Five ways not to fool yourself

or: designing experiments for understanding performance

Tim Harris
10 February 2018
https://timharris.uk

Performance experiments are often used to show that a new system performs better than an old system, and to quantify how much faster it is, or how more efficient it is in the use of some resource. Frequently, these experiments come toward the end of a project and – at times – seem to be conducted more with the aim of selling the system rather than providing understanding of the reasons for the differences in performance or the scenarios in which similar improvements might be expected. Mistrust in published performance numbers follows from the suspicion that we optimize what we measure or that we measure what we have already optimized.

This article summarizes some of the techniques I use in performance evaluation in order to try to get a better understanding of why a system behaves as it does, and why changes I make to the system lead to differences in performance. In part, the aim of these techniques is to be able to explain the performance of the system better when presenting it to other people, but the aim is also to provide me with a better understanding of the system while working on it, and to avoid me fooling myself over why some change has some particular effect.

The examples here focus mainly on multi-threaded algorithms running on a single machine—problems such as building a shared-memory hash-table with multiple threads manipulating the table at the same time, or implementing a graph analytics algorithm such as PageRank walking over a graph held in memory. While those kinds of example provide a concrete focus and a source of "war stories" from my own work, I hope that many of the techniques will carry over to other settings.

1. Measure as you go

Early in a project, develop good test harnesses for running experiments and scripts for plotting results. Automate as much as possible. Plot what you measure. Be careful about trade-offs. Beware of averages.

Developing test harnesses early in a project provides time to refine the experiments being run and the results being plotted from them—I find that both of these steps usually take longer than expected.

In addition, being able to run tests on an existing system helps characterize how well it performs already (and perhaps lets you estimate the scope for possible improvements). For example, in our work on co-scheduling parallel OpenMP workloads we were able to identify which combinations of workloads showed particular problems and which combinations worked well (https://timharris.uk/papers/2014-eurosys.pdf, Fig 1). Identifying these examples helped focus our attention on cases which were not performing well.

I try to automate as much as possible, usually with one script running the complete set of experiments needed for a given paper or presentation, and then a second script to take the results from those experiments and to create graphs / heat-maps / etc. from the raw data. Splitting these steps into two scripts lets me re-plot the same data in different ways while experimenting with graphical formats. Automation is there to help avoid errors (consistent environment each time, and no problems with cut-n-paste of results into Excel), and also to avoid manual work being an impediment to repeating experiments (e.g., I have scripts to generate PDF-format graphs to include in papers via LaTeX, and EMF formats of the same results to include in PowerPoint).

I try to follow a rule: plot what I measure, not what I configure. For example, if a test is supposed to add 1M entries to a hash-table then I will have it count the number of entries it adds and output that along with the timing. If a test is supposed to run with 16 threads pinned to different cores, then I will have each thread query the OS for the core it is running on, and then count the number of different cores that are seen. This approach guards against a whole set of bugs in which a configuration setting does not make it through to the program under test—e.g., experiments may be misconfigured by typos in environment variables, or their might be bugs in string parsing.

Ideally, I try to explore performance over a range of workloads. For instance, in experiments on a shared-memory hash-table, I would include workloads which are read-only through to workloads which are 100% updates. I would also vary the size of the hash-table from the smallest plausible size (fitting easily in the last-level cache) up to the sizes which comfortably exceed the size of the caches. Comparing algorithms across this range of scenarios helps to identify trade-offs, and to show whether a new algorithm is uniformly preferable to an existing one, or whether it has costs as well as benefits (perhaps an algorithm using optimistic concurrency control gives a significant improvement in read-mostly cases, but is very poor for writes).

Finally, when reporting results, be careful about the use of averages. In multi-threaded workloads I try to report results on a per-thread basis in addition to overall results such as the aggregate throughput across all threads. I tend to have programs report this kind of per-thread detail in their output and then to perform averaging during post-processing. This lets me generate "internal" versions of results showing more detail than I would include in a publication—for instance, I would look at the rates of progress of different threads, or a histogram or CDF of the completion times of requests.

Even if a workload looks symmetric with all threads attempting an identical set of operations, it can happen that some threads make much faster progress than others—sometimes this happens because of system effects (some threads are on CPUs physically closer to the memory that they are accessing), but sometimes it happens because some threads are "lucky" in access to locks or some other resource, and once lucky they can continue being lucky (https://dl.acm.org/citation.cfm?doid=2612669.2612703). This kind of skew in performance can become a significant problem in some cases if it leads to straggler problems, and it is not unknown for improvements in aggregate performance to occur alongside reductions in fairness across threads.

2. Include lightweight sanity checks

Incorrect algorithms are often fast! Build in correctness checks that are sufficiently cheap to leave on in all runs. Do not output any results if the checks fail.

It is easy to accidentally make an algorithm incorrect while trying to optimize it. I try to guard against this by including some always-on correctness checks—either as a separate non-timed part of each experimental run, or ideally by making the checks fast enough to keep them enabled all of the time.

For example, in some of our early work on transactional memory (TM), we had experiments with threads swapping pairs of items in a fixed-size array. The array would start with N distinct values in it, and at the end of the test we would expect there still to be N distinct values. More often than not, bugs in the TM system would manifest as values getting duplicated in the array. In more complex cases we might have a set that starts with S items, and a workload during which N new items are inserted and D items are deleted—at the end of the test we would expect S+N-D items left in the set. The overhead of these checks can be reduced by having each thread count its own numbers of successful insertions and deletions (and taking care over issues like false sharing between per-thread counts).

In other cases it is possible to check invariants over a data structure's representation. For instance, in the implementation of a balanced binary tree, are the depths of the shortest and longest paths to leaves approximately equal? Does each node in the tree have exactly one parent? In an ordered tree, are the values actually in order? The checks can be made after the timed portion of an experiment has run.

For safety it is best to perform these checks before outputting any timings or other results from the experiment. That guards against the risk that scripts for processing the results will pick up the timings but miss the fact that the run contained errors.

3. Understand simple cases first

Start with simple experiments whose behaviour should be easy to understand, even if these tests are not representative of real systems, or actual benchmarks that people will care about. Do these simple experiments behave as expected?

There are many sources of complexity in production environments: Machines may be shared between multiple workloads. The operating system may dynamically move threads between cores or between sockets. Hardware may dynamically control the power settings for CPUs or other devices. Multiple threads may share a core—sometimes with their own hardware treads, and sometimes multiplexed over a single hardware thread. Memory pages may vary in size, or be migrated by the OS between sockets.

The list of possibilities is long, and changes between systems. Worse, these kinds of behaviour often occur due to interactions across different parts of the system—e.g., a change to the order in which threads are created in a workload may change the way that the OS places the threads onto the CPUs in the machine, which in turn may change the socket on which memory is allocated by the OS to those threads. The risk is that these kinds of unintended changes can have more of an effect on performance than the actual difference between two versions of a workload.

I try to mitigate these risks by starting with simple experiments in which everything is "nailed down", meaning that threads are pinned to specific CPUs, memory is allocated on fixed sockets, hyper-threading is disabled, and power management features are turned off (as far as possible). Following the rule to "plot what I measure, not what I configure", I have threads query the OS to check which CPU they are running on, and I check the socket that data structures have been allocated on, and the page sizes being used (e.g., using the APIs from libnuma on Linux).

In these tests I try to make the workload as simple as possible—in the case of an in-memory hash-table, this might mean populating the table with a random mix of values, and then having the timed part of the experiment involve threads performing read-only look-ups on the table. For a parallel hash-table I would start by looking at runs with just 1 thread and comparing the performance against that of a simple sequential hash-table—is the overhead low? After that, I would extend the number of threads and expect near-perfect scaling, and I would expect near-identical results from one run to the next.

This kind of test is unlikely to correspond to a workload that anyone will actually care about. However, it is useful to run because the results often *do not* come out as expected, and it is usually then much simpler to identify the reason for the unexpected performance and to fix it in this simple controlled setting. For instance, continuing with the example of read-only look-ups on a hash-table, a lack of scaling may be caused by an occurrence of false sharing, or perhaps by contention for a lock within a library, or within the OS.

I will also use these simple workloads to make some additional basic checks: Are the experiments running for long enough that they are no longer subject to warm-up effects—if I double the duration of the experiments then do the results change? If I measure memory consumption during the experiment then has it become stable? If I measure the CPU consumption in the experiment then is it behaving as expected (e.g., can the mix between user-mode and system-mode execution be explained?). If I watch the system-call activity with strace then do I see unexpected system calls, or unnecessary filesystem accesses, etc.?

I keep these simple workloads in a test suite, even once they are performing as expected. This helps pick up any performance regressions in the future.

4. Look beyond timing

Try to identify the reasons for the differences between the performance of workloads. Look for differences in resource utilization, and statistics from performance counters.

Once I have simple tests performing as expected, I try to understand the reasons for the differences in performance between – say – the old version of an algorithm and a new optimization that I am working on. I will continue to run these tests “nailed down” in order to avoid possibilities that different versions of the code may provoke different behaviour from the OS or the hardware (examining those differences and their consequences may be important for production settings, but in the interest of decomposing the problem I will start with the simplest case).

The aim is to try to link three aspects of the system:

- The overall timings or performance measurements from an experiment (say, the time to execute an iteration of PageRank on a graph, or the throughput of operations on a hash-table).

- Measurements of resource use during the experiment (say, the number of instructions executed, or the number of misses from the last-level cache).

- Differences between the algorithms being executed.

The central idea is that linking changes in code through to changes in performance in this way helps gain confidence that there is a genuine causal link: i.e., that the change in the code leads to an increase/reduction in the use of some resource, and that in turn explains the increase/decrease in performance that is measured.

For example, in many in-memory graph analytics algorithms, there is a choice between using an "array of structs" representation of data (with a single array indexed by graph vertex ID, holding multiple pieces of information about that vertex) or a "struct of arrays" (with multiple arrays, each holding one piece of information—as in a columnar database). Different representations work well in different settings, but in one particular scenario I was considering combining two arrays into a single array-of-structs, under the expectation that both fields in the resulting 2-element structs would be needed together, and that storing them together would halve the number of cache lines fetched (and, in addition, under the hypothesis that the rate of cache line fetches was limiting performance). In that case I did achieve approximately 2x performance with the array-of-structs representation. I checked with the CPU performance counters that the number of last-level-cache (LLC) misses per PageRank execution was approximately halved, that other performance counters were generally unchanged (e.g., that the number of memory accesses and the number of TLB misses were the same), and that the memory throughput was unchanged (measured in GB/s, with it being the bottleneck in both cases).

Quantitatively, there are two useful ways to look at this: First, the back-of-the-envelope costs of different operations in an algorithm (as in, for instance, one of Jeff Dean's talks https://research.google.com/people/jeff/stanford-295-talk.pdf). Are the changes made to an algorithm roughly consistent with the changes in performance? Another way that I sometimes explore a workload's performance is to deliberately *add* additional operations or additional use of a particular resource and then to see if that *harms* performance—for instance, forcing a program to perform additional division operations, or forcing it to incur extra TLB misses. The intuition here is that it is usually easier to add work to an algorithm than it is to optimize it. If this additional work does not harm overall performance, then that helps rule out the significance of that particular kind of operation. The Coz system uses this technique methodically in code profiling (https://github.com/plasma-umass/coz).

In addition to those back-of-the-envelope costs, the second quantitative technique I use is to look at the maximum capabilities of a given system (e.g., the maximum rate at which the CPU can execute a particular kind of instruction, or the maximum bandwidth between the LLC and main memory). Is a given algorithm close to these limits? Is a change trading off the use of some bottleneck resource for some other resource that is more plentiful?

On most CPUs there are hundreds of performance counters and it is not possible to collect all of their values all of the time. Generally I will start by collecting results from a few different kinds of run: (1) running without any performance counter data, (2) running collecting information about the workload's structure (counts of instructions, numbers of memory reads, numbers of floating point operations, numbers of exceptions/traps), and (3) collecting information about memory system performance (TLB performance, miss rates at different levels in the cache, and ideally b/w use on each memory channel). The aim in #1 is to confirm that there is no significant overhead from using performance counters. The aim in collecting #2 is to confirm that there are no unexpected differences in behaviour between workloads (say, if a change to an algorithm leads to an unexpected increase in the number of instructions because it prevents a compiler optimization).

After ruling out those problems, comparing #3 across workloads is usually most relevant for my work. I will be looking at these numbers both to see where there are intended differences between workloads (such as the array-of-structs optimization reducing LLC miss counts), and also to confirm that there are no unintended differences (e.g., that changing array representation does not add or remove skew between memory channels).

One final cautionary tale: be careful about "per cycle" metrics, particularly if you are taking measurements with power management enabled (and hence with the duration of each cycle changing in response to the load). I tend to focus on counts of operations taken across the complete execution of an algorithm, or on per-second bandwidth numbers that are independent of cycle time.

5. Moving toward production settings

Examine the extent to which observations made in simple controlled settings carry over to more complex environments.

Once I think I understand the behaviour of a workload pinned to particular hardware resources, I will then start removing these constraints and checking whether the behaviour changes in a more general setting. I will usually start by enabling power management features (but leaving threads and memory placement fixed), and then remove controls over memory placement, and finally run with thread placement handled by the OS.

Much of my work is based on comparing old vs new versions of an algorithm, and in those scenarios this final generalization step usually proceeds in one of two ways: first – and perhaps ideally – the old-vs-new comparison may be largely unchanged; perhaps enabling a feature such as Turbo Boost on Intel CPUs helps both algorithms, but their relative performance remains similar.

The more frustrating case is that the relative performance of the different algorithms diverges from what was seen in the controlled settings. Unfortunately there are many possible explanations for this, and usually some investigation is needed to identify what is happening (indeed, it might be prudent to do this in any case—even if performance looks consistent with the controlled setting, it might be that there are multiple factors involved which happen to cancel out).

Generally I will try to decouple a number of aspects of this problem and start by measuring where the OS is placing the threads and memory in the different workloads, and the actual frequencies that the CPUs are running at:

- If these measurements show *the same* placements are occurring between the workloads then the question becomes "Why do the workloads behave differently with this placement, compared with the simpler placement used previously". Measuring CPU performance counter results and memory bandwidth use may provide some guidance into the source of the difference. If not, then a further approach is to manually pin threads and memory again and to explore additional placements.

- If the measurements show *different* placements for the different workloads, then the question becomes "Why is the OS allocating resources differently in these two cases"?

  If the difference is in terms of memory placement then it is worth investigating how memory is allocated in the workloads—e.g., if a first-touch policy is being used then whether memory is initialized by being written to by different threads in the two cases, or perhaps there is a difference in how objects are freed and re-allocated.

> If the difference is in terms of thread placement, then this is going to be dependent on the specific software versions involved and the exact reason may be difficult to identify. In some cases it may be due to the way in which threads are created in the workload, or the way that they synchronize with one another (blocking in the kernel, versus spinning in user mode), or even the previous state of OS scheduler data structures when starting the workload.

Conclusion

This article is a work-in-progress; suggestions are welcome.

My emphasis here is more on how I try to structure my work and performance investigations; this does not necessarily translate directly into how I would present work in a research paper or a presentation.

One idea which I do now try to emphasise in papers and presentations is to explain the causal link between a change to an algorithm and the resulting change in performance—not just that a new system is better than an old system "by up to 400%", but that (1) the change in the algorithm leads to a change in the use a given resource or the rate at which a particular operation occurs, and that change is sufficient to explain the performance trends seen, and also (2) that the effects are not due to extraneous changes (interaction with compiler optimizations, OS thread placements, etc.).

Other resources

The title of this article is inspired by Bailey's classic description of "Twelve Ways to Fool the Masses" (http://crd-legacy.lbl.gov/~dhbailey/dhbpapers/twelve-ways.pdf). That paper focuses on how to present results from supercomputing in the most favourable light. McSherry *et al*'s "Scalability! But at what COST?" touches on related issues in the era of "big data" (http://www.frankmcsherry.org/assets/COST.pdf).

Jain's textbook "The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling" covers many aspects of performance analysis, including simulation and queueing theory along with the design of experiments and handling of statistics (https://www.amazon.co.uk/Art-Comp-Systems-Perform-Analysis/dp/0471503363)

Huff's "How to lie with statistics" is a lighthearted introduction to sampling, averages, errors handling, graphs, "and other tools of democratic persuasion" (https://www.amazon.co.uk/How-Lie-Statistics-Penguin-Business/dp/0140136290). Fleming and Wallace's "How not to lie with statistics: the correct way to summarize benchmark results" describes the choice between using the arithmetic mean and geometric mean when summarizing sets of results (https://dl.acm.org/citation.cfm?id=5673). These means (and others) still seem to be used on an excessively ad-hoc basis.

Heiser's "Systems Benchmarking Crimes" (https://www.cse.unsw.edu.au/~gernot/benchmarking-crimes.html) include the inappropriate use of the arithmetic mean, along with many other recurring problems such as selective reporting of results, comparison against improper baselines, and poor statistical techniques. The full set of examples is useful reading; examples of the "crimes" are frustratingly common.

Mytkowicz *et al*'s paper on "Producing Wrong Data Without Doing Anything Obviously Wrong!" shows how seemingly innocuous changes to an experiment's setup can lead to performance differences of the same magnitude as many compiler optimizations (http://cis.upenn.edu/~cis501/papers/producing-wrong-data.pdf). If these factors are not understood then they can lead to incorrect conclusions being drawn. The *Stabilizer* system provides support for automatically and repeatedly randomizing the placement of functions, stack frames, and heap objects, thereby helping to avoid recurring anomalous layouts (http://plasma.cs.umass.edu/emery/stabilizer.html).

Brown's examples from benchmarking concurrent data structures show how the conclusions from performance results can be dependent on details such as the choice of memory allocator, and the choice of thread placement (https://www.youtube.com/watch?v=x6HaBcRJHFY). Along with the "Producing Wrong Data…" paper, this motivates my emphasis on understanding the reasons for performance results rather than simply presenting overall timing results.

Hoefler and Belli's recent paper "Scientific Benchmarking of Parallel Computing Systems" assesses the state of recent practice in HPC publications and proposes techniques to present data in a way that lets the reader interpret what is being shown and to draw their own conclusions (https://htor.inf.ethz.ch/publications/img/hoefler-scientific-benchmarking.pdf).

The emerging SIGPLAN empirical evaluation guidelines provide a great checklist of best-practices to follow across the programming language research community (http://www.sigplan.org/Resources/EmpiricalEvaluation/).

Acknowledgements

This article is based on a talk I originally gave at a NICTA summer school in 2016, and feedback there from Gernot Heiser, Steve Blackburn, and others. Subsequent versions of the talk benefited from many discussions with Dave Dice (see Dave's blog for

interesting descriptions of performance problems in production systems https://blogs.oracle.com/dave/).