BENCHMARKING "Hello, World!"

"A good performance evaluation provides a deep understanding of a system's behavior, quantifying not only the overall behavior, but also its internal mechanisms and policies. It explains why a system behaves the way it does, what limits that behavior, and what problems must be addressed in order to improve the system."[10]

SIX DIFFERENT VIEWS OF THE EXECUTION OF "HELLO, WORLD!" SHOW WHAT IS OFTEN MISSING IN TODAY'S TOOLS

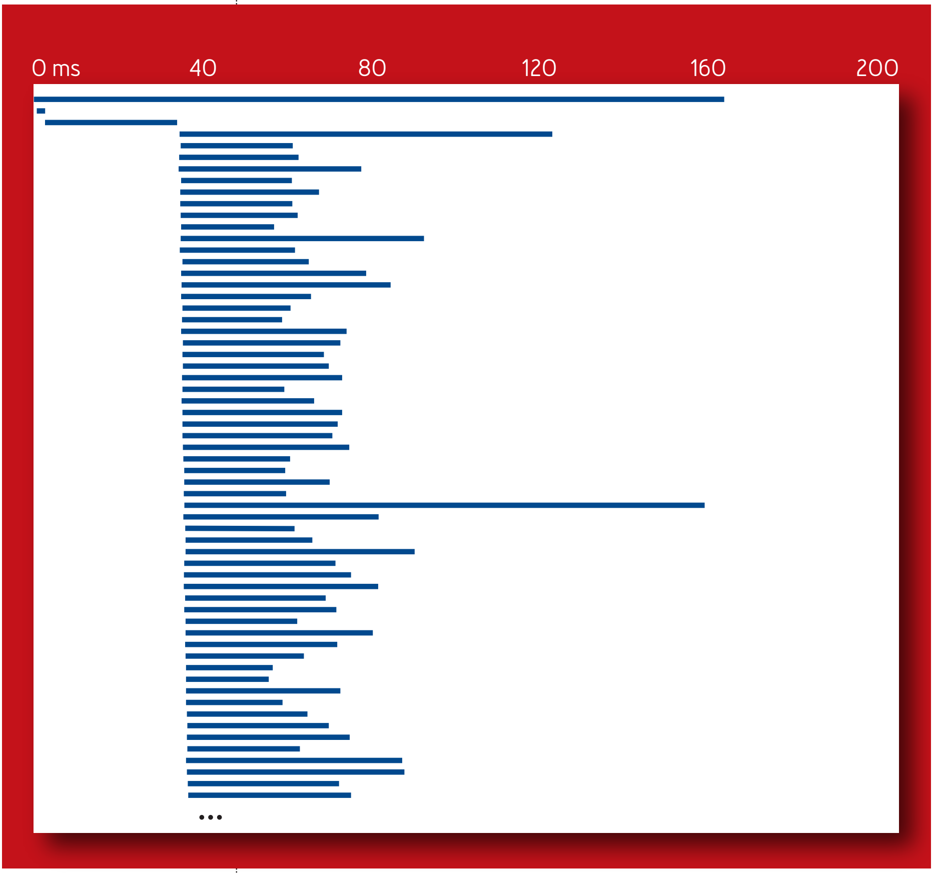
RICHARD L. SITES

As software moves off the desktop and into data centers, and cell phones use server requests as the other half of apps, the observation tools for large-scale distributed transaction systems are not keeping up with the complexity of the environment. Exploring a simpler environment can help expose some of the problems that confront today's tool users and tool builders. There is a lot to be learned from careful observation of a program and its complete surrounding context, even one as trivial as "Hello, World!". This article walks through six different views of the execution of "Hello, World!" to see what is often missing in today's tools; the same analysis applies to complex data-center software. Tool designers and tool users must insist on filling in the gaps. *If you can't see it, you can't fix it.*

Too often a service provider has a performance promise to keep ("99 percent of all web-page requests will be served within 800 milliseconds...") but few tools for measuring the existence of laggard transactions, and none at all for understanding their root causes. Such promises are built on sand.

Figure 1 is part of a single web-search query that takes 160 milliseconds, shown across the top line, with the partial call tree of the remote services it uses shown

FIGURE 1: **DAPPER OUTPUT**



underneath (circa 2005). That query is, in fact, spread out across some 2,000 servers, each doing part of the search. The diagram shows part of the top-level RPC (remote procedure call) tree, with the work distributed across time on the x-axis and across different servers on the y-axis. After preliminary calls at the upper left, there are 93 parallel calls to subsearches done on 93 different servers (not all shown). These take varying amounts of time to return, from about 30 to 120 milliseconds. One in particular takes about four times longer than the rest. The top-level server executes very little code and is mostly waiting for this straggler to finish. Without observing these cross-server dynamics, we have no idea why the total query takes so long. But even with good RPC tools such as Dapper[12] and its lookalikes, the question remains why an individual server doing a CPU-bound piece of the work takes four times longer than others doing nearly identical work.

Why does this matter? Because most transaction systems provide apparent speed by doing many suboperations in parallel, returning an overall result when all suboperations complete. At about 100:1 parallel fanout, this means that *almost all top-level searches run at the speed of the 99th-percentile slowest subsearch.* That is why it matters. With 1,000 transactions per second, there are (by definition) 10 of them every second that exceed the 99th-percentile response time. What tools can be used to find the root causes of such slow responses?

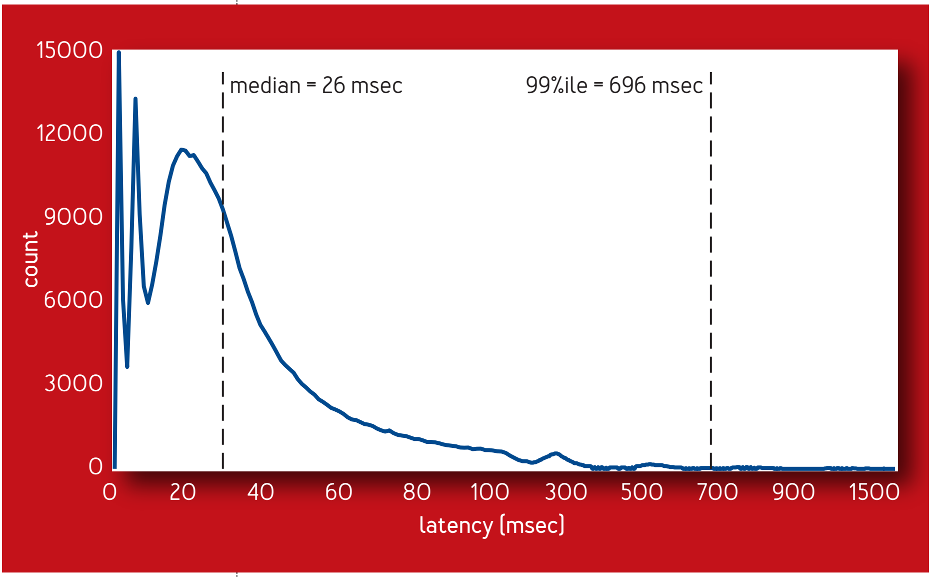
As Sun Microsystems' Bryan Cantrill[1] observed more than 10 years ago, there have been two profound shifts of observation: from development to production and from programs to systems. Looking at an empty development system running a single program provides little insight. Today we must look at live production systems running dozens of programs simultaneously to understand their real performance and the underlying software dynamics. But we must look with tools that show enough information to be useful in that context—in particular, tools that can help reveal why that laggard server response took four times longer than the others. Traditional benchmark programs by design have repeatable performance and can be studied in isolation. Online transactions, even the same transaction repeated, vary and do not have repeatable performance. Observation tools for benchmarks are generally not very useful for transactions.

The problem of poor tools can be shown by exploring the simplest of all programs, helloworld.c, slowly revealing the underlying dynamics that are invisible in most of today's performance tools. Who would bother to benchmark "Hello, World!"? Hold your skepticism that this is so trivial that it is uninteresting.

Programmers have pictures in their heads of what their programs are doing. Those pictures are too simplistic and essentially always wrong. Sometimes the performance problem they are looking for is in the missing parts, or in the not-visible dynamics of "event A causes event B", with time sequencing and cause and effect not observed at all.

Especially "interesting" laggard transactions are those that take too long on a heavily loaded production system but not on a development system or a nearly idle production system. On a single server, such laggards can be caused by complex interactions and interference from other transactions or other programs.[5] It is not predictable which transactions will encounter interference, and they typically run at full speed if repeated. A histogram of response times in such an environment shows a long tail of infrequent slow responses, with a 99th-percentile response time that is perhaps 10 times the average response time, as seen in figure 2. The figure is a latency (response time) histogram of a single disk read from a network disk-server program (circa 2012). The median is 26 milliseconds, but the 99th-percentile latency is 696 milliseconds—25 times slower. (The time scale in the diagram changes by 10 times at 100 milliseconds and again at 1,000 milliseconds). Google's Jeffrey Dean and Luiz Andre Barroso[2] discuss data-center tail latency in more detail.

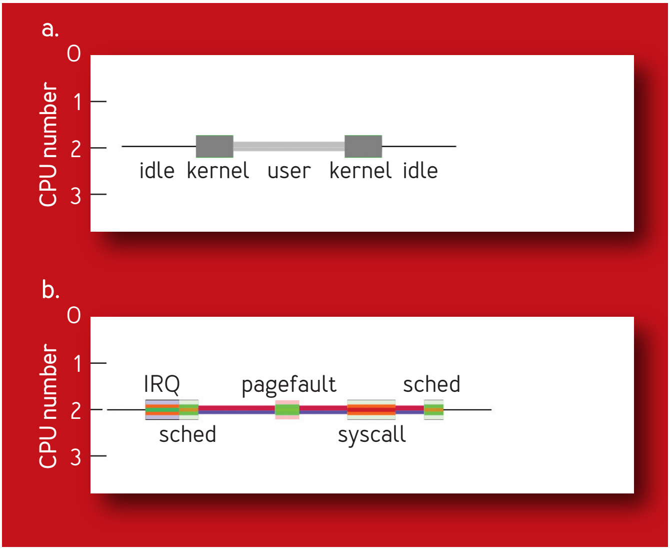
FIGURE 2: **LATENCY HISTOGRAM**



TODAY'S OBSERVATION TOOLS

In exploring possible performance-observation tools, it is important to understand what is missing from the tool's purview. Too much missing means uninformed guesses or misdirected effort. You could understand a lot about the software running on a single server if you could see what each CPU core is doing every nanosecond. Then you could draw little diagrams, as in figure 3, showing elapsed time on the multiple cores of a single server. Figure 3 shows activity on a single CPU core #2, with elapsed time running left to right, thin black lines showing nanoseconds spent in the idle loop, half-height rectangles showing user-mode execution time, and full-height rectangles showing kernel-mode execution time. Times are shown in gray on the top

FIGURE 3: **ACTIVITY ON A SINGLE CPU CORE #2**



and with colors on the bottom.

Colored kernel-mode full-height areas have light blue/green/red backgrounds and black/gray/none outside edges for interrupt request (IRQ), scheduler and system call, and page fault, respectively. The kernel foreground has stripes in two colors, allowing about 250 combinations to be distinguished by interrupt/syscall/fault number. User-mode half-height areas also have stripes in two colors, allowing programs to be distinguished by process ID number. The colors of the horizontal stripe pairs have no particular meaning other than being distinctive for each system or process number. The vertical widths and positions of the stripes have no particular meaning; just their left-to-right length is meaningful, showing elapsed time.

These diagrams show all that is happening on a single CPU core. Every nanosecond is covered, with nothing missing. The concept of "nothing missing" observations is vital. If portions are missing, you cannot make strong statements about what is happening and, in particular, cannot make strong statements about what is *not* happening.

Here is helloworld.c,[6] embellished with two extra `kutrace::mark_a` statements to mark the execution instant of the first line of the main program and of the last line:

```
int main(int argc, const char** argv) {
  kutrace::mark_a("Hello");
  printf("hello world\n");
  kutrace::mark_a("/Hello");
  return 0;
}
```

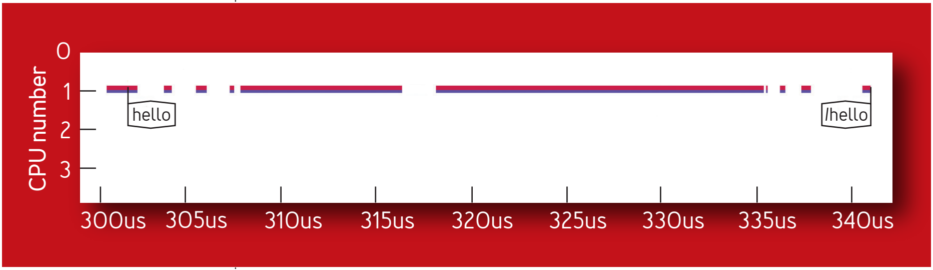
What can be observed about the execution of this simplest program, using various kinds of tools? Draw a little picture in your head of how figure 3 looks for helloworld and then continue reading.

Drilling down through successive levels of observation detail will illustrate what is likely missing from the picture in your head. The diagrams show helloworld running under the Linux operating system on a four-core x86 processor. Running under Windows or on a different processor would give similar results.

Observing user-mode execution of a single program

Many of today's observation tools can observe something about the user-mode code in a single program. Figure 4 is a little picture of the main program's user-mode code in helloworld. It is running on CPU #1, and its half-height user-mode rectangles are drawn with a pair of colors based on the PID (process ID). The elapsed time is about 40 microseconds. The boxed labels `Hello` and `/Hello` mark the execution of the first and last lines of the main program, generated by the `kutrace::mark_a` statements.

FIGURE 4: **THE HELLOWORLD MAIN PROGRAM USER-MODE ACTIVITY**



A CPU profiling tool such as gprof,[4] Visual Studio Profiling Tools,[8] or VTune[7] will provide a breakdown of main-program CPU time by subroutine name or perhaps by line number or call stack, but without any time sequencing. Of course, for helloworld, with just one line of code, this is fairly uninteresting, but profiles are invaluable for identifying or summarizing where time is spent in batch programs. They are not as useful for understanding distributed transactions.

Some programs have multiple threads, but many observation tools can observe only the first thread, or they might observe all threads but merge the results into a pretend single thread. Both of these designs hide information—they leave you a victim of the so-called streetlight effect[9], looking under the lamppost where the light is instead of looking where your lost keys are.

Looking carefully at the diagram in figure 4, you notice a little bit of user-mode execution before the Hello marker of the first line of the main program. You also notice some gaps—white space showing that time marches on but no user-mode code is executing (nor is the idle loop executing). The picture in your head usually skips over this kind of detail.

User-mode tools show any time in these gaps as belonging to the first following user-mode piece of code. Thus, whatever is in the white gaps is charged to some convenient nearby piece of user code, distorting the user-code analysis and hiding the gaps. The gaps represent 10 microseconds of the 40 microseconds shown earlier. Read on.

Observing the entire user-mode execution of a single program

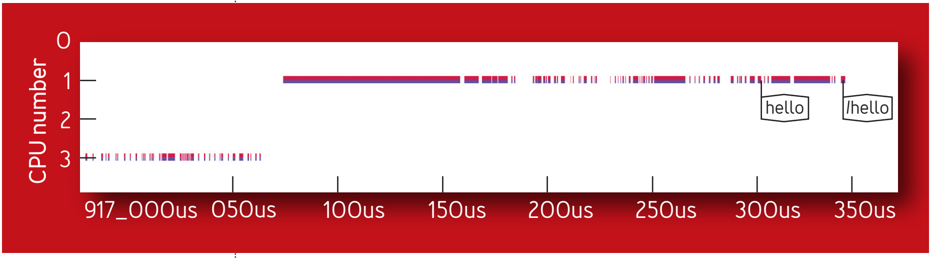
One thing missing from figure 4 is the execution time spent before the first line of `main()` and after the last line of `main()`. You might think this doesn't matter.

When I worked at Adobe, we had a mystery in the 8- to 10-second startup time of an early version of Acrobat. User-mode main-program tools did not reveal the problem. *They were blind to it.* We eventually built an in-house tool with coverage of *all* the Acrobat code. It immediately revealed the hidden time spent in C++ constructors, many of which were run before the first line of `main()`, and these were taking nearly seven seconds. Moving the work in these constructors to a separate nonblocking thread dramatically improved the application startup time. The multimonth mystery was solved by the very first output of a tool that could actually observe the problem. The fix took about 20 minutes.

Remember, if you can't see it, you can't fix it.

Figure 5 shows the entire user-mode execution of helloworld, including all the startup and shutdown time.

FIGURE 5: **ALL OF THE HELLOWORLD USER-MODE ACTIVITY**



This superset of figure 4 shows all of the helloworld user-mode activity on two of four CPUs, including the time before and after `main()`. The elapsed time is about 400 microseconds. The boxed labels `Hello` and `/Hello` mark the execution of the first and last lines of the main program, as in figure 4.

This figure reveals that the helloworld process actually started execution on CPU #3 and was then migrated by the operating system to CPU #1 (it does not reveal *why*). What is all that CPU time doing? It is time spent loading the program and initializing it. We rarely think about where a program comes from when we run it. In this case, it came from the in-RAM file-system cache (because I had run the program before; the elapsed time is too short to have come from disk; running with an empty file-system cache adds another ~11 milliseconds on the front waiting for the disk activity).

There is more CPU time than sometimes expected in parsing the .exe image, relocating the instructions of the image, allocating and initializing the runtime stack, getting stdin/stderr/stdout open and available, etc. There is also some time spent after the main program exits. We will get to that in a moment.

Perhaps you have encountered data-center programs or command-line scripts (Perl, Python, JavaScript) that run every five minutes. Or perhaps in a database environment each query is parsed and optimized before accessing the storage devices. Spending several seconds at startup or shutdown is unimportant for long executions but matters for short ones. Three seconds of startup every five minutes is one percent of the total time. Ten such programs/scripts could consume 10 percent of the total time. A distinguishing characteristic of startup time is that it is serialized with respect to the eventual real work, so it undermines any subsequent parallelization. Christina Delimitrou and Christos Kozyrakis[3] explore the relationship between serialization and tail latency at several levels.

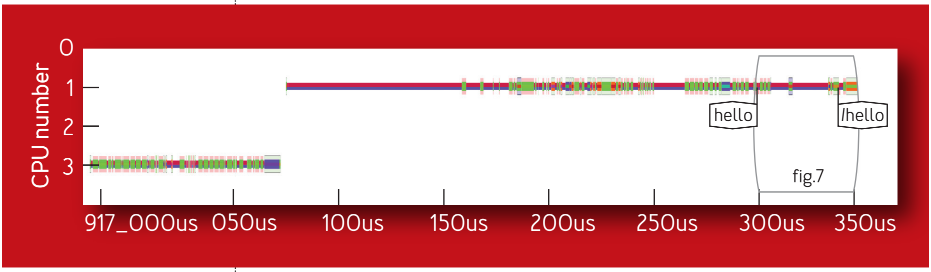
Observing the entire kernel- plus user-mode execution of a single program

Now, about those gaps. They are time spent in kernel-mode execution (figure 6). This is the same 400 microseconds of figure 5, but with the kernel-mode activity explicitly shown. Most of the time is spent handling page faults. The kernel time is shown in full-height rectangles.

The tall pink-background rectangles are page-fault handling, the greenish ones are system calls, and the two with bluish backgrounds (~180 microseconds, ~320 microseconds) are interrupts. There is more time in kernel code than in user code.

It is less common for performance-analysis tools to

FIGURE 6: **KERNEL-MODE ACTIVITY SHOWN EXPLICITLY**

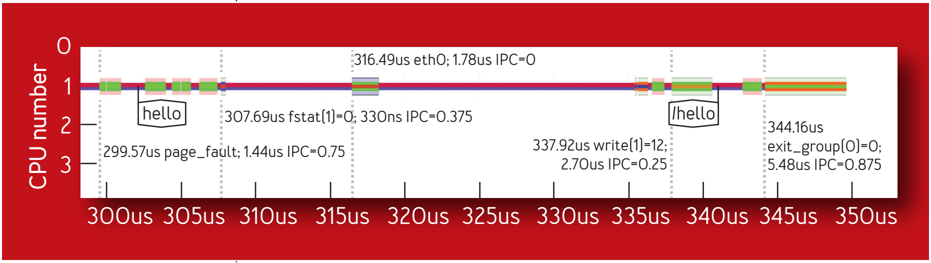


show details of kernel time. *Most are blind to it.* Now that you can see them, you might ask, What is causing all those page faults? It turns out that program loading typically maps an executable disk image into virtual memory and then uses page faults to transfer pages of that image off disk into physical memory. (If the image is already in RAM in a file-system cache, these transfers just copy from memory to memory, but still take up a significant amount of time.) The time hit of these page faults might occur substantially later—when an infrequent part of the program is first used—taking unexpected milliseconds of disk-access time and perhaps hurting some transaction's latency.

There are also page faults closer to the first line of `main()`. The program's runtime heap is set up by a `brk()` system call that extends the address space just above the loaded program. A common implementation of this builds page tables for all the newly allocated heap pages but points them all to a single kernel all-zero page and marks them copy-on-write. This saves the time of zeroing all the new heap pages or, more precisely, defers that time. Whenever a heap page is first written to, a page fault occurs; the kernel allocates a real physical page and copies all the zeros to it, marks the page read-write, and returns.

Figure 7 zooms in a bit on `main()` to see individual time spans and to fill in the gaps of figure 4. This is figure 6 zoomed in on helloworld's user-mode plus kernel-mode activity. The elapsed time is about 50 microseconds. At the far left are four page faults, then a short 330-nanosecond call to `fstat()`, then an Ethernet interrupt, a pair of unlabeled calls to `brk()`, a page fault, a call to `write()` of

FIGURE 7: **DETAIL OF MAIN PROGRAM FROM FIGURE 6**



the 12 bytes "`hello world<cr>`", another page fault, and then a system call to `exit_group()`.

The first four page faults are the tail end of allocating the heap. The `fstat` call is not in the C source but is buried in the C runtime library. It is checking that `stdout` is open before writing to it. What are the calls to `brk()` at 336 microseconds and the subsequent page fault? Printf allocates something on the heap, so the heap needs extending within the C runtime library. Few tools reveal possible page faults and zeroing after every `malloc()`, including those in system libraries.

At the far right, the call to `exit_group` takes 5.5 microseconds to wind down the program, a relatively short shutdown sequence.

But wait! There is more.

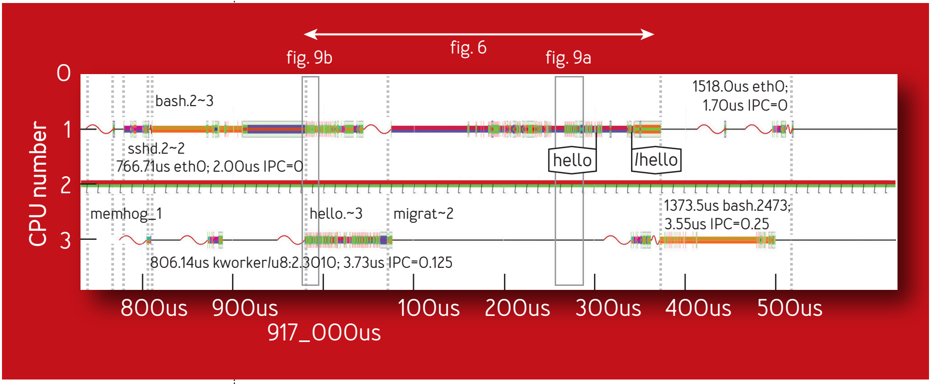
Observing the entire kernel- plus user-mode execution of a single program, plus other programs

So far, we have looked at all of helloworld but are still blind to what else is happening on the entire computer. A couple of unrelated interrupts have taken CPU time away, but recall that laggard data-center transactions can be caused by interference from other transactions or other programs. Tools that can observe a single program are blind to the existence and effects of other programs.

What else is running on the same server? Figure 8 shows all the programs running on this little four-CPU server: helloworld plus all the other programs running. CPU #0 is completely idle and CPU #2 is running a CPU-bound batch program; other programs run on CPUs #1 and #3. The elapsed time is about 800 microseconds. Figure 6 is drawn from the middle 400 microseconds. This is the "nothing missing" picture showing every nanosecond of every CPU core.

The helloworld program was actually run via ssh (secure shell) from a remote machine. At the top left on CPU #1 is an Ethernet interrupt bringing the incoming command line `./helloworld`, followed by the sshd (ssh daemon)

FIGURE 8: **HELLOWORLD PLUS ALL THE OTHER PROGRAMS RUNNING**



program forwarding it to the command-line processor bash.2 via a kernel thread kworker0 on CPU #3. This instance of the Linux bash shell clones itself starting at about time 900 microseconds on CPU #1, creating hello on CPU #3. The migration program moves this to CPU #1. (This is the missing reason mentioned from figure 5; Linux preferentially executes a cloned program on the CPU core that executed the `clone()` system call, when that core is free.) After hello exits, there is some unlabeled sshd activity to transmit the result back, and then bash.2 runs again on CPU #3 to finish shutting down helloworld and send out bash's prompt for the next remote command. During the entire time, an unrelated program memhog_1 is running on CPU #2, beating that core's first-level data cache to death and marking every four iterations of its outer loop. CPU #0 happens to be idle the entire time shown but is in fact taking timer interrupts outside the figure.

Only with tools that can observe time sequencing can you see cause and effect. Only with a nothing-missing picture can you notice that periodic background tasks cause periodic slowdowns of user-facing transactions. Only with tools that can observe *everything* running on a server can you hope to understand sources of serialization and interference. Finally, only with fine-grained microsecond-scale observations can you see the dynamics of this interference.

In the example, a lot of serialized interaction occurs between the sshd/kworker/bash/helloworld programs, each waiting for work from another, but without much cross-program interference to see. There is some interference, nonetheless. In a large-scale distributed transaction system, serialized interactions and interference are prevalent.

Take a look at the red sine waves. Those are synthetic indications of how long it takes this particular x86 CPU to come out of its power-saving deep sleep C6 state—about 30 microseconds. The effect is that it can add 30 microseconds to the time it takes to wake up a sleeping task. If some complex software progresses by passing work successively among dozens of threads, each sleeping while waiting for work, then the additional delay adds up. It can mean transactions are slower on idle servers than on reasonably loaded ones that don't sleep often. It also can ruin the response time of RDMA (remote direct memory access) protocols that depend on microsecond-scale responses to network interrupts.

One Ethernet interrupt slowed down (took CPU time away from) helloworld at time 316.5 microseconds in figure 7. What is the effect on a web-server program that happens to run on a CPU core that handles the bulk of disk interrupts and the disks are highly busy? Or runs on a CPU core that handles the bulk of network interrupts and the network is highly busy? That program silently suffers in a way not observed by many tools.

I once worked on a disk-server box at Google and undersized the CPU's computation ability by 25 percent. To size the system, I used measurements from an in-house accounting tool that charged for total CPU time spent in various user-mode programs, as well as kernel-mode code run on their behalf. I was sadly unaware that the accounting tool did not know which program to charge

Never Too Small to be Ignored

In complex software, an assumption that the time involved is small and can be ignored is death. I worked for a while on Gmail, in a normal development environment with offline load tests used to run a bunch of fake mail transactions and measure whether the current release was faster or slower than before. Of course, rare tertiary transactions were not in the load tests because the time involved was "small and could be ignored"—until we found a user with 100,000 mail messages and an offline program that used the tertiary IMAP (Internet Message Access Protocol) interface to access them and that program had an N**2 bug in the number of mail messages and that caused each access to trigger a Gmail bug that took 20 minutes of server CPU time each time the user logged in. It took much too long to find the one user and the untested Gmail path and then fix it, and add IMAP transactions to the load tests.

for interrupt processing of incoming network packets. Outbound packets were charged to the sender, but for incoming ones the receiver was not known until after they were processed. The accounting program just threw away the incoming network interrupt time, a fundamental design flaw. It should have simply had a category for "incoming network" or even "other" and been designed so that all time was assigned to *some* category, "nothing missing." The assumption, of course, was that the time involved was small and could be ignored. (See sidebar.)

The unseen reality was that one entire CPU core was consumed by interrupt handling of heavy network traffic. This didn't matter much on a 32-core compute server, but it made a 25 percent difference on a little four-core disk server. (The project recovered only because the disk-server software folks spent two painful months making that software 30 percent faster.)

Now take a look at the idle time in figure 8. More specifically, look at all the transitions out of idle. What are these? Each transition out of idle fundamentally represents some program that is waiting for some event; that event finally happens, so the program becomes runnable again. The key information is what happens (probably on a different CPU) just before a core comes out of idle. The activity there—disk interrupt, timer interrupt, freeing a contended software lock, and so on—indicates what the program was waiting for. Within a single server, some laggard transaction delays are not from CPU interference *per se*, but from waiting on other programs or other threads of the same program.

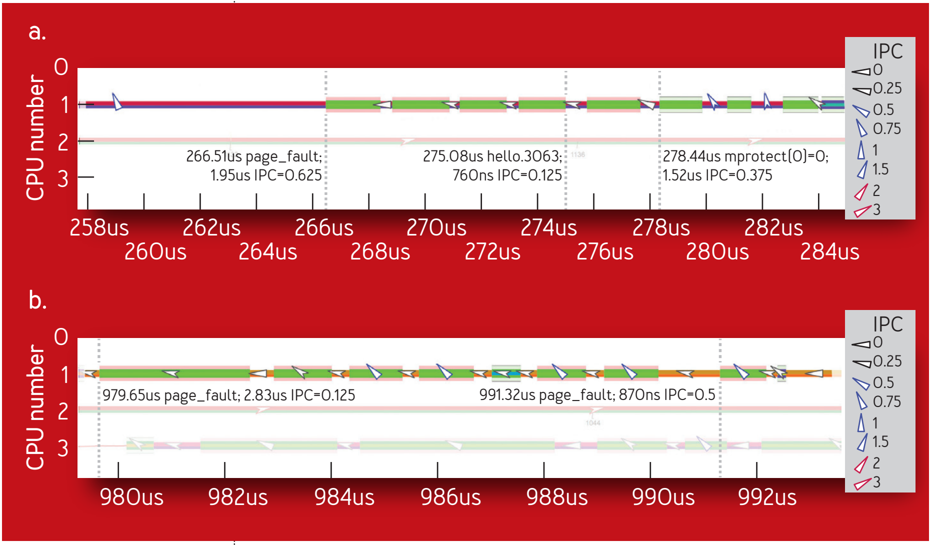
In this environment, you want broad-brush tools that can identify the *reason* for each wait, not just that so many seconds were spent waiting for unknown events. Then you can see the unexpected program dynamics (event A causes event B) that are causing much too much waiting. Perhaps a burst of 163 transactions that are supposed to be done in parallel across 16 cores are, in fact, running sequentially on a single core, making the "last guy" wait over 10 times longer than the picture in the programmer's head. Real event in my former life.

Observing Cross-Domain and Cross-CPU interference

There are times, however, when the CPU-bound portion of some transaction is executing but processing unusually slowly.

Figures 9a and 9b zoom in on some of the page faults, but in addition they show the IPC (instructions per cycle) spent in each time span. This is not some coarse average over several seconds; it is a fine-grained measurement for each and every microsecond-scale time span shown—just the sort of measurement advocated a dozen years ago.[11] At its peak, our chip can execute four IPCs. A low IPC means

FIGURE 9: **PAGE FAULTS AND OTHER ACTIVITY FROM FIGURE 8**



that most of the execution slots are wasted. For the first page fault in figure 9a, labeled `266.51us page_fault; 1.95us IPC=0.625`, the time span is 1.95 microseconds, or about 6,000 cycles on this particular machine. During this time, CPU #1 executed about 3,700 instructions, so 3,700/6,000 = 0.625 IPC. There is a wealth of information here about interference. Some page faults and other activity from figure 8 with IPC are also shown. User-mode code slows down because of kernel-mode cache interference and then recovers.

In figure 9a the angle of the little triangle on the helloworld user code on CPU #1 at about 259 microseconds indicates that hello executed about 0.75 IPC. Now look at the little piece of hello just after the first page fault. Its IPC dropped below 0.125. In fact, hello runs slowly with low IPCs after each of the five page faults, then it picks up again on the right after the `mprotect` calls. You are looking directly at the speed consequence of page-fault kernel code trashing the caches that helloworld is using.

Figure 9b shows the opposite effect. Kernel-mode code starts slowly with a cold cache because of user-mode cache interference and then speeds up. Look at the low IPC for the first page fault on CPU #1 and then look at the page-fault code's IPC rising on subsequent page faults. It is getting faster because more of its stuff is in the caches; just as in figure 9a, the user program is getting slower as less of its stuff is in the caches. Also note that the IPC in the page-fault handler is much higher than in the user code over this small set of time intervals. Conventional wisdom is that kernel code always executes with a cold cache, so it is inherently slow. That is seen to be false when you can actually observe the true CPU behavior.

A Note About Observation Overhead

A tool that gives you fine-grained observations but slows down the programs being observed by 20 times is fine for understanding SPEC (Standard Performance Evaluation Corporation) benchmarks and other batch programs. It is useless in a production data center. Even a seven percent slowdown, such as caused by tcpdump, is unacceptable in a live user-facing data center during the busiest hour of the day. The more interesting tail-latency problems, of course, occur only in a live user-facing data center during the busiest hour of the day. What to do?

Only fine-grained observation tools with one percent or less CPU overhead are acceptable for live user traffic. This means that tool designers and tool users must measure each tool's overhead and insist on it being tiny. I generated the diagrams in this article with a software tool[13] carefully engineered to have ¼ percent overhead when tracing 200,000 kernel-user transition events per second per CPU core, and less than one percent overhead when also tracking IPC (because of the incredibly slow read of the x86 instructions-retired performance counter, and then the slow divide).

The table in figure 10 gives a short list of CPU observation tools (other tools can observe disk and network traffic, not discussed here). These performance observation tools are all findable via a web search for "<name> linux" or "<name> windows". Lowercase names are Linux tools, capitalized names are Windows tools.

Sampling tools periodically examine the CPU program counter to give an overview of top CPU resource consumers by subroutine or line number; they typically require recompilation from source code to use effectively. Counting tools periodically examine various hardware and software counters, either for a single program or for all programs running on a server. Because they execute infrequently, sampling and counting tools are fairly low overhead and therefore could be run on live data-center machines. Tracing tools record time-sequenced events, either for a single program or for all programs running on a server. They typically slow down execution by 30 times or so, making them completely unsuitable for live data-center machines. KUTrace, the tool used to produce figures 3-9, is

FIGURE 10: **A SMALL SAMPLE OF PERFORMANCE OBSERVATION TOOLS**

TOOL	TYPE	ONE PROG. MAIN() USER-MODE	ANY EXE ANY LANG NO RE-COMPILE	ONE PROG. ENTIRE USER-MODE	KERNEL-MODE SYSCALL	KERNEL-MODE FAULT	KERNEL-MODE IRQ	ALL SERVER PROG.	LOW OVER-HEAD
gprof profiler VTune profiler Visual Studio profiler	sample	★							★
strace Process Monitor	trace	★	★	★	★				
Visual Studio	count	★	★	★	★	★	★		★
perf Performance Monitor Task Manager	count	★	★	★	★	★	★	★	★
ftrace	trace	★	★	★	★	★	★	★	
kutrace	trace	★	★	★	★	★	★	★	★

the exception with less than one percent overhead. Tracers are the only tools that can directly reveal cross-thread and cross-program interference. Only full traces have the "nothing missing" property.

CONCLUSIONS

There is a lot to be learned from careful observation of a program and its *complete* surrounding context, even one as "trivial" as helloworld. Some or all of these were probably missing from the picture in your head:

- User-mode execution gaps

- Startup/shutdown time
- Page-fault time, interrupt time, system-call time
- More time in kernel code than in user code
- Late page-in of rarely used code
- `fstat()`, `malloc()` from within C runtime library
- Possible page faults and zeroing after every `malloc()`
- Resuming from power-saving sleep states
- Serialized waiting for others; not executing at all
- Cache interference

If you substitute for helloworld a nontrivial data-center program, you can learn even more about that program and its interactions with interfering processes on a single server. Finding and tracking down one such interference effect at Google paid for 10 years of my salary, in terms of money saved after a 20-minute fix.

As more and more software moves off the desktop and into data centers, and more and more cell phones use server requests as the other half of apps, observation tools for large-scale distributed transaction systems are not keeping up. This makes it tempting to look under the lamppost using simpler tools. You will waste a lot

Related articles

Network Applications Are Interactive
The network era requires new models, with interactions instead of algorithms.
Antony Alappatt
https://queue.acm.org/detail.cfm?id=3145628

Sir, Please Step Away from the ASR-33!
To move forward with programming languages we need to break free from the tyranny of ASCII.
Poul-Henning Kamp
https://queue.acm.org/detail.cfm?id=1871406

Simplicity Betrayed
Emulating a video system shows how even a simple interface can be more complex—and capable—than it appears.
George Phillips
https://queue.acm.org/detail.cfm?id=1755886

of high-pressure time following that path when you have a sudden complex performance crisis.

Instead, know what each tool you use is blind to, know what information you need to understand a performance problem, and then look for tools that can actually observe that information directly—and do so with low enough overhead and distortion to remain useful.

You may now resume your skepticism.

Acknowledgments

My thanks to Lance Berc, V. Bruce Hunt, Amer Diwan, Mark Hill, Michael Brown, and the ACM referees for materially improving the draft of this article.

References

1. Cantrill, B. 2006. Hidden in plain sight; *acmqueue* 4 (1), 26-36.
2. Dean, J., Barroso, L. A. 2013. The tail at scale. *Communications of the ACM* 56 (2), 74-80.
3. Delimitrou, C., Kozyrakis, C. 2018. Amdahl's law for tail latency. *Communications of the ACM* 61 (8), 65-72.
4. Graham, S. L., et al. 1982. gprof: a call graph execution profiler. Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices 17 (6), 120-126; https://docs.freebsd.org/44doc/psd/18.gprof/paper.pdf.
5. Gregg, B. 2012. Thinking methodically about performance; *acmqueue* 10 (12).
6. Kernighan, B. W. 1974. Programming in C: a tutorial; https://www.lysator.liu.se/c/bwk-tutor.html.
7. Intel. 2018. Modern processor performance analysis.

Intel Developer Zone; https://software.intel.com/en-us/intel-vtune-amplifier-xe.

8. Microsoft. 2015. Beginners guide to performance profiling. Microsoft Developer Network; https://msdn.microsoft.com/en-us/library/ms182372.aspx.

9 Norton, R. A. 2010 Streetlight Effect; https://en.wikipedia.org/wiki/Streetlight_effect.

10. Ousterhout, J. 2018. Always measure one level deeper. *Communications of the ACM* 61 (7), 74-83

11. Purdy, M. 2006. Modern performance monitoring. *acmqueue* 4 (1), 48-57.

12. Sigelman, B. H., et.al. 2010. Dapper, a large-scale distributed systems tracing infrastructure. Google AI; http://research.google.com/pubs/pub36356.html.

13. Sites, R. 2017. KUTrace: where have all the nanoseconds gone? Tracing Summit, Prague, Czech Republic; https://tracingsummit.org/w/images/3/30/TS17-kutrace.pdf.

Dr. Richard L. Sites *wrote his first computer program in 1959 and has spent most of his career at the boundary between hardware and software, with a particular interest in CPU/software performance. He was head of a VAX microcode team at Digital Equipment Corporation, and then with Rich Witek, co-architect of the DEC Alpha processors. With Michael Uhler, he invented the performance counters found in nearly all processors today. With Ben Sigelman, et al., he helped design and has heavily used the Dapper RPC tracing tool. With Ross Biro, he built the first version of KUTrace at Google. He has done low-overhead microcode and software tracing at DEC, Adobe, Google, and more recently consulting at Tesla. He*

studied at MIT, University of North Carolina, and Stanford University. He holds 35 patents and is a member of the National Academy of Engineering.