





UNIVERSITY OF CALIFORNIA





Benchmarking C++ Code

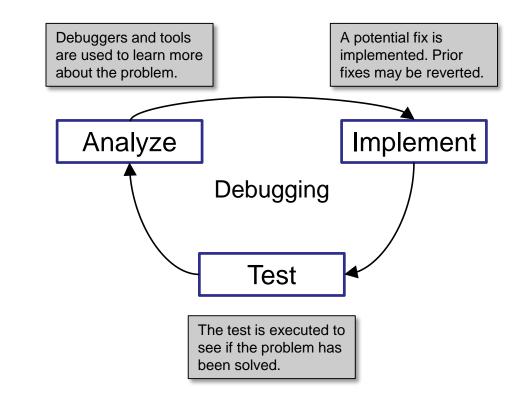
Bryce Adelstein Lelbach aka wash <balelbach@lbl.gov>
Computer Architecture Group, Computing Research Division

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).

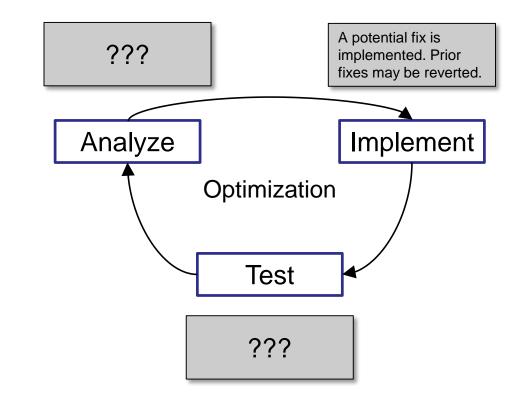




The Problem with Performance

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.





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.



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



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.



Variance

Computers can reproduce answers, not performance.

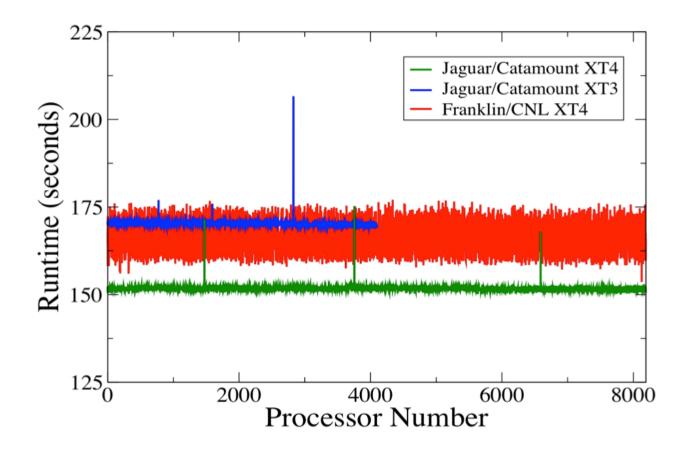
- Larger memory segments may have variance in access times due to physical distance from the CPU.
- Additionally, OS activities can cause non-determinism.
 - Some hardware interrupts require OS handling immediately after delivery.
 - Migration of non-pinned processes can affect the performance of CPU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

Observer Effect: all forms of instrumentation change the results.

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



Variance



Source: Cy Chan, John Bachan





Statistics: A great way to lie to yourself.



Statistics: A way to extract conclusions from your data



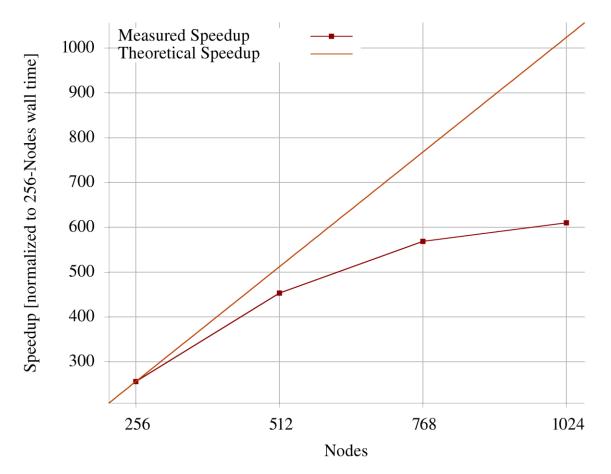
Statistics: The science of data... collection analysis interpretation presentation



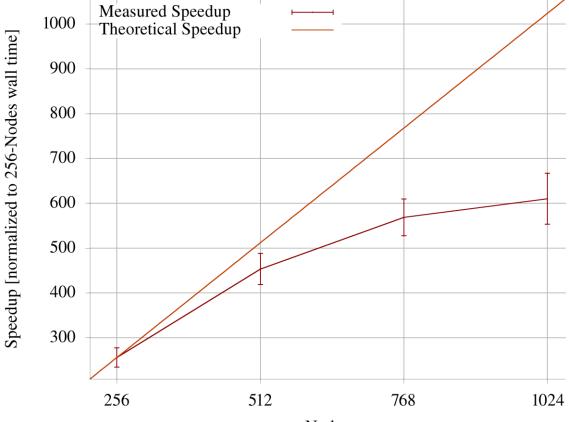
Statistical Best Practices



AMR Test, Strong-Scaling



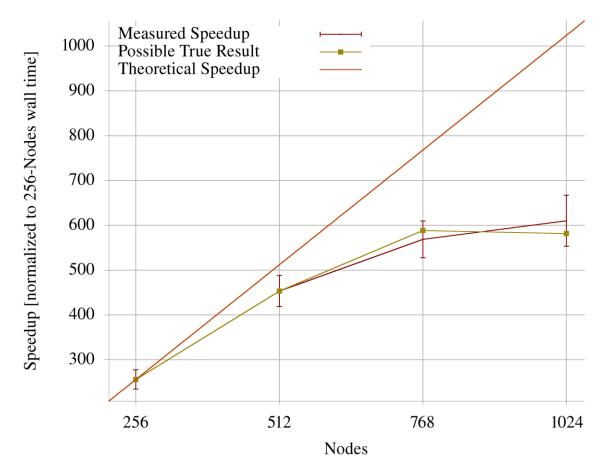




AMR Test, Strong-Scaling (with uncertainty)

Nodes





AMR Test, Strong-Scaling (with uncertainty)



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.



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.



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;</pre>

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



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.



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.



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.



Uncertainty

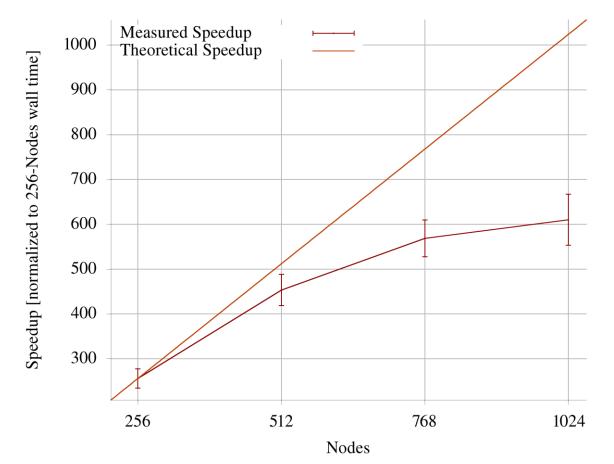
Given **uncorrelated** averaged data *A* and *B* with standard deviations σ_A and σ_B , and constants *a* and *b*.

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 $f = A/B$	$\sigma_{f} \approx f \sqrt{\left(\frac{\sigma_{A}}{A}\right)^{2} + \left(\frac{\sigma_{B}}{B}\right)^{2}}$



Statistical Best Practices

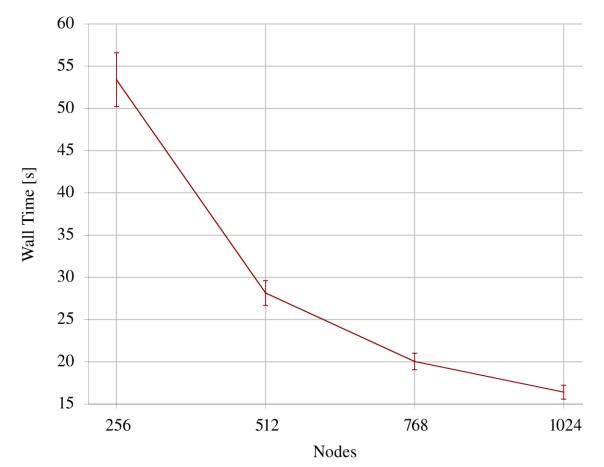




AMR Test, Strong-Scaling (with uncertainty)



AMR Test, Walltime





Statistical Best Practices



"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



```
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)</pre>
                                             << "\n"
              << "Median: " << median(acc) << "\n"</pre>

    "Stdev: " << stdev << "\n";
</pre>
}
```



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;</pre>

-- 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



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



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:

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



```
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)</pre>
                                             << "\n"
              << "Median: " << median(acc) << "\n"</pre>

    "Stdev: " << stdev << "\n";
</pre>
}
```



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:

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



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:

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



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.



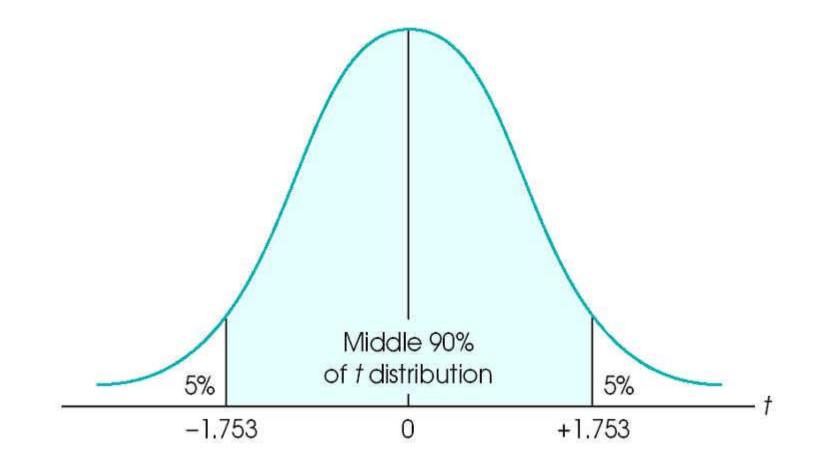
Confidence Intervals

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.



Confidence Intervals

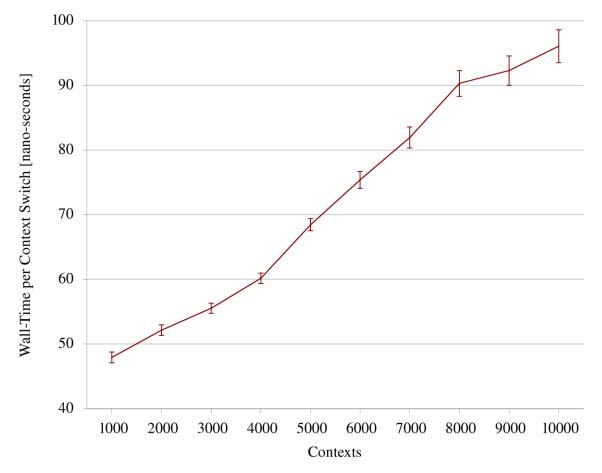




Statistical Best Practices

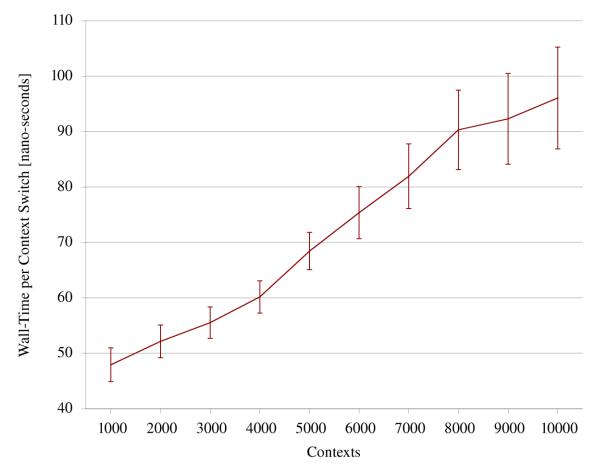


Context Switching Overhead (95% CI), Intel Sandybridge





Context Switching Overhead (UNC), Intel Sandybridge





Mean-Median Test

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

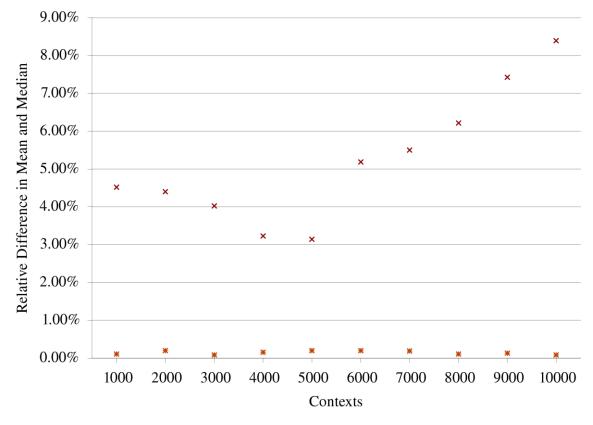
• There are graphical (QQ plot), informal/back-of-the-envelope and rigorous normality tests.

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.



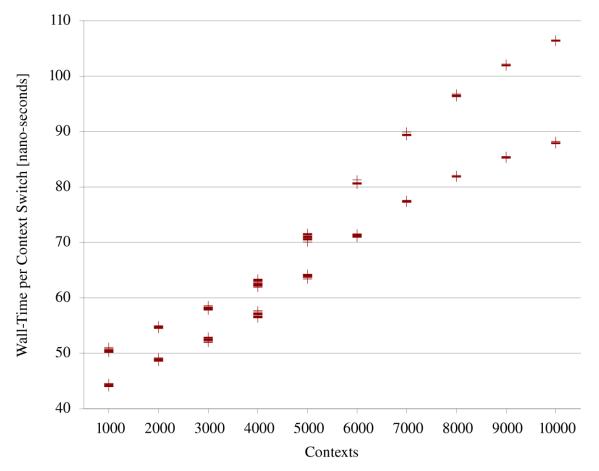


Context Switching Overhead (Mean-Median Test), Intel Sandybridge

Sandybridge Xeon E5-2690, 2x8 cores, 2.90GHz × Ivybridge Xeon E5-2670 v2, 2x10 cores, 2.50GHz *



Context Switching Overhead (Scatter), Intel Sandybridge





Time-Based Benchmarking



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).



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).





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

<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.



Example: high_resolution_timer

Time-Based Benchmarking



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()
    Ł
        return std::chrono::duration cast<std::chrono::nanoseconds>
            (std::chrono::steady clock::now().time since epoch()).count();
    }
 private:
    std::uint64 t start time ;
};
```



Non-Time-Based Benchmarking



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.



Example: Instrumenting operator new

Non-Time-Based Benchmarking



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;
```



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.



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.



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.



Performance Analysis Tools



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.



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, higherlevel 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.



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.



General Exploration Gen	ieral Explo	ration vie	wpoin	it (<u>change</u>) @					Intel VTune	Amplifier XE 201
🛛 📟 Collection Log 🛛 🖶 Analysis Target 🛛 🗛	Analysis Type	🛚 Summary 🍕	Bottom	-up 💊 Event 🤇	Count 🗷 Platfor	m					
Grouping: Function / Call Stack											¢ 5. 9
	*			Filled Pig	oeline Slots	Unfilled Pipel	ne Slots (S				
		Instructions	СРІ	\mathbb{D}	\gg		2				
Function / Call Stack	Clockticks v	Retired	Rate	Retiring	Bad Speculation	Back-End Bound	Front-End Bound				
[MKL BLAS]@dger	12.3%	23.5%	0.307	0.791	0.000	0.073	0.163				
climate_mini_app::imex_per_box_operat	9.8%	9.3%	0.614	0.390	0.017	0.564	0.030				
intel_memset	7.4%	2.8%	1.522	0.164	0.000	0.848	0.013				
FArrayBox::plus	6.2%			0.442	0.007	0.520	0.035				
[MKL BLAS]@dgbmv	6.1%			0.432	0.030	0.436	0.105				
hpx::threads::policies::local_priority_que			26.169	0.030	0.027	0.905	0.038				
[MKL LAPACK]@dgbtf2	5.0%			0.685	0.000	0.122	0.217				
FArrayBox::performCopy	4.4%			0.089	0.010	0.886	0.017				
[Outside any known module]	4.2%			0.094	0.002	0.748	0.157				
climate_mini_app::advection_diffusion_	3.7%			0.601	0.000	0.303	0.110				
[MKL BLAS]@dtbsv Selected 1 row(s):	3.6% 12.3%		0.589 0.307	0.447	0.049	0.364 0.073	0.148 0.163				
	12.5%	23.5%	0.307	0.791	0.000	0.073	0.105				
್ಷಾ-್-್ 0.5s 1s 1.5s 2s 2	2.5s 3s 3.5	is 4s 4.5s	5s	5.5s 6s 6	5.5s 7s 7.5s	8s 8.5s	9s 9.5s	10s 10.5s 11	s 11.5s 1	2s 12.5s	🛛 Thread
Clima	u mahanan munah	and a state of the						and the state of the second			🗧 🗹 🔲 Rur
Clima											Hardv
Clima		allinette, pålande, bisatt									Hardwa
Clima								and the second second			Hardwa
Clima										a dia 1 dia 12 dia 44 dia 4 dia 4 dia 4	IN IN
Clima	and the second							and the state of the			1
Clima					And the second sec			and the second s			
Clima	anter anter a succession and the second s	and a state of the				Charles and a second				alle alle she she she	
Clima	AND A STREET, S										-
Hardw			hite and a		***				la de la composition	A444	i
No filtplied. 💝 Process: Any Proces	5		read A	ny Thread		▼ Module:	Any Module		-	<u> </u>	»



▲ ি ▷ ▷ ▷ □ ● ☞ ○ rooige X General Exploration Ger	oral Expl	aration via	upair	at (al	hang												Intel 1		≡ mplifier XE 2016
•		oration view	•				+ B DI	atform									inter	v fulle A	
Grouping: Function / Call Stack	Analysis Type	■ Summary 😡	Botton	n-up	• Even	t Coun		actorm											÷ =
Filled Pipeline Slots									Unfilled Pipeline Slots (Stalls)										
		>		Back-End Bound															
Function / Call Stack	Retiring	Bad Speculation		Memory Latency			Memo	ory Rep	lace	М	emory	Reissu	les	Fla	Fla	Slow	Front-End Bound		
		speculation	LLC	LLC	LLC	DTL	Con	Dat	L1D	L2	LLC	Loa	Spli	Spli	4K	Divi		LEA	Bound
[▶] [MKL BLAS]@dger	0.815	0.000	0.038	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.023	0.002	0.114	0.000	0.000	0.035	0.169
A significant proportion of cycles is spent dealing with false 4k aliasing between loads and stores. Use the source/assembly view to identify the aliasing loads and stores, and then adjust your data layout so that the loads and stores no longer alias. Threshold: ((((5 * LD_BLOCKS_PARTIAL.ADDRESS_ALIAS) / CPU_CLK_UNHALTED.THREAD) > 0.1) * (CPU_CLK_UNHALTED.THREAD / > 0.05))																			
[MKL LAPACK]@dgbtf2	0.686	0.026	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.009	0.188	0.000	0.068	0.226
climate_mini_app::imex_per_box_opera	0.384	0.025	0.000	0.000	0.000	0.000	0.000	0.000	0.003	0.000	0.000	0.000	0.000	0.071	0.000	0.081	0.000	0.061	0.157
FArrayBox::performCopy	0.056							0.009											
▷ [MKL BLAS]@dtbsv	0.468							0.000											
climate_mini_app::advection_diffusion_	0.509							0.000											
climate mini app::imex per box opera Highlighted 0 row(s):		0.000	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.001	0.000	0.000	0.000	0.000	0.020	0.000	0.000	0.130	
	2.5s 3s 3	and the second		5.5s		Ġ.5s	75	7.5s	8s	8.5s	95 0	.5s	105 1	D <u>.5</u> s 1	<u>15 11</u>	5s 12	25 12	55	V Thread V Thread Hardw Hardwa Hardwar Hardwar
★ Filter: shown ♥ Process: Any Proce Call Stack Mode: User functions + 1		▼ Th ode: on ▼ Loop	read: <mark>/</mark> Mode [.]			nlv	•		▼ Mo	dule: 🖊	ny Mo	dule				•			1



	sembly 📃 📑 🗿 🥺 🧕 🔍 Assembly grouping: <u>Address</u>		J											 	
									D-				peline Slots (S [_]		
S 🔺	Source	Clockticks				1 - 4			Back-End Boun						
L.			пс	-	emory		-	Dat	Memory Rep Dat L1D L2				emory	_	
133						DIE		Ducin	L10			Loui	Spir	Ξ	
134	<pre>ke0 = profile.horizontal_stencil(h0, FY, FZ, inv_dh_c);</pre>													H	
135	<pre>ke1 = profile.horizontal_stencil(h1, FY, FZ, inv_dh_c);</pre>													H	
136	<pre>ke2 = profile.horizontal_stencil(h2, FY, FZ, inv_dh_c);</pre>													Π	
137	<pre>ke3 = profile.horizontal_stencil(h3, FY, FZ, inv_dh_c);</pre>														
138		0.0%	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	d	
139	kE(h0) = ke0;	0.1%	0.175	0.000	0.520	0.057	0.000	0.000	0.000	0.005	0.000	0.000	0.000	d	
140	kE(h1) = ke1;	0.0%	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	d	
141	kE(h2) = ke2;	0.0%	1.000	0.000	0.000	0.900	0.000	0.000	0.003	0.000	0.000	0.000	0.000	C	
142	kE(h3) = ke3;														
143	}														
144															
145	<pre>// Handle the remainder, if there is one.</pre>														
146	<pre>for (; k <= upper[2]; ++k)</pre>	0.0%	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	d	
147	{														
148	<pre>IntVect h0(i, j, k);</pre>														
149															
150	<pre>kE(h0) = profile.horizontal_stencil(h0, FY, FZ, inv_dh_c);</pre>	0.1%	0.000	0.000	0.191	0.017	0.000	0.000	0.000	0.001	0.000	0.000	0.000	C	
151	}	1.9%	0.144	0.000	0.348	0.249	0.000	0.000	0.069	0.040	0.008	0.000	0.000	d	



💵 🖆 📚 🕩 🗟 🕸 📽 💽 🛛 r001ge 👘 r003maco	r004ge r00	5bw X		=
Bandwidth Memory Usage view	wpoint (<u>change</u>)	。		Intel VTune Amplifier XE 2016
Collection Log Analysis Target Analysis		ottom-up		
package_11600% package_01600%				
Grouping: Function / Call Stack	*	u c Mi		
Function / Call Stack	CPU Time ▼ * 23.141s	LLC Mis	Module ClimateMiniApp3d.Linux.64.icpc.ifort.OPT.HPX.XEON.ex	mkl blas avv ydger
<pre> climate mini app::imex per box operators<cli< pre=""></cli<></pre>			ClimateMiniApp3d.Linux.64.icpc.ifort.OPT.HPX.XEON.ex	
 intel memset 	13.330s		ClimateMiniApp3d.Linux.64.icpc.ifort.OPT.HPX.XEON.ex	
FArrayBox::plus	11.409s		ClimateMiniApp3d.Linux.64.icpc.ifort.OPT.HPX.XEON.ex	
▶ [MKL BLAS]@dgbmv	11.037s	0	ClimateMiniApp3d.Linux.64.icpc.ifort.OPT.HPX.XEON.ex	mkl_blas_avx_dgbmv
[MKL LAPACK]@dgbtf2	9.078s	0	ClimateMiniApp3d.Linux.64.icpc.ifort.OPT.HPX.XEON.ex	mkl_lapack_dgbtf2
[Outside any known module]	7.978s	0		[Outside any known module]
BaseFab <double>::performCopy Selected 1 row(s):</double>			ClimateMiniApp3d.Linux.64.icpc.ifort.OPT.HPX.XEON.ex	BaseFab <double>::performCopy(BaseFab<doub< td=""></doub<></double>
	Any Thread		✓ Any Module ✓ on	▼ Functions only ▼
🗴 🛛 No filtplied. 💝 Any Process	Any mread		 Any Module on 	



Intel Vectorization Adviser

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

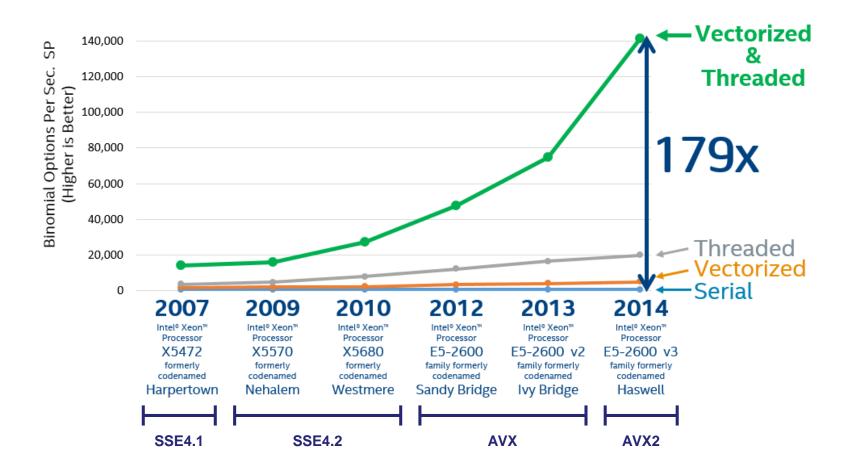
• Integrates the Intel compiler's vectorization reports into the GUI performance profiling framework.

📕 Where should I add vect	Intel Advisor XE 2016												
🍄 Summary 🚭 Survey Report 🍅 Refinement Reports 💧 Annotation Report 🐐 Suitability Report													
Elapsed time: 54.44s Vectorized Image: Sector integration of the sector integrates of t													
Function Call Sites and Loops	۵	Vector Issues	Self	Total	Trip [≫] Counts	Loop Turo	Why No Vectorization?	Vectorized Loops		^			
		🖉 Vector Issues	Time▼	Time		Loop Type	why NO vectorization:	Vecto	Efficiency				
i>७ [loop at stl_algo.h:4740 in std::tr			0.170s I	0.170s l		Scalar	non-vectorizable loop ins.						
🖃 🛄 [loop at loopstl.cpp:2449 in s234_]		💡 2 Ineffective peeled/rem	0.170s l	0.170s l	12; 4	<u>Collapse</u>	Collapse	AVX	~100 <mark>%</mark>				
₅>⊍ [loop at loopstl.cpp:2449 in s			0.150s l	0.150s l	12	Vectorized (B		AVX					
₃> 🝊 [loop at loopstl.cpp:2449 in s			0.020s I	0.020s I	4	Remainder							
±> 🝊 [loop at loopstl.cpp:7900 in vas_]			0.170s l	0.170s l	500	Scalar	vectorization possible but.						
🗄 ⊍ [loop at loopstl.cpp:3509 in s2		I High vector register	0.160s l	0.160s	12	Expand	Expand	AVX	~6 <mark>9%</mark>				
🖽 🐸 [loop at loopstl.cpp:3891 in s279_]		💡 2 Ineffective peeled/rem	0.150s l	0.150s l	125; 4	<u>Expand</u>	Expand	AVX	~9 <mark>6%</mark>				
🖽 🐸 [loop at loopstl.cpp:6249 in s414_]			0.150s l	0.150s l	12	<u>Expand</u>	Expand	AVX	~100 <mark>%</mark>				
i> 🖱 [loop at stl_numeric.h:247 in std		Assumed dependency	0.150s l	0.150s l	49	Scalar	vector dependence preve			~			
<		-					-						

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



Intel Vectorization Adviser



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



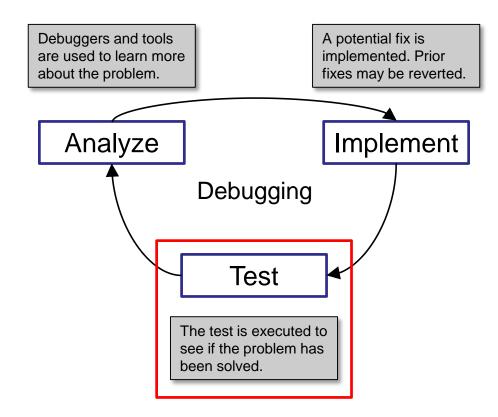


Idea: Let's write unit and regression tests for performance, just like we do for correctness.

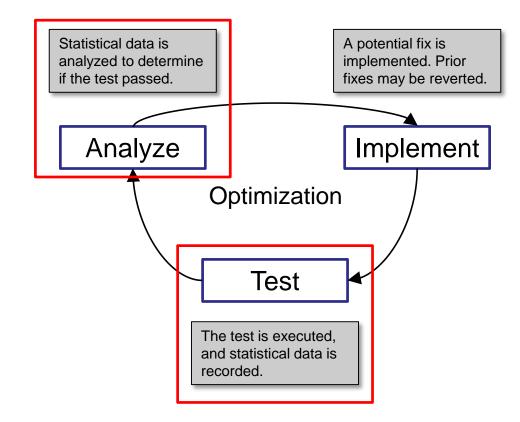
Challenges:

- 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.











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.



Stateful Performance Tests





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.



Stateless Performance Tests

Whenever you face a design trade-off with performance implications, write a stateless test!



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.









UNIVERSITY OF CALIFORNIA