Irreproducible Benchmarks Might Be Sometimes Helpful

Jochen Liedtke Nayeem Islam Trent Jaeger Vsevolod Panteleenko
Yoonho Park

Thomas J. Watson Research Center
IBM
Hawthorne, NY 10532
{jochen,nayyem,jaegert,vvp,yoonho}@us.ibm.com

1 Why Benchmarks?

Historically, benchmarks have been used for commercial purposes. A customer develops or selects a benchmark that generates a load that is considered to be typical for her/his applications. The benchmark is executed on various machines to find the most appropriate vendor and machine.

In OS research, we use benchmarks for completely different purposes that can be subsumed under the headline "understanding a complex system":

1. Rating:
   We run a benchmark, then we modify something in the examined system (hardware or software), and run the benchmark a second time. We use the observed performance difference between both benchmarks as a measure characterizing the effects of the modification.

2. Discovering experiments:
   We run a benchmark and trace, count, or measure certain events, e.g. cache misses, system calls, hits in the file cache, etc.
   With such experiments, we try to "understand" a system and to find a more or less sophisticated model for it. In the most ambitious case, such a model would allow us to predict the effects of modifying the system; in the simplest case, it should at least help us to identify bottlenecks.

3. Validating experiments:
   Once we have a model or theory, we use benchmarks to validate or invalidate our theory. If the results we get are in accordance with our predictions, the theory might be right. Otherwise, we have to modify our model, since we still do not understand what is going on.

Rating (1), a typical engineering method, gives us first hints to guide our research; however, it needs to be substantiated by understanding (2,3), a typical method of science. Without this understanding, benchmarks can be useless because they then can suffer from ugly "nonfeatures":

- *Non-Transitivity:*
  Combining two non-unerstood optimizations that both have proven their value by benchmarks may result in the opposite effect.

- *Instability:*
  Even though an optimization improves one benchmark, it might have no positive effect on applications that are considered to be very similar to said benchmark.

- *Non-Linearity:*
  Executing *n* benchmarks concurrently can show a completely different behaviour.

We use microbenchmarks to analyze effects in isolation. Although in many cases we have a better understanding of microbenchmarks than of macrobenchmarks, we need macrobenchmarks to analyze effects in the complex context of a "realistic" system. Then, the *realism*, the *analyzability* and our *understanding* of these macrobenchmarks is crucial for their relevance.

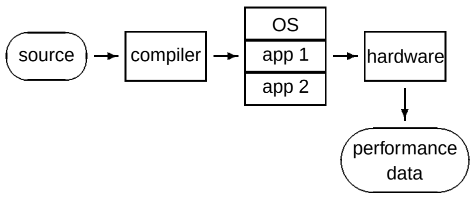
Generating such realistic, analyzable and understandable benchmarks for batch-type scenarios (few, long-running, non-interactive jobs) or for mono-server scenarios (one server, incoming requests more or less randomly) seems to be understood to a certain degree. However, it is hard to generate a realistic benchmark for an interactive system like a workstation. Assume you have recorded a complete user interaction sequence with the system. The two most critical problems for replaying this recording are:

a) It is not obvious when to trigger the next user action. You need a very sophisticated external system that synchronizes the virtual user with the system to be measured: The virtual user should move the mouse to a button and click on it when the button is displayed, not before and not long after.

b) Even worse, not only the time but also the sequence of user actions is influenced by the system's behaviour. Assume a user compiles a program in one window and types a letter in another one. If in a later experiment, the compiler executes faster, the virtual user could realistically stop typing earlier (when the compilation finishes), switch back to the first window, do some work, and then finish the letter. In the original experiment, the user could finish typing the letter before the compilation was ready.

Classic Benchmarks

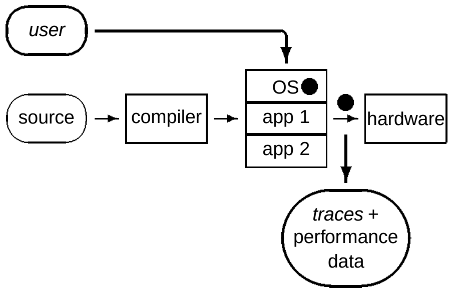
A classic benchmark has a fixed set of source programs as input. They are compiled and then executed under an OS on a hardware system:



The immutability of the source ensures the reproducibility. Thus any experiment with a modified compiler, operating system, or hardware, measures the performance effects of the modifications relative to the benchmark.

2 Stochastic Benchmarks

When we extend the system by a human user or a similar source of unpredictable and irreproducible inputs, we loose the strict reproducibility:



However, we can record the system's behaviour inside the operating system and at the interface between software and hardware (denoted by the bullets). Although the real benchmark input is not reproducible, the recorded traces are. So we could at least replay them against modified hardware and get a more realistic benchmark for hardware systems. In Section 3, we will discuss further potential applications.

To get realistic and representative data, we would have to record hours, perhaps even days or weeks of workstation usage. The according traces would be huge: in one hour, about one trillion (10^{12}) instructions can be executed. Even the best compression algorithm cannot manage this amount of data. Even worse, we would need very expensive logic-analyzer hardware to get these traces. Any software-implemented system-global tracing at the instruction level would slow down the system dramatically and thus heavily distort the interactive behaviour of a human user.

Stochastic Probing

Both difficulties can be overcome by *random sampling*.

In OS research, random sampling techniques have been used for minimally-invasive measurements, e.g. for continuous profiling [1, 4, 5] and for cheap fine-grained time measurements [3]. Because of its low costs and it stochastic nature, random sampling is well suited for measuring real systems realistically.

Stochastic benchmarks try to apply this principle to benchmarking: For stochatically distributed

monitoring intervals, the entire system is monitored. Depending on the benchmark, low-level or high-level events may be traced: memory accesses, or instructions executed, or file accesses, or disk read/writes, or ethernet packets, etc. In essence, the recorded traces *are* the stochastic benchmarks.
Four properties are important:

1. The intervals must be selected at random to exclude systematic influences. Nevertheless, systematic restrictions are possible; e.g. one could exclude all low-activity intervals from monitoring.

2. The system in its entirety is traced during a monitoring interval, including kernel and device drivers.

3. The intervals have to be contiguous and long enough so that their analysis delivers valuable information. Monitoring 100,000 memory accesses, for example, is probably sufficient to run it against a different L1-cache architecture; monitoring 1,000 instructions may be sufficient to see the effect of a different code-generation technique.

4. The intervals have to be short enough to prevent the user from noticing them. In particular instruction-level tracing slows a system dramatically down. However, if this happens only for 1/20 second, a human user will probably not realize it.

Dropping the reproducibility automatically eliminates the above mentioned problem (a). To a certain degree, the short monitoring intervals also make the benchmarks robust against (b). The randomization of monitoring intervals hides changes in the sequence of user actions as long as the overall behaviour of a human user remains basically the same. (Of course, dramatic improvements in performance will change a user's behaviour. Probably the user will even change its favorable applications. But then, we need to draw a new benchmark anyhow, since the old one becomes unrealistic.)

3 Potential Applications

On the one hand, dropping the reproducibility increases realism and widenes the applicability of benchmarks. On the other hand, the replay-only feature of a stochastic benchmark severely restricts the possible experiments. For instance, due to their different APIs, there is no simple way to replay something generated on an NT system against a Unix system. What replay experiments are possible?

Architecture Experiments

Architecture experiments are good candidates for stochastic benchmarks. For instance, we can run stochastic traces against various cache or TLB architectures. Page-table schemes, second-level software TLBs, etc., can be evaluated in a similar way. Years ago, Hill and Smith [2] found that direct-mapped cache architectures perform well enough for many batch-type applications. Newer processors, with their multi-level caches and dramatically increased cache penalties might behave substantially different. In particular, it is not clear how object-oriented programming, fine-grain multithreading, increased interactivity with users and increased use of libraries affect the cache performance in real life.

Code-generator experiments might also to a certain degree be possible. Once we have random instruction traces for a known set of programs and a specific compiler, we can translate the traces against a modified code-generation policy and run the modified traces against a simulation of the original hardware.

Interpreting Traces Semantically

For higher-level experiments, it is essential to interpret the traces by labeling the traced instructions with higher-level semantical information. This can be done by using our knowledge (i.e., the binaries and sources) of the operating system and perhaps even of some applications that were used to produce the traces. We can identify semantic points like "enter kernel", "context switch to thread *x*", "read *n* bytes from file *y*", "exit kernel", etc. When we keep binaries and sources together with the traces, we can refine such an interpretation or even reinterpret it at a later time.

The information about context switches, interrupts, page faults, etc. that is part of such an interpreted trace permits us to disentangle the traces and get per-thread views of them.

OS Experiments

The above mentioned techniques enable us to simulate binaries or sources that are to a certain degree modified. Examples for fine-grain OS experiments are:

– Run the traces against a different scheduling policy, e.g. shorter time slices, different priorities, modified scheduling of interrupt handlers.

– Modify the page-allocation policy of the OS and see how the L2-cache misses and the application's execution time are affected.

– Optimize selected parts of the OS, e.g. change the cache working set of the fast ipc path, and analyze the resulting effects on the entire system.

– Generate a synthetic real-time load and analyze how it interferes with the stochastic traces.

Instead of tracing all memory accesses, we can apply the stochastic-benchmark techniques as well on selected OS-internal events only. Since they occur with much lower frequency, we can trace them for order-of-magnitude larger intervals than instructions. Combining of both types of traces could enable experiments with coarse-grain OS policies, e.g. modified paging policies.

Similar techniques can be used to measure and compare application-specific policies and algorithms. Monitoring could in this case be restricted to the according application. However, monitoring the entire system permits us to measure the interference costs with the OS and other applications.

Furthermore, we could think about statistic experiments that are inspired by medical studies and are not really benchmarks: For comparing the original hardware/software system S and the modified system S , switch randomly between both systems and stochastically monitor both systems. Of course, the system switches must be invisible to the user.

4 Fundamental Problems

Uncertainty By Incompleteness

This problem is best illustrated using an n-way set-associative cache. A monitoring interval gives us a sequence of memory accesses which can be easily mapped to the different sets of the cache. Unfortunately, we cannot decide for the first n accesses per set whether they are hits or misses. This uncertainty comes from incomplete information: we monitored only a limited interval and have no information about the global system state (the hot cache in this case) when the interval began.

One might propose taking a snapshot of the system state at the beginning of each monitoring interval. For a large L2 cache, this could be rather expensive. Sometimes, it is even technically impossible. However, the largest disadvantage of snapshots is that they help to analyze the first measurement *but they do not help benchmarking.* If we run the recorded traces in a second benchmark experiment against a different cache architecture, we cannot expect the same initial cache state per interval as we found with the original cache architecture. So the old snapshots are obsolete.

Similar problems come up for most of the previously mentioned experiments. Incomplete information about the system's state incurs uncertainty. Therefore, we look for general methods to reduce this uncertainty. Currently, the most promising method is based on the following idea:

i) Divide each monitoring interval into two halves.

ii) Count the events in the second half that would be uncertain if you would not know the first half.

iii) Find out how many of them have become certain (and whether they are hits or misses) if you use the information of the first half. This maps some originally uncertain events to hits, some to misses and leaves some uncertain.

iv) With a high probability, the same percentages apply to the uncertain events of the first halves (which are real uncertainties). So the uncertainty is probabilistically reduced.

A precise description of this method as well as a discussion of why it works and why many simpler methods do not work is beyond the scope of this extended abstract.

A Probabilistic Theory of Feedback

Most of the experiments mentioned in Section 3 include feedback possibilities. For example, changing the page allocation might influence the cache hit rate and therefore also change the preemption points. It is not clear up to which degree the original traces remain valid. We feel that we need a probabilistic model of these feedbacks. Currently, this problem is completely open.

References

[1] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? In *16th ACM Symposium on Operating System Principles (SOSP)*, pages 1–14, St. Malo, October 1997.

[2] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, December 1989.

[3] J. Liedtke. A short note on cheap fine-grained time measurement. *Operating Systems Review*, 30(2):92–94, April 1996.

[4] S. E. Perl, W. E. Weihl, and B. Noble. Continuous monitoring and performance specification. Technical Report 153, DEC Systems Research Center, Palo Alto, CA, June 1998.

[5] X. Zhang, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith. System support for automatic profiling and optimization. In *16th ACM Symposium on Operating System Principles (SOSP)*, pages 15–25, St. Malo, October 1997.