



Copyright (C) 2015 Bryce Adelstein lelbach

BERKELEY LAB

Statistical Best Practices

Process:

- Form a hypothesis: how do you expect performance to change?
- Come up with a test to determine if your hypothesis is right.
- Gather data.
- Statistically analyze data.
- Draw conclusions.

Copyright (C) 2015 Bryce Adelstein lelbach

Statistical Best Practices

Come up with a test to determine if your hypothesis is right.

- Identify independent/dependent/control variables.
- Determine what relevant metric you'll use (metric will be derived from dependent variables).
- Consider the assumptions you're making:
  - Assumptions about independence of variables.
  - Assumptions about distribution of samples.
    - Usually we assume a normal distribution.

Copyright (C) 2015 Bryce Adelstein lelbach

Gathering Data

Amortizing: When measuring “small” events, we often measure by amortization to reduce the observer effect.

- E.g. time an N-iteration for loop and divide by N to get the amortized time per iteration.

```
high_resolution_timer t; // Start timing.
for (std::size_t i = 0; i < N; ++i)
    A[i] = A[i] + B[i] * C[i];
double time_per_iteration = t.elapsed() / N;
```

- We treat this as one sample, not N samples.

Copyright (C) 2015 Bryce Adelstein lelbach

BERKELEY LAB

Gathering Data

Sampling: Each independent measurement we take is a sample.

- Samples are representative of the “population” (AKA the true performance).
- Our goal is to gather samples in sufficient quantity and quality to be representative of the population.

It’s crucial to both sample within one execution of the test and across multiple executions of the test.

- Gathering data across multiple executions gives a better representation of system noise.

Copyright (C) 2015 Bryce Adelstein lelbach

Gathering Data

Running “hot” vs “cold”.

- Often, you need to make sure that both your test as a whole (e.g. each execution), and the particular region your measuring (e.g. each sample) are not running “cold” on the CPU.
  - I/O, caching and branch prediction may be off if you’re running cold.

You can do this by doing some warmup executions/runs before you start measuring.

- E.g. don’t measure first execution or first few iterations.

Copyright (C) 2015 Bryce Adelstein lelbach

Uncertainty

Uncertainty: representation of the amount of error in a certain measurement.

- Instrument uncertainty: the inherent amount of uncertainty in an instrument.
  - Ex: if your clock ticks in microseconds, it would have an instrument uncertainty of +/- 500 nanoseconds (1/2th the unit of measurement).
- The sample standard deviation of a set of samples is a frequently used method for estimating the uncertainty of the average of the samples.

When dealing with derived metrics that use averaged data, you can formulate a derived uncertainty based on the uncertainties of the averaged data.

Copyright (C) 2015 Bryce Adelstein lelbach

Uncertainty

Given **uncorrelated** averaged data A and B with standard deviations σ_A and σ_B, and constants a and b.

Function	Standard Deviation
$f = aA$	$\sigma_f = a\sigma_A$
$f = aA \pm bB$	$\sigma_f = \sqrt{a^2\sigma_A^2 + b^2\sigma_B^2}$
$f = AB$ or $f = A/B$	$\sigma_f \approx \lvert f \rvert \sqrt{\left(\frac{\sigma_A}{A}\right)^2 + \left(\frac{\sigma_B}{B}\right)^2}$

Copyright (C) 2015 Bryce Adelstein lelbach

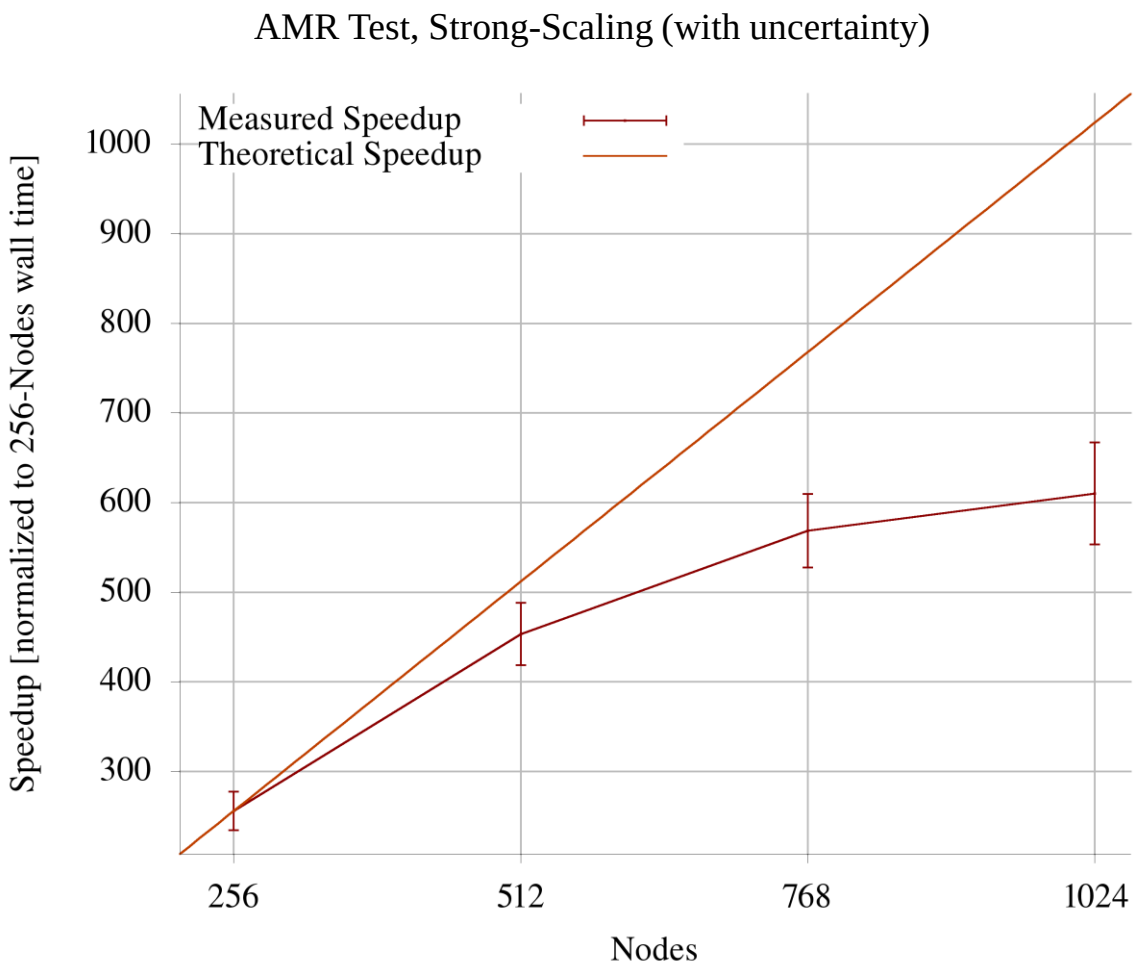
BERKELEY LAB

Case Study: CFD AMR Scaling

Statistical Best Practices

BERKELEY LAB

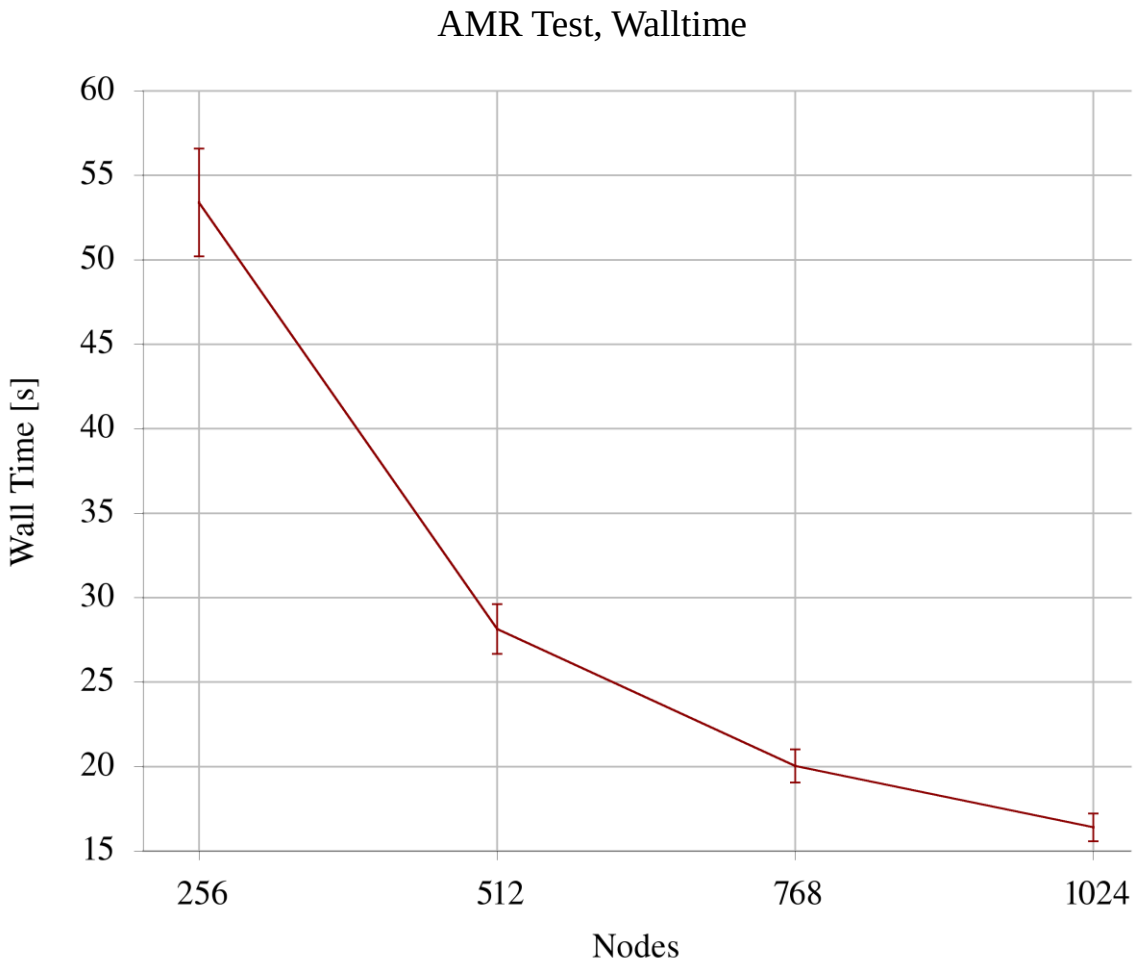
Case Study: CFD AMR Scaling



Copyright (C) 2015 Bryce Adelstein lelbach

BERKELEY LAB

Case Study: CFD AMR Scaling



Copyright (C) 2015 Bryce Adelstein lelbach

Example: Boost.Accumulators

Statistical Best Practices

BERKELEY LAB

Example: Boost.Accumulators

"Boost.Accumulators provides accumulators to which numbers can be added to get, for example, the mean or the **standard deviation**."

The Boost C++ Libraries, Boris Schäling

Copyright (C) 2015 Bryce Adelstein lelbach

Example: Boost.Accumulators

```
using namespace boost::accumulators;

int main()
{
    accumulator_set<
        double, stats<tag::count, tag::mean, tag::median, tag::variance>
    > acc;

    acc(42);

    // ... Accumulate data ...


    auto stdev = std::sqrt(variance(acc));

    std::cout << "Mean:   " << mean(acc)   << "\n"
              << "Median: " << median(acc) << "\n"
              << "Stdev:  " << stdev       << "\n";
}
```

Copyright (C) 2015 Bryce Adelstein lelbach

BERKELEY LAB

Example: Boost.Accumulators

On Jan 17, 2012, at 3:38 PM, Victor Yankee wrote:

> How can I calculate the Sample Standard Deviation over a std::vector of doubles using accumulators?
>
> Or is there a faster way in boost math or some such?

Google is your friend (second hit for "boost accumulator standard deviation"):
http://stackoverflow.com/questions/7616511/calculate-mean-and-standard-deviation-from-a-vector-of-samples-in-c-using-boos
and http://stackoverflow.com/questions/4316716/is-it-possible-to-use-boost-accumulators-with-vectors

```
accumulator_set<double, stats<tag::variance> > acc;
for_each(a_vec.begin(), a_vec.end(), bind<void>(ref(acc), _1));

cout << mean(acc) << endl;
cout << sqrt(variance(acc)) << endl;
```

-- Marshall

Marshall Clow Idio Software <mailto:mclow.lists_at_[hidden]>

A.D. 1517: Martin Luther nails his 95 Theses to the church door and is promptly moderated down to (-1, Flamebait).
-- Yu Suzuki

Copyright (C) 2015 Bryce Adelstein lelbach

Example: Boost.Accumulators

But this is the POPULATION standard deviation (variance divided by N) I think.

What I was asking for was how to calculated the SAMPLE standard deviation (variance devided by N-1).

Thanks,
Vic

Copyright (C) 2015 Bryce Adelstein lelbach

Example: Boost.Accumulators

Two different forms of standard deviation

- Uncorrected, takes the standard deviation of an entire population:

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2}$$

- Corrected, takes the standard deviation of a sample of a population:

$$\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \mu)^2}$$

Copyright (C) 2015 Bryce Adelstein lelbach

Example: Boost.Accumulators

```
using namespace boost::accumulators;

int main()
{
    accumulator_set<
        double, stats<tag::count, tag::mean, tag::median, tag::variance>
    > acc;

    acc(42);

    // ... Accumulate data ...

    auto n = count(acc);
    auto stdev = std::sqrt(variance(acc)*(n/(n-1.0)));

    std::cout << "Mean:   " << mean(acc)   << "\n"
              << "Median: " << median(acc) << "\n"
              << "Stdev:  " << stdev       << "\n";
}
```

Copyright (C) 2015 Bryce Adelstein lelbach

BERKELEY LAB

Gathering Data

Process for collecting good data:

- Take individual measurements in your code. Use amortization if relevant.
- Accumulate multiple measurements and uncertainty estimations in code.
- Gather results from multiple executions of the test, and recompute uncertainty estimations.
  - Given two averages, μ_1 and μ_2 (and a combined average μ), of n_1 and n_2 data points, with sample standard deviations σ_1 and σ_2, the combined sample standard deviation of both datasets is:

$$\sigma = \sqrt{\frac{n_1^2\sigma_1^2+n_2^2\sigma_2^2-n_2\sigma_1^2-n_2\sigma_2^2-n_1\sigma_1^2-n_1\sigma_2^2+n_y n_x\sigma_1^2+n_y n_x\sigma_2^2+n_1 n_2(\mu_1-\mu_2)^2}{(n_1+n_2-1)(n_1+n_2)}}$$

Copyright (C) 2015 Bryce Adelstein lelbach

BERKELEY LAB

Confidence Intervals

Confidence Interval: a way to describe the amount of uncertainty associated with a sample of a population.

- Constructed from three pieces of information:
  - Confidence level (r) - e.g. 90%, 95%, 99%.
  - Statistical data, including sample size (n).
  - Uncertainty for the data (σ).

$$CI = \frac{z\sigma}{\sqrt{n}}$$

- z is the critical value. For large sample sizes, you can look this up in a table. For small sample sizes, use the Student's *t* inverse cumulative distribution function:

$$\mathrm{z} = T_{inv}(1 - r, n - 1)$$

Copyright (C) 2015 Bryce Adelstein lelbach

BERKELEY LAB

Confidence Intervals

One of the useful things you can do with confidence intervals is determine the correct sample size, based on an initial "pilot" set of samples.

- Given a margin of error e_m, a critical value z, an uncertainty σ, and a mean μ:

$$n = \left(\frac{z\sigma}{\frac{e_m}{2}\mu}\right)^2$$

- If this calculation indicates an unreasonable large sample size is needed, the experiment may need to be redesigned.
- Typically, if your uncertainties are big relative to your data (mean and standard deviation have the same magnitude), there is too much noise to get meaningful results from your data.

Copyright (C) 2015 Bryce Adelstein lelbach

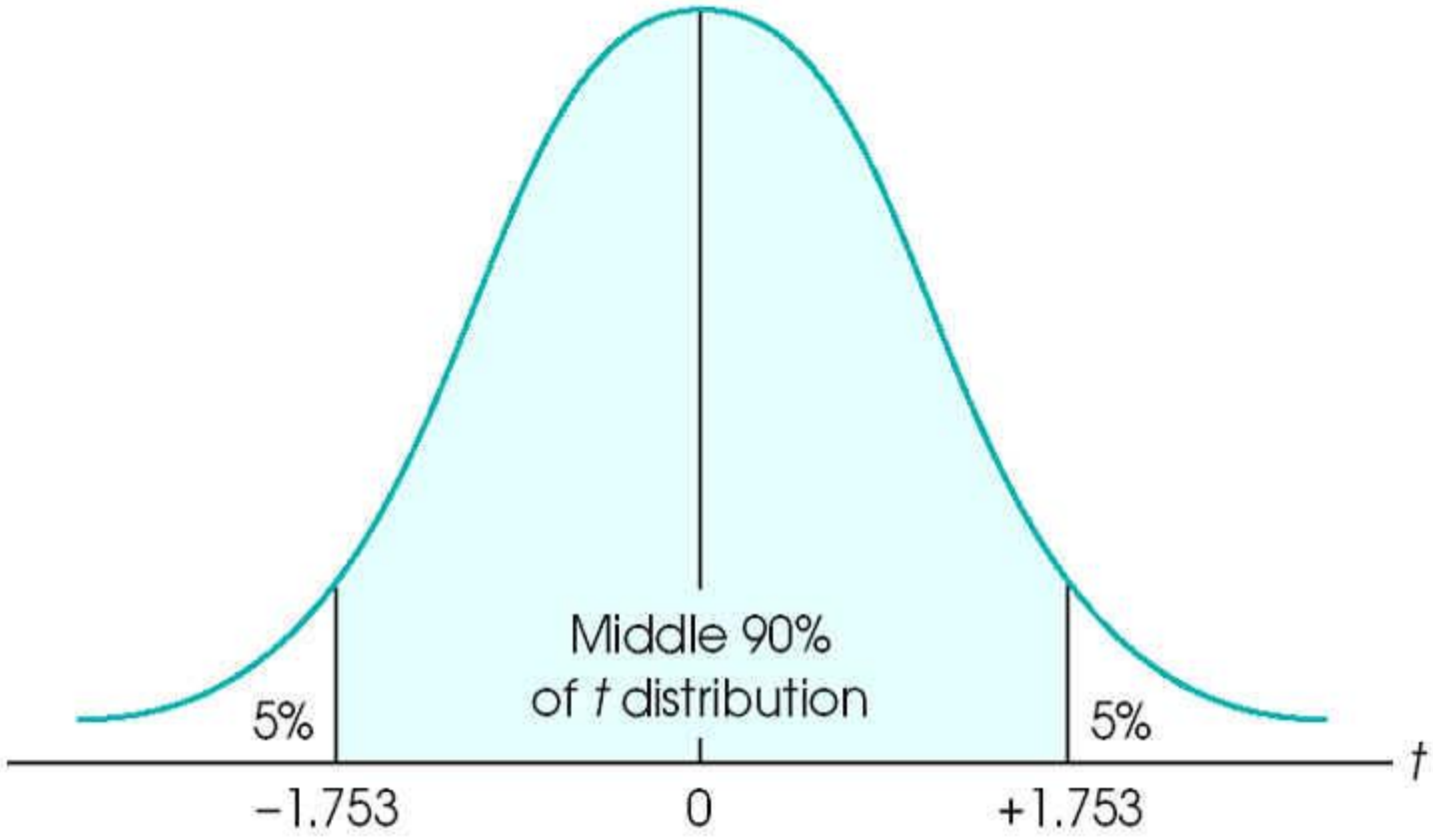
Confidence Intervals

Meaning of confidence intervals

- If the true performance lies outside of the 95% confidence interval, then an event occurred which had a probability of 5% or less of happening.
- A 95% confidence interval does **not** mean that 95% of the data lies within the interval.
- A confidence interval isn't a range of plausible values for a sample mean. It can be interpreted as an estimate of plausible values for the population.

Copyright (C) 2015 Bryce Adelstein lelbach

Confidence Intervals



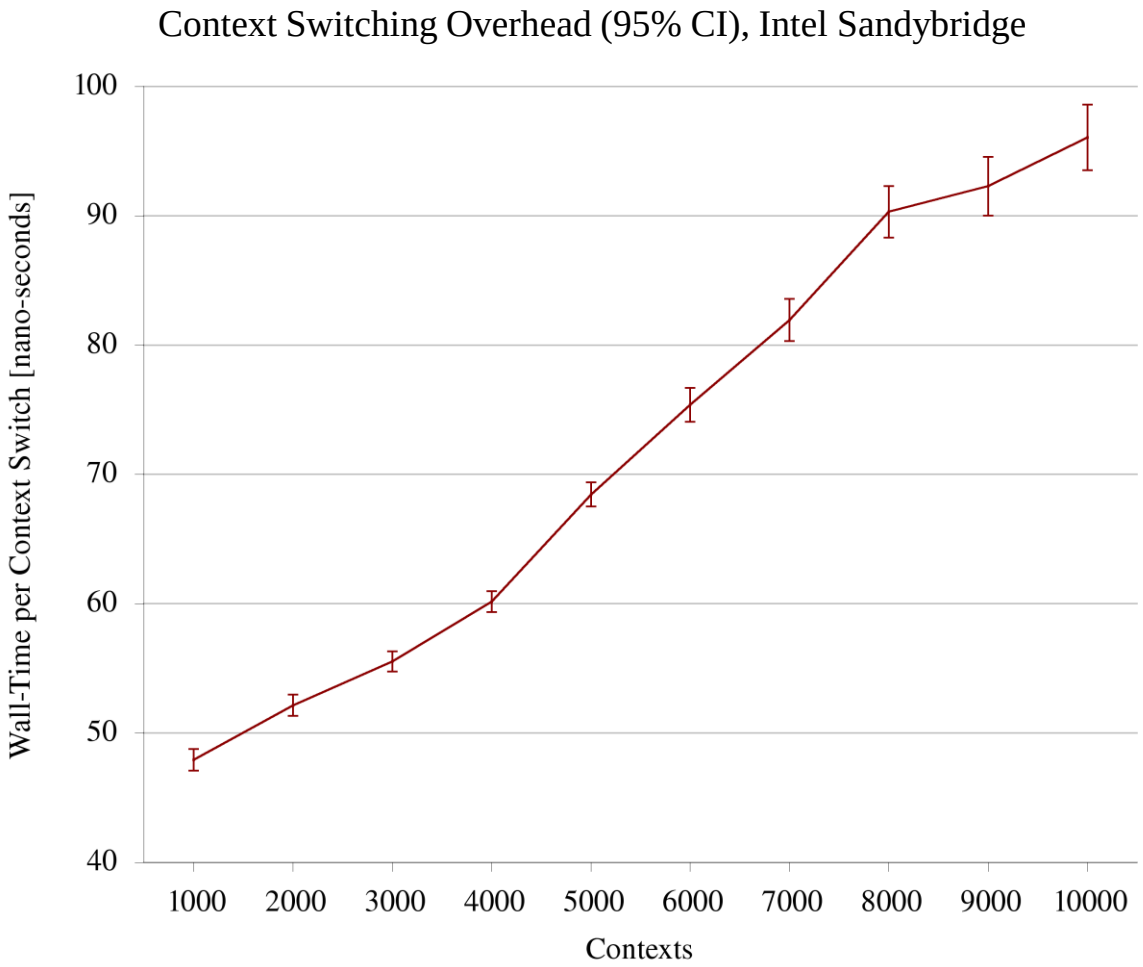
Copyright (C) 2015 Bryce Adelstein lelbach

Case Study: HPX CS Overhead

Statistical Best Practices

BERKELEY LAB

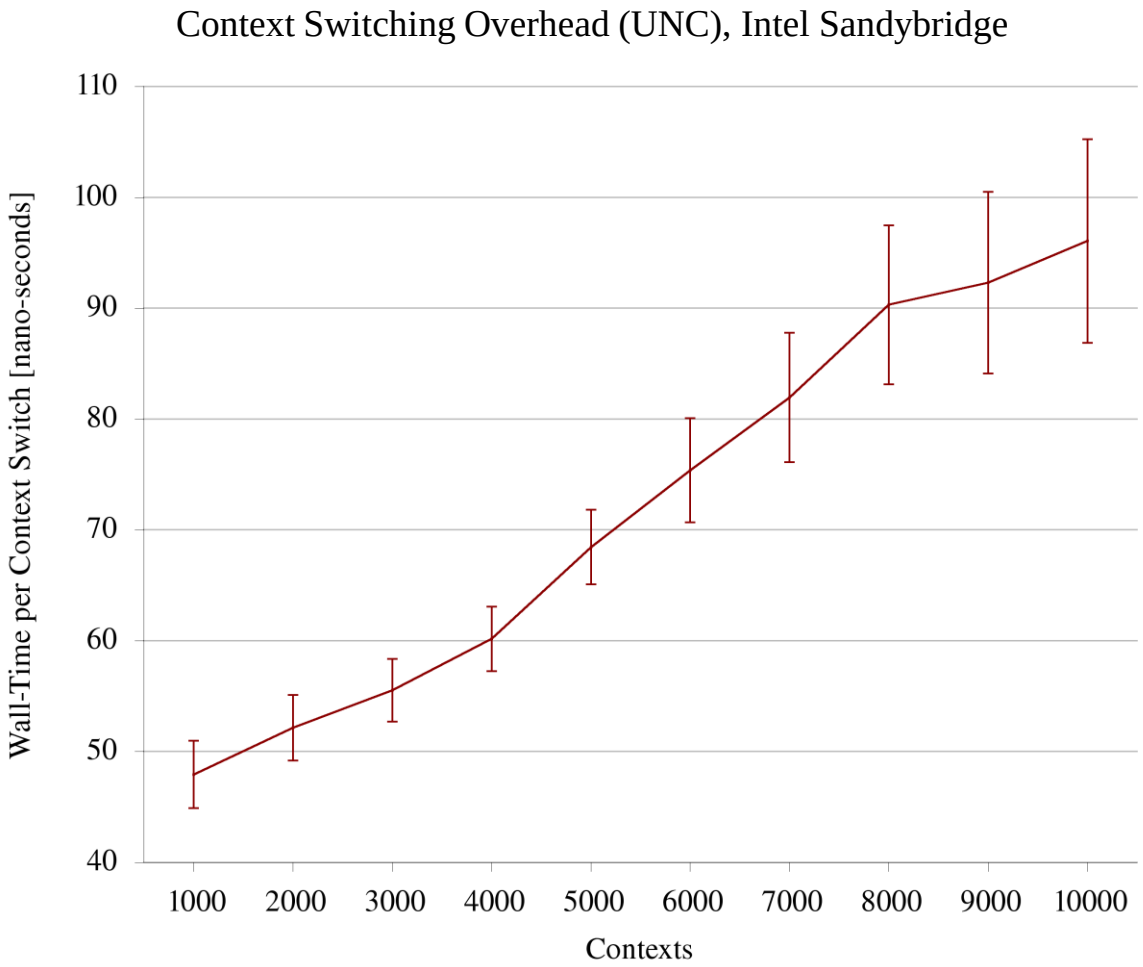
Case Study: HPX CS Overhead



Copyright (C) 2015 Bryce Adelstein lelbach

BERKELEY LAB

Case Study: HPX CS Overhead



Copyright (C) 2015 Bryce Adelstein lelbach

BERKELEY LAB

Mean-Median Test

Normality test: Tests to determine if a data-set fits a normal distribution well.

- There are graphical (QQ plot), informal/back-of-the-envelope and rigorous normality tests.

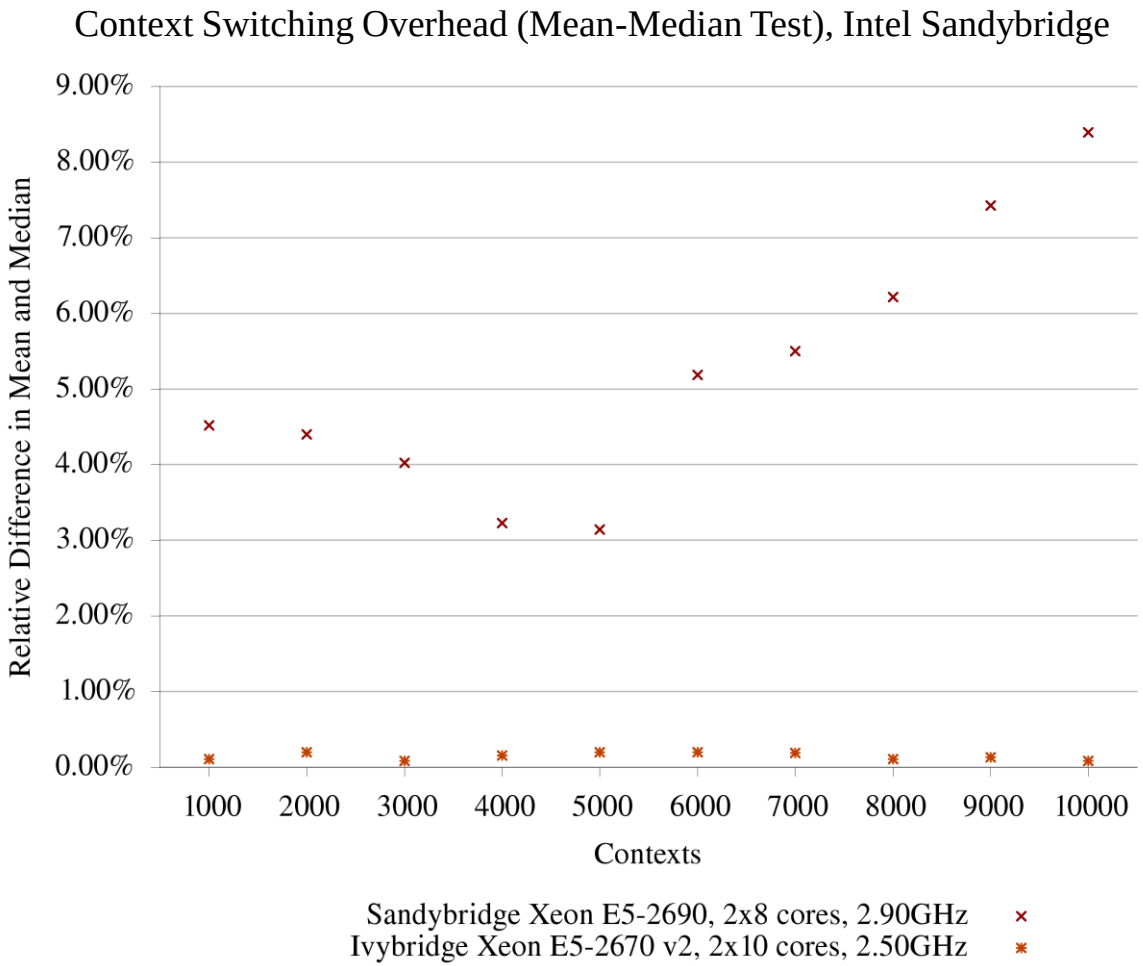
The mean (μ), median (m) and mode of normally distributed data should be the same, so…

$$\frac{|\mu - m|}{\max(\mu, m)}$$

- This will give you the relative difference between the mean and median (a percentage represented as a decimal). If this is larger than 1%, your data is probably not normally distributed.

Copyright (C) 2015 Bryce Adelstein lelbach

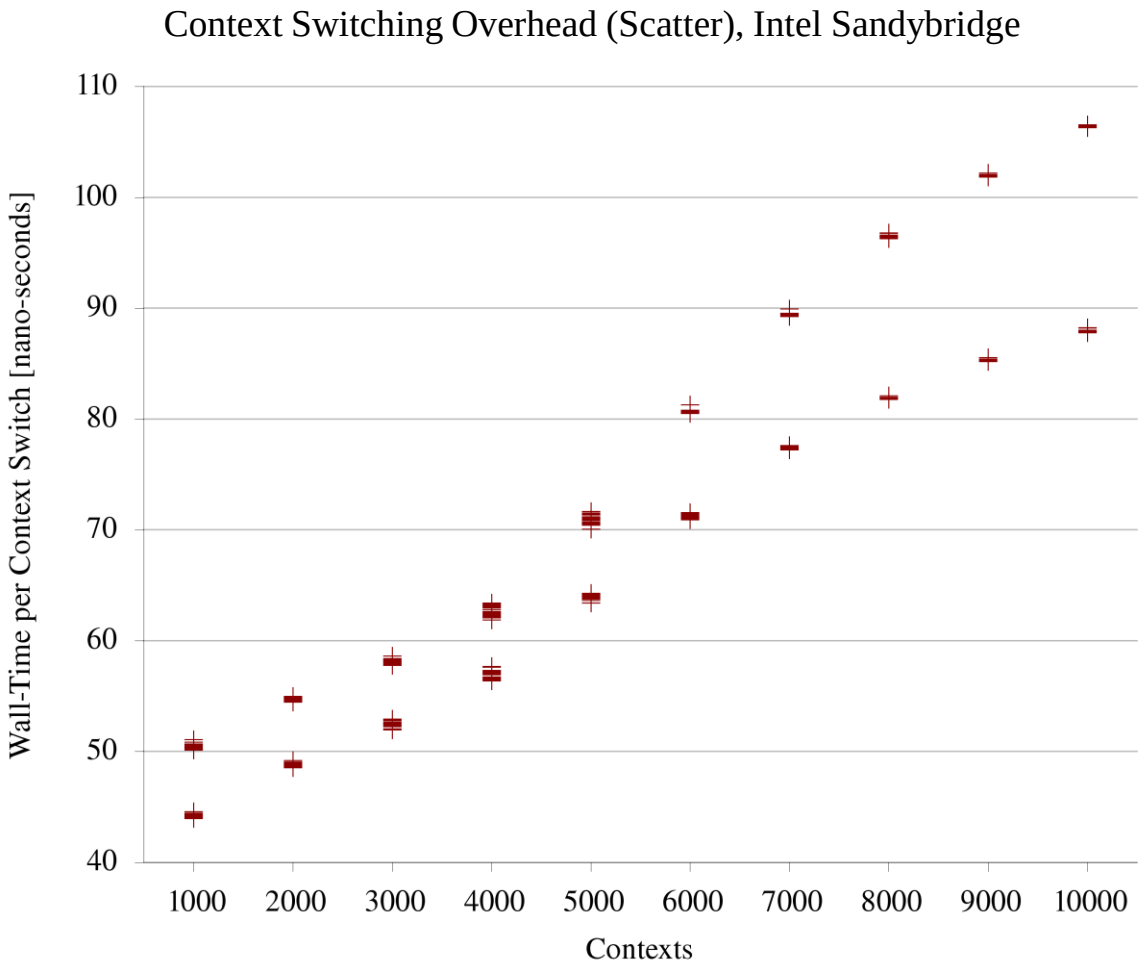
Case Study: HPX CS Overhead



Copyright (C) 2015 Bryce Adelstein lelbach

BERKELEY LAB

Case Study: HPX CS Overhead



Copyright (C) 2015 Bryce Adelstein lelbach

BERKELEY LAB

Time-Based Benchmarking

BERKELEY LAB

Time-Based Benchmarking

We have access to a few different clocksources for benchmarking on modern (x86) CPUS:

- System-wide high-resolution clock:
  - Monotonic, **frequency-stable**, higher latency and overhead.
  - Resolution is in nanoseconds.
  - Times can be passed between threads.
  - *nix, this is accessed via clock_gettime reading CLOCK_MONOTONIC.
  - Windows, this is accessed via QueryPerformanceCounter/Frequency.
  - Suitable for measuring most events (microseconds and up).

Copyright (C) 2015 Bryce Adelstein lelbach

Time-Based Benchmarking

We have access to three different clocksources for benchmarking on modern (x86) CPUS:

- Timestamp Counter (TSC):
  - Monotonic, lower latency and overhead.
  - Resolution is in CPU cycles (with caveats), tick is in base clock cycles.
    - All newer (4-5 year old) CPUs guarantee a constant TSC frequency, even if the CPU frequency changes (e.g. frequency scaling, Intel Turbo mode).
    - Constant TSC frequency == timing data is **not** representative of # of cycles executed.
    - Ticks with the base clock, which runs at 100 or 133 Mhz (depending on microarchitecture).
  - Assembly instruction(s) for reading this counter.
  - Cycle counts are thread-specific.
  - Suitable for measuring short events (cycles to minutes).

Copyright (C) 2015 Bryce Adelstein lelbach

<chrono>

Standard facilities for manipulating dates and times, introduced in C++11

- Three types:
  - Duration: A span of time, defined as some number of ticks of some time unit.
  - Time Point: A duration of time that has passed since the epoch of specific clock.
  - Clocks: An object with a starting point and a tick rate, which can be queried for the current time.

<chrono> is the best way to measure durations that are microsecond magnitude or large.

Source: cppreference.com

Copyright (C) 2015 Bryce Adelstein lelbach

BERKELEY LAB

<chrono>

Clock	Description
system_clock	Wall clock time from the system-wide realtime clock.
steady_clock	Monotonic clock that will never be adjusted.
high_resolution_clock	The clock with the shortest tick period available.

Copyright (C) 2015 Bryce Adelstein lelbach

BERKELEY LAB

Example: high_resolution_timer

Time-Based Benchmarking

BERKELEY LAB

Example: high_resolution_timer

```
struct high_resolution_timer
{
    high_resolution_timer() : start_time_(take_time_stamp()) {}

    void restart()
    { start_time_ = take_time_stamp(); }

    double elapsed() const // Return elapsed time in seconds.
    { return double(take_time_stamp() - start_time_) * 1e-9; }

    std::uint64_t elapsed_nanoseconds() const
    { return take_time_stamp() - start_time_; }

  protected:
    static std::uint64_t take_time_stamp()
    {
        return std::chrono::duration_cast<std::chrono::nanoseconds>
            (std::chrono::steady_clock::now().time_since_epoch()).count();
    }

  private:
    std::uint64_t start_time_;
};
```

Copyright (C) 2015 Bryce Adelstein lelbach

BERKELEY LAB

Non-Time-Based Benchmarking

BERKELEY LAB

Memory Benchmarking

Approaches to instrumenting memory allocation:

- What do we want to look at?
  - Objects (allocated/deallocated)
  - Memory (total, per object size, per object type)
- External tools:
  - googleperftools/TCMalloc (MALLOCSTATS)
  - MemTrack
- Overload operator new/delete
  - Writing a member operator new/delete is a great technique for tracking memory performance for a specific object.
  - I suggest a static member variable to store the performance data; if you need thread safety, use thread-local storage and accumulate afterwards.

Copyright (C) 2015 Bryce Adelstein lelbach

Example: Instrumenting operator new

Non-Time-Based Benchmarking

BERKELEY LAB

Example: Instrumenting operator new

```
struct A {
    static std::size_t allocated;

    static void* operator new(std::size_t sz)
    {
        allocated += sz/sizeof(A);
        return ::operator new(sz);
    }
    static void* operator new[](std::size_t sz)
    {
        allocated += sz/sizeof(A);
        return ::operator new(sz);
    }
};

std::size_t A::allocated = 0;
```

Copyright (C) 2015 Bryce Adelstein lelbach

BERKELEY LAB

Counting Copies/Moves

When we started transition the HPX codebase to support move semantics a few years ago, we wrote some tests to make sure we got it right.

- We passed mock objects that count copies/moves through our framework and looked at the results.
- Once we were confident our interfaces were doing things right (minimizing the number of copies, etc), we wrote unit tests to verify the move/copy counts wouldn't change.
- Especially important for us – HPX is an asynchronous programming framework, so there are places where we duplicate data to facilitate asynchrony.
  - We wanted to ensure we only copied async() arguments once.

Copyright (C) 2015 Bryce Adelstein lelbach

Hardware Performance Counters

X86 processors have a diverse set of hardware performance counters.

- Pros:
  - Low overhead.
  - Very diverse and descriptive information.
- Cons:
  - Microarchitecture specific.
  - Some counters are estimations, or suffer from inaccuracies (overcounting, etc).
  - You need very specialized knowledge to use these for performance analysis.
    - Fortunately, there's an awesome tool which has this knowledge baked into it.

Copyright (C) 2015 Bryce Adelstein lelbach

Hardware Performance Counters

Low-level frameworks for accessing hardware counters from within your code:

- Linux: PAPI framework
- Windows: Performance Counter framework
- Mac: kpc.h

There are some external sampling-based profiling tools that provide access to this information.

- Ex: Intel VTune Amplifier.

Copyright (C) 2015 Bryce Adelstein lelbach

Performance Analysis Tools

BERKELEY LAB

Intel VTune Amplifier

Sampling-based profiling tool: runs your application, and collects “snapshots” of performance metrics while your program is running.

- Works on Intel processors, Windows/Linux/Mac OS X/Android, not tied to any particular compiler.
- Requires no code changes to use.
- Multiple data sources: timers, hardware performance counters and operating system metrics.
- Performance data can be viewed per function or at assembly/source code granularity.
- Analyzes everything: kernel calls, sub-processes, threads, etc.

Copyright (C) 2015 Bryce Adelstein lelbach

Intel VTune Amplifier

Sampling-based profiling tool: runs your application, and collects “snapshots” of performance metrics while your program is running.

- Provides built-in analysis passes which derive useful, higher-level performance metrics from micro-architecture specific raw hardware counters.
- Also supports user-defined analysis passes.
- Support for instrumenting parallel and distributed code.
  - Built-in support for OS-threading frameworks.
  - Built-in support for OpenMP, MPI and Intel TBB.
  - Provides an instrumentation API which parallel programming frameworks can use to inform the profiler about their threading and concurrency data structures.

Copyright (C) 2015 Bryce Adelstein lelbach

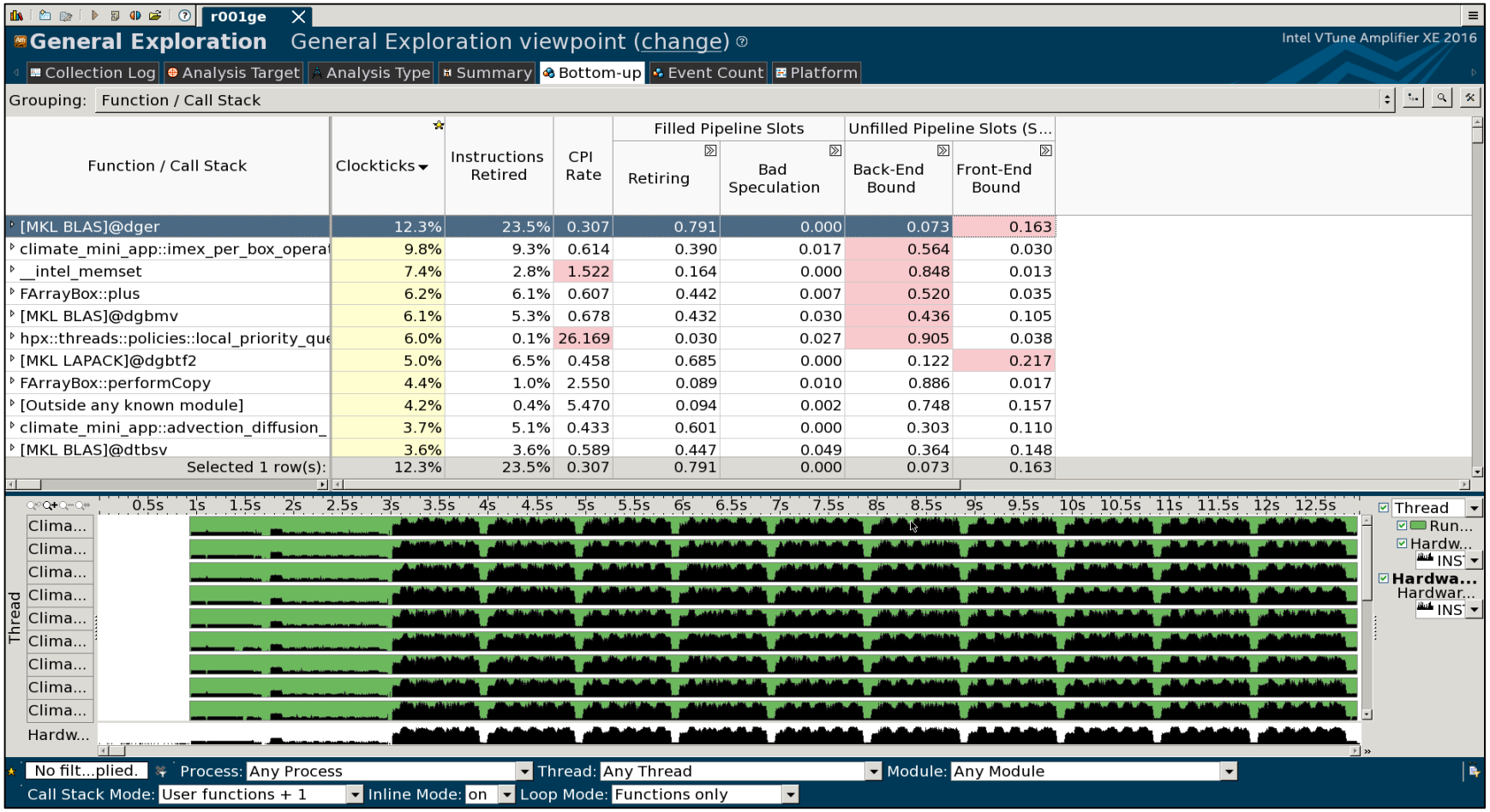
Intel VTune Amplifier

Sampling-based profiling tool: runs your application, and collects “snapshots” of performance metrics while your program is running.

- Powerful GUI.
  - Standalone Windows/Linux/Mac GUI as well as integration with Visual Studio and Eclipse.
  - Data can be collected remotely via the command line interface and then fed into the GUI.
  - Great interface for filtering data (e.g. focusing in on just one section of the program’s execution).
  - Built-in analysis passes contain a lot of information about how to interpret results.

Copyright (C) 2015 Bryce Adelstein lelbach

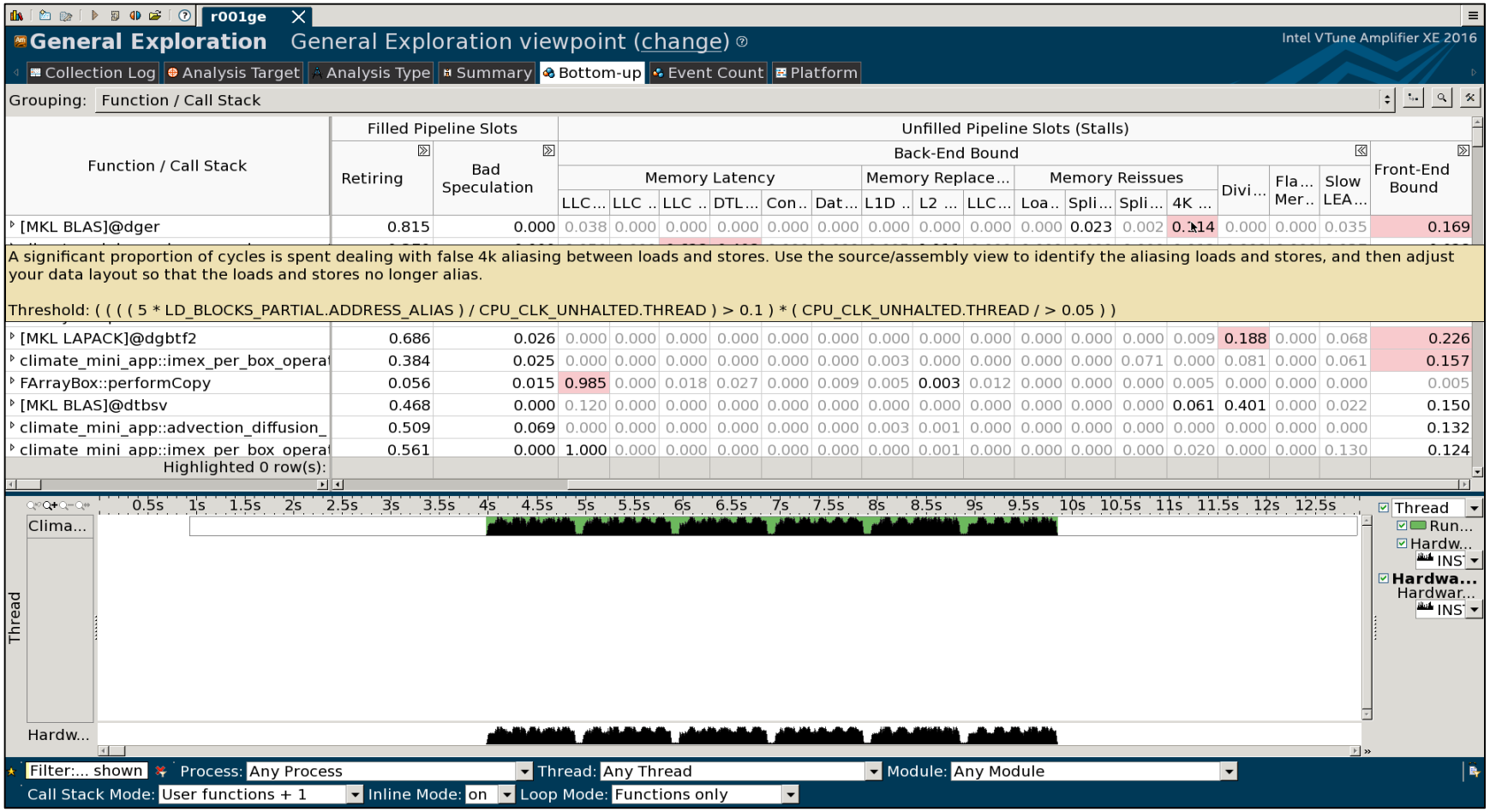
Intel VTune Amplifier



Copyright (C) 2015 Bryce Adelstein lelbach

BERKELEY LAB

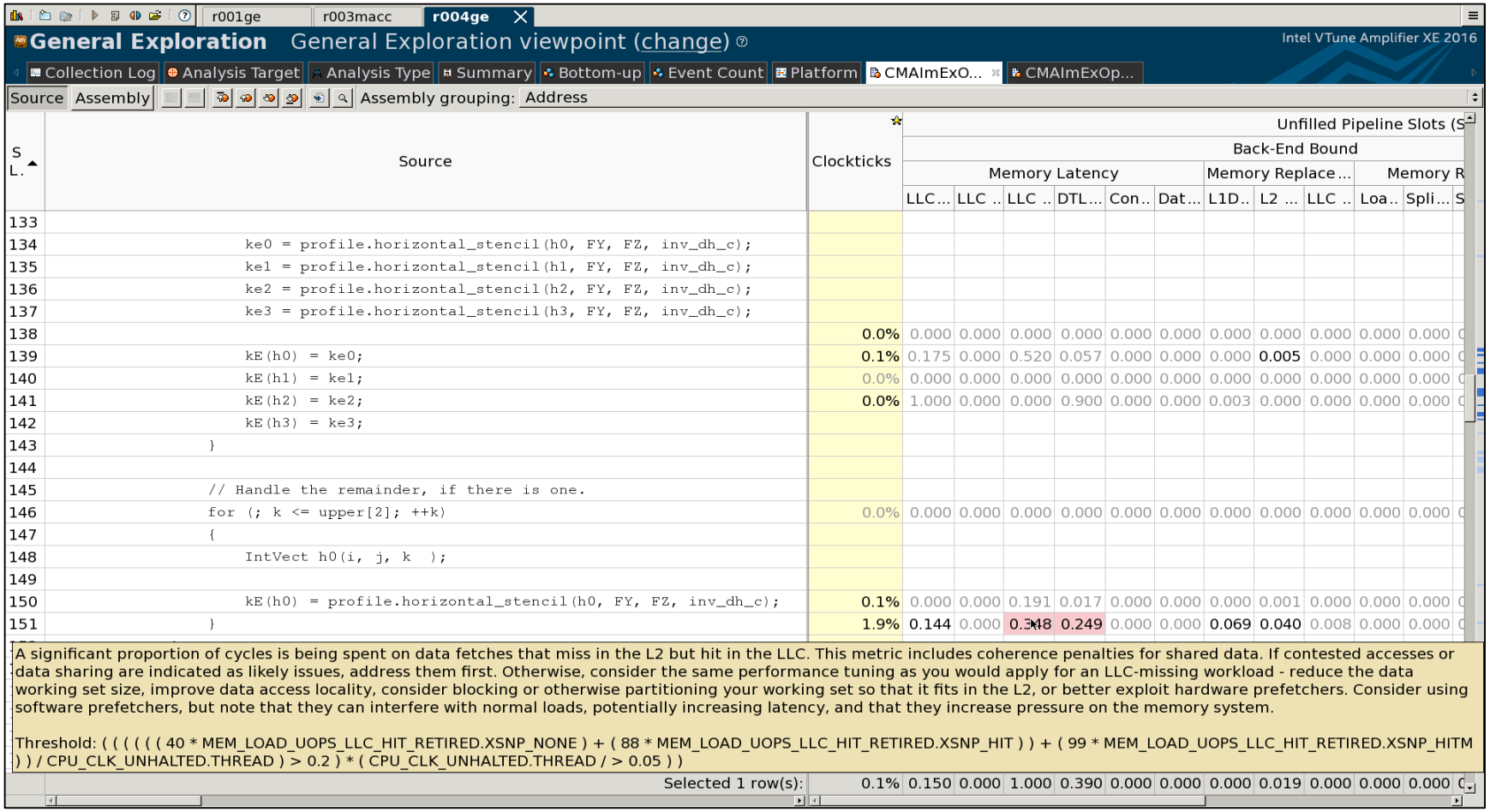
Intel VTune Amplifier



Copyright (C) 2015 Bryce Adelstein lelbach

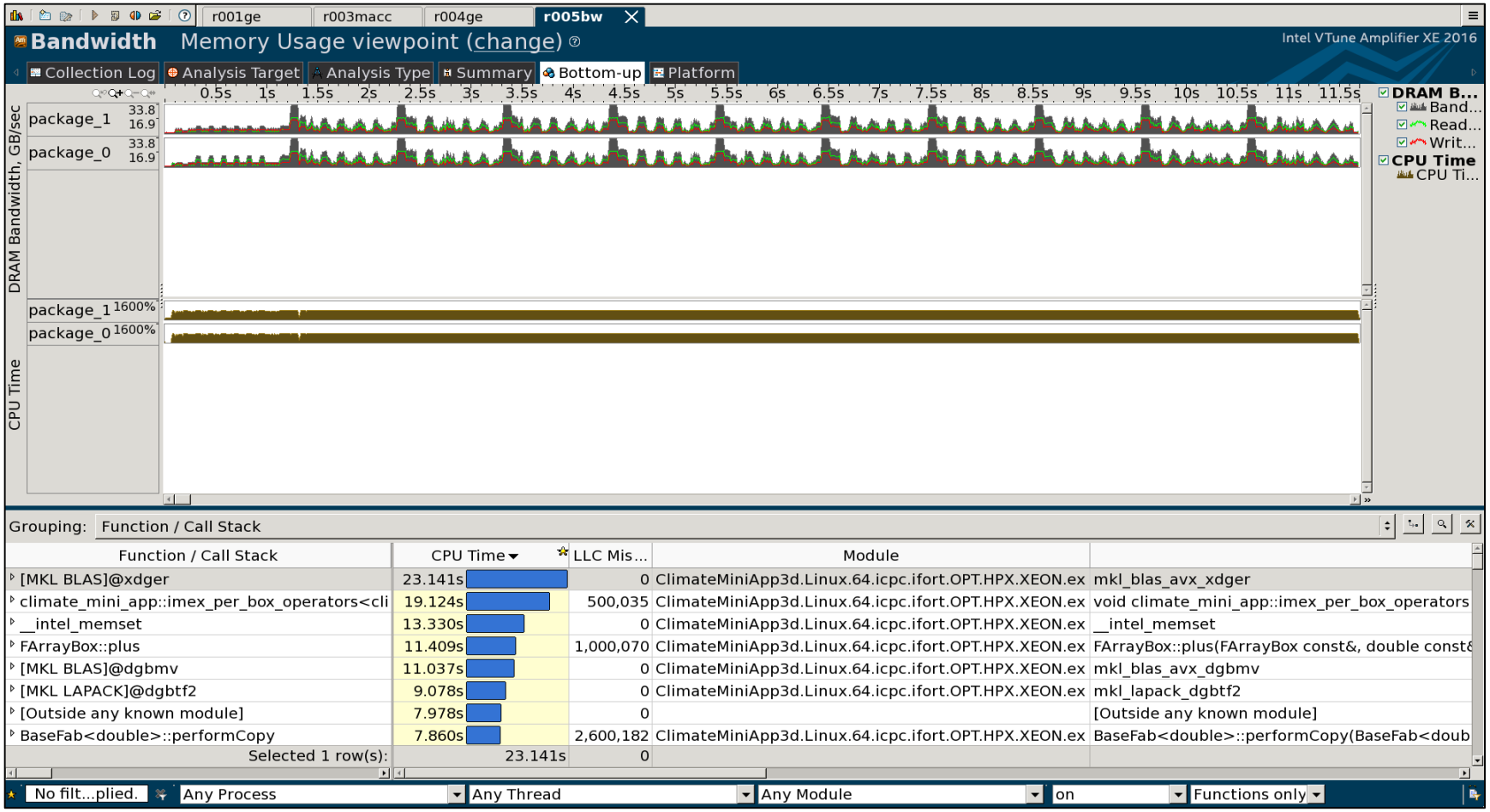
BERKELEY LAB

Intel VTune Amplifier



Copyright (C) 2015 Bryce Adelstein lelbach

Intel VTune Amplifier



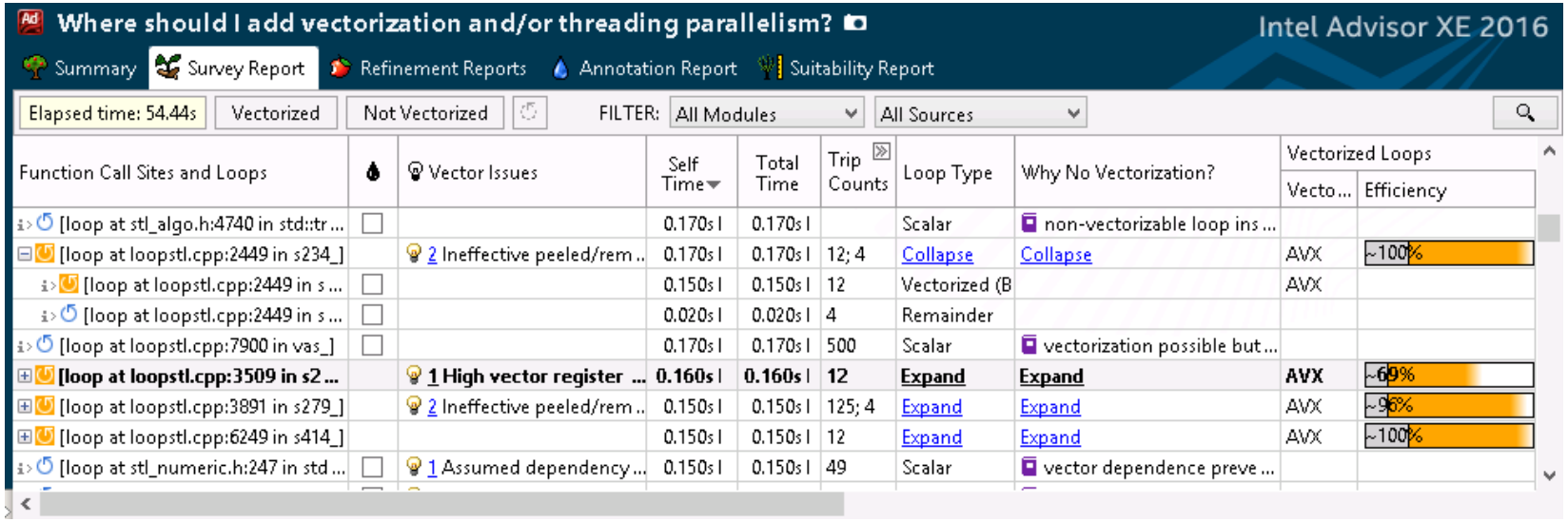
Copyright (C) 2015 Bryce Adelstein lelbach

BERKELEY LAB

Intel Vectorization Adviser

New tool in Intel Parallel Studio XE 2016: Vectorization Adviser.

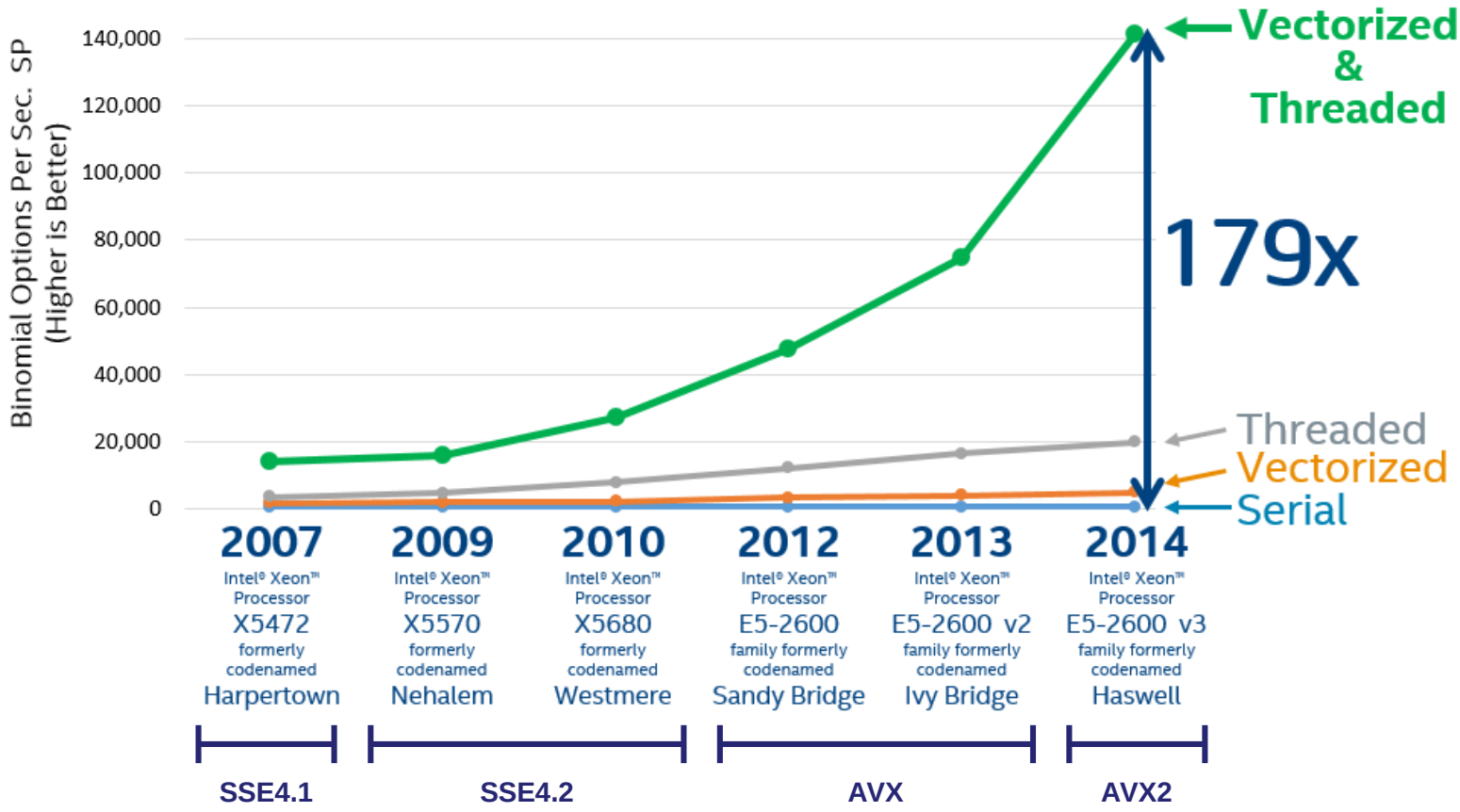
- Integrates the Intel compiler's vectorization reports into the GUI performance profiling framework.



Source: https://software.intel.com/en-us/intel-advisor-xe/

Copyright (C) 2015 Bryce Adelstein lelbach

Intel Vectorization Adviser



Source: https://software.intel.com/en-us/intel-advisor-xe/

Copyright (C) 2015 Bryce Adelstein lelbach

BERKELEY LAB

Write Performance Tests

BERKELEY LAB

Write Performance Tests

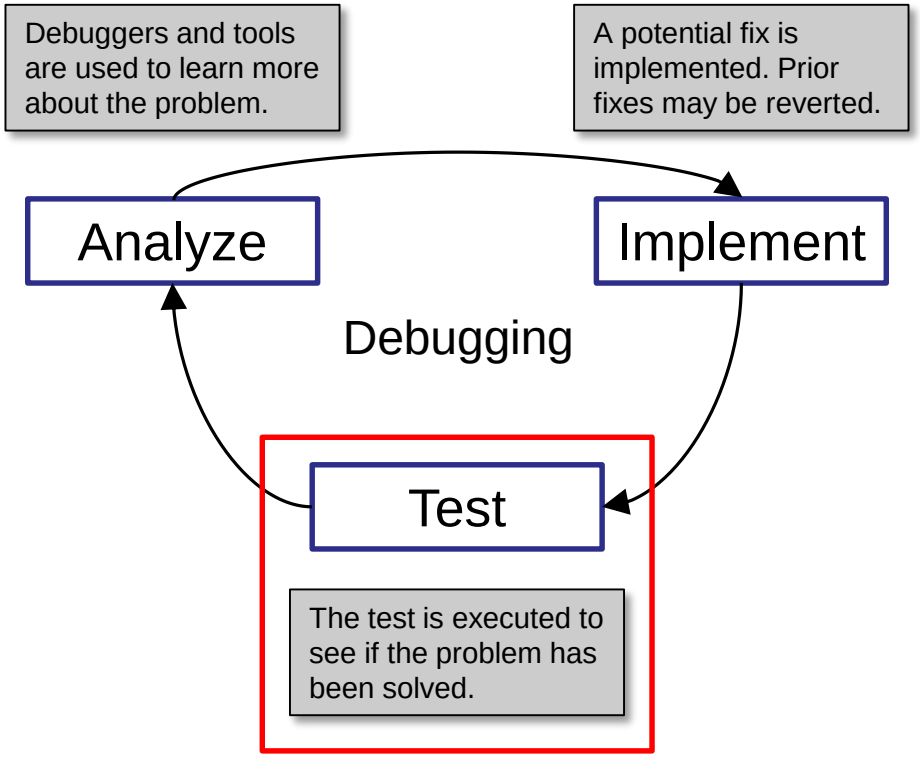
Idea: Let's write unit and regression tests for performance, just like we do for correctness.

Challenges:

- Implementing automated performance testing that follow the kind of best practices we've been talking about requires a lot of machinery.
- If your performance tests are stateful (rely on the results of previous automated tests), you need even more machinery.
- You need more than just the machinery to run the tests - you also need automated analysis to determine whether the test has failed.

Copyright (C) 2015 Bryce Adelstein lelbach

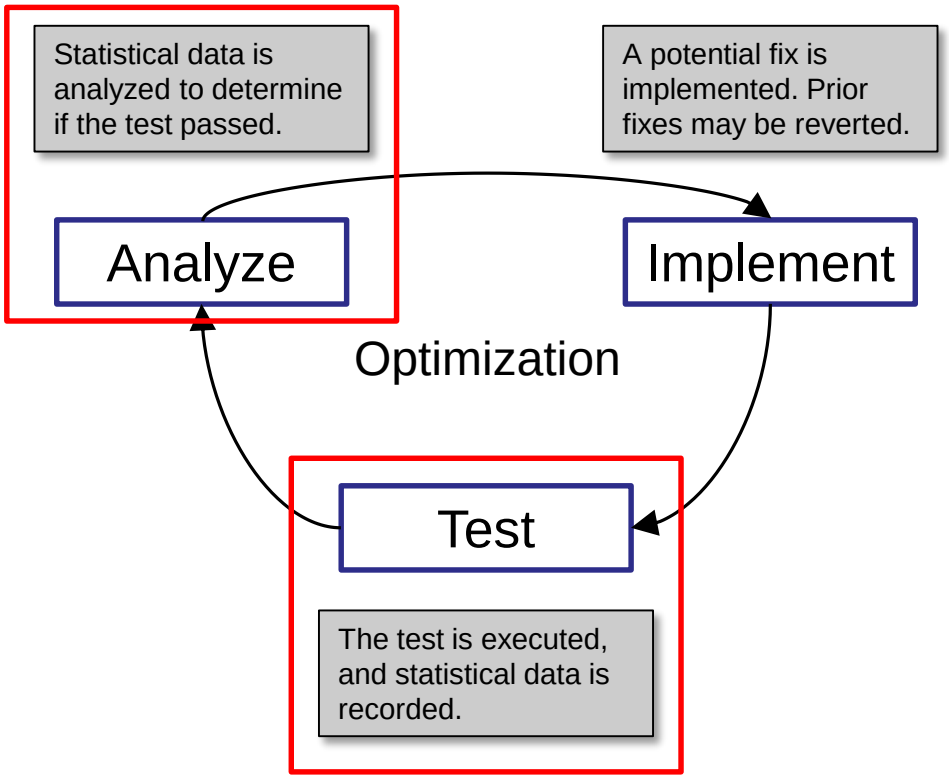
Write Performance Tests



Copyright (C) 2015 Bryce Adelstein lelbach

BERKELEY LAB

Write Performance Tests



Copyright (C) 2015 Bryce Adelstein lelbach

BERKELEY LAB

Stateful Performance Tests

Stateful performance tests: performance benchmarks which yield results that cannot be interpreted without contextual information.

- Output: absolute values.
- Most of your existing benchmarks are already stateful.
- To automate these tests, the current performance (e.g. trunk) needs to be compared against some prior results. There's two options for doing this:
  - Automated build system stores prior results for comparison. Requires more machinery, but allows you to track performance over time.
  - Automated build system checks out and builds an older version of the code to compare against.

Copyright (C) 2015 Bryce Adelstein lelbach

Stateful Performance Tests

Copyright (C) 2015 Bryce Adelstein lelbach

BERKELEY LAB

Stateless Performance Tests

Stateless performance tests: performance benchmarks which test for a performance "failure", and can provide a **"yes/no"** answer without external data.

- Output: relative values.

The idea is to compare different implementation options which you believe to have a performance impact.

- Ex: Lockfree queue vs lock-based queue.
- Ex: Recomputing data locally vs overhead for sharing.
- Ex: Algorithmic complexity testing.

Copyright (C) 2015 Bryce Adelstein lelbach

Stateless Performance Tests

Whenever you face a design trade-off

with performance implications,

write a stateless test!

Copyright (C) 2015 Bryce Adelstein lelbach

Summary

- Take a scientific approach to performance benchmarking.
  - Hypothesize, design a test, run the test, analyze, draw conclusions.
- Manage and test your assumptions about your tests.
- Collect a statistically significant quantity of data.
- Measure and propagate error.
- Develop unit and regression tests for performance.

Copyright (C) 2015 Bryce Adelstein lelbach

U.S. DEPARTMENT OF
ENERGY

UNIVERSITY OF
CALIFORNIA