To Catch a Failure

The Record-and-Replay Approach to Debugging

case study

**A DISCUSSION
WITH ROBERT
O'CALLAHAN,
KYLE HUEY,
DEVON O'DELL,
AND
TERRY COATTA**

When work began at Mozilla on the record-and-replay debugging tool called *rr*, the goal was to produce a practical, cost-effective, resource-efficient means for capturing low-frequency nondeterministic test failures in the Firefox browser. Much of the engineering effort that followed was invested in making sure the tool could actually deliver on this promise with a minimum of overhead.

What was not anticipated, though, was that *rr* would come to be widely used outside of Mozilla—and not just for sleuthing out elusive failures, but also for regular debugging.

Developers Robert O'Callahan and Kyle Huey recount some of the more interesting challenges they faced in creating and extending *rr*, while also speculating about why it has experienced a surge in popularity more recently. O'Callahan, a Mozilla Distinguished Engineer, led the *rr* development effort. Huey also worked on *rr* while at Mozilla. Both have since left to create their own company, Pernosco, where their focus has turned to developing new debugging tools.

Helping to steer the discussion are two other engineers of note: Devon O'Dell, a senior systems engineer at Google who has never made a secret of his keen interest in debugging, and Terry Coatta, the CTO of Marine Learning

Systems.

DEVON O'DELL What frustrations with other debuggers spurred you to start working on rr?

ROBERT O'CALLAHAN The original motivation was that Mozilla was running lots of tests—as organizations do—and many of them failed nondeterministically, which made these things really difficult to debug. When you're running tests at large scale—say, thousands or millions of tests on each commit—low failure rates like these can get to be pretty annoying.

You can just disable those tests, of course. But actually *fixing* them, especially since many correspond to underlying product bugs, seems far more appealing. So, we thought about how to do that and concluded that a tool able to record a test failure and then replay it as often as necessary would be just the thing—if we could build it, that is—since that would really lower the risk. Obviously, if a test needs to be run thousands of times to catch a failure, you would like for that to be automated. You would also want to be able to debug that failure *reliably*. That's why the basic idea we started out with was: How can we record tests with the lowest overhead possible and then have the ability to replay the execution as often as needed?

TERRY COATTA The simplistic solution here, of course, would be: “Hey, no problem. I just record *everything* this processor does.” So, just for clarity, what constitutes “low overhead” in the world of debuggers?

RO Good point. From the user's perspective, it just means the code doesn't appear to run any slower when you're recording than when you aren't. Achieving that, obviously, requires some extra work. I should add, for comparison's

sake, that many tools in this space would slow a program by a factor of 3, 5, 100—or even as much as 1,000—depending on the technology. Our goal was to get to less than a 2x slowdown.

It's certainly the case that the technology you pick is going to have a big influence on the amount of overhead. As you just mentioned, if you were to instrument so you could look at everything the CPU does, that obviously would require a lot of work.

DO What would you say it is about the rr approach that lets you achieve the execution speed you were looking for? How does that vary from other debuggers?

RO The fundamental idea is one that several systems share: If you can assume the CPU is deterministic and you're able to record the inputs to your system—say, to your virtual-machine guest or to some process—and you're also able to reproduce those inputs perfectly during replay, then the CPU will do the same thing. Hopefully, this means you can avoid actually having to monitor what the CPU does. That's the idea, anyway. And it isn't a new idea, either.

So, we record only what crosses the process boundary—for example, system calls and signals. Crossing the process boundary isn't so frequent since that will cost you time just by virtue of crossing a protection domain.

KYLE HUEY We do record each injection of nondeterministic state. So, any time there's a syscall or some sort of asynchronous signal, we record that. But while we have to record all these things, we don't create any of our own.

RO So, here's the deal: Record-and-replay tends to be all or

nothing. That means you basically need to catch all these different forms of nondeterminism. If you miss something, the behavior you'll observe when you replay is probably going to diverge from what actually happened, which means you're going to end up in a very different state since programs are very chaotic. Which is to say, you really need to make sure you've nailed down all those sources of nondeterminism and have recorded every single one.

TC What do you require from the hardware so as to collect all the necessary information?

RO First, there's that big assumption we've already talked about, which is that the CPUs are deterministic such that, whenever you run the same sequence of instructions from the same starting state, you ought to end up with the same results, the same control flow, and everything else. That's critical, and, obviously, there are some instructions that don't meet these requirements—something that generates random numbers, for example. That clearly would not be deterministic.

This means we need a way to trap any nondeterministic behaviors or instructions, or at least have some means for telling programs they should avoid these things. For example, with X86 architectures, there's a CPUID instruction you need to watch out for. So, for most modern CPUs and kernels, we did some work to ensure there's an API you can use to say, "Hey, on all CPUID instructions in this process, you need to trap and make sure they don't use RDRAND. Do your own thing instead." The same goes for hardware transactional memory, which is something modern Intel CPUs provide for.

The problem is, while the transactions themselves are

OK, they can sometimes fail for spurious reasons. Which is to say, you might choose to run a transaction, and, behold, it works! It doesn't abort! But if you run it again, it's just possible the transaction *will* abort, owing to some internal cache state or something else in the environment you can't see or control. That effectively means we need to tell our programs to avoid using transactional memory.

Another thing we rely upon—and this is really important—is having some way to measure progress through a program. That's so we can deliver asynchronous events, like signals or context switches, at exactly the same point during replay as they came up during recording. If the CPUs are deterministic and you execute one million instructions starting from some particular state while recording, you should end up in the same state in the replay. But if you execute one million and three instructions in the replay, you're probably going to end up in some different state, and that would be a problem.

This is just to say that we need a way to count the number of instructions executed, ideally such that we can then, during replay, stop the program once that many instructions have been executed. This happens to be pretty much what hardware performance counters allow you to do, which means we really depend upon hardware performance counters to make this all work. And that, as it turns out, is probably also the key to getting rr to work with low overhead. Hardware performance counters are basically free to use—especially if you're just counting, rather than interrupting your program to sample.

While it's very fortunate for us that these counters are available, we also depend on them being absolutely

reliable—and that’s the hard part. Basically, you need to make sure the count of events is the same each and every time you execute a particular instruction sequence. This also means, if your counter happens to be one that counts the number of instructions retired, it needs to report the same number of instructions retired as were actually executed. This can be quite a challenge since many people use performance counters just to measure performance, in which case this property isn’t really essential. If measuring performance is your use case and the counter is off by just a few instructions, it’s no big deal. But we, on the other hand, actually do care about this. The brutal truth is that a lot of these counters don’t measure exactly what they say they do. Instead, they deliver spurious overcounting or spurious undercounting.

That’s a big problem for us. So, we had to look for a reliable counter we could use with Intel devices, which were our initial targets. Ultimately, we found one that’s reliable enough—a conditional branch counter, actually. That’s what we use now to count the number of retired conditional branches, and we’ve found it actually does exactly what the manufacturer says it does. And that’s really what makes `rr` possible. Had we found there were no counters that were accurate in this way, this project simply would not have been feasible.

TC Does Intel *claim* its counters are accurate? Or does it offer no assurances at all?

RO That’s a difficult question to answer. What I can tell you is that many of the cases where its counters deliver slightly erroneous values are documented as errata in the product datasheet. So, I believe there are people at Intel

who care—perhaps not enough to fix the issue, but at least enough to divulge the problems.

TC Are you concerned that the one precious performance counter you know you actually can rely upon might suddenly go nondeterministic on you?

RO Absolutely, and I'm hopeful that articles in leading computer science publications might serve to draw attention to this very concern.

DO You earlier mentioned the assumption of determinism you make in light of the obvious *lack* of determinism you find with execution orderings in multithreaded programs. It's my understanding `rr` runs everything in a single thread more or less to counter this, but I imagine there's probably much more to it than that.

RO To be clear, I should point out that `rr` runs everything on a single core. That's very important. We context-switch all those threads onto a single core. By doing so, we avoid data races since threads aren't able to touch shared memory concurrently. We control scheduling that way and so are able to record the scheduling decisions, using the performance counter to measure progress. In this way, context switching becomes deterministic during replay, and we no longer have to worry about data races.

DO Does this also help you with issues such as memory reordering?

RO Yes. If we're talking about weak memory models, you ensure everything will be sequentially consistent by forcing all of it onto a single core. There is, therefore, a class of memory-ordering bugs that cannot be observed under `rr`.

TC You've mentioned that you require the hardware to be capable of deterministically measuring progress in certain ways. But it also sounds as though there's a whole raft of constraints you face here. For example, it sounds like you need to be able to interface with the scheduler in order to understand or map all the threads down to a single core.

RO We don't need to integrate with the operating-system scheduler. Actually, we use the `ptrace` API to accomplish high-level control of what's going on here. That's a complicated API with a number of different features, but one of the things you can do with it is to start and stop individual threads. We use it to stop all the threads and then start just the ones we want to run. After that, whenever our scheduler decides, "Hey, it's time for a context switch," we interrupt the running threads as necessary and start a different thread. Then we'll pick another thread and run that using our scheduler. Essentially, the operating-system scheduler isn't left with anything to do. We control everything, and that's good since integrating with the operating-system scheduler is kind of a pain.

Just the same, there's still at least some interaction with the operating-system scheduler in the sense that, whenever a program goes into a system call and the system call blocks, we need to be able to switch threads in our scheduler. That means basically setting aside the thread that was running to wait for an exit system call. In the meantime, we can also start running a different thread. We need an operating-system interface that will tell us when a thread has blocked and basically has been descheduled in the kernel.

Fortunately, there is just such an interface out there. Linux gives us these scheduling events—performance events, actually. There's a genuine Linux performance counter API that provides us with a way to be notified whenever the thread we're running blocks and triggers one of these deschedules. We use that for our event notification.

TC It sounds like you've got this kind of super-detailed integration that leaves you dependent on certain very specific features of the processor. I suppose you rely on some interesting pieces of the operating system as well. As you've already suggested, this can lead to revelations of some behaviors people aren't necessarily expecting. This leads me to wonder if any interesting episodes have come up around some of these revelations.

RO Because of the requirement that the performance counters be perfectly accurate, some kernel changes unexpectedly hurt us. In one recent instance, there was a change that landed in the kernel and then ended up briefly freezing the performance counters until a kernel interrupt could be entered, and that led to a few events being lost.

We seem to be the only ones who really care about this sort of thing. Anyway, we ended up finding that bug, and, hopefully, we'll get it fixed before the kernel is released. That sort of thing tends to happen to us quite a bit.

KH In the same vein, we found a bug in the Xen hypervisor related to a workaround to a bug in the Intel silicon that goes so far back into the dark days that nobody seems to be quite sure which CPU originally introduced it. What happened there was, if something ended up in the performance-monitoring unit's interrupt handler and

was counted as zero, it automatically got reset to “1” since, apparently, if you left it at zero, it would just trigger another interrupt. And that, as you might imagine, is enough nondeterminism to break `rr` since it adds one event to the count. It took us something like a year to find that, which proved to be just no end of fun. And then they never did actually get it fixed.

RO Another aspect of this is that the sheer breadth of complexity with the X86 architectures is pretty staggering. The more instructions there are, the more ways there are to get things wrong. For example, there are a number of special instructions for accessing certain weird registers so you can ensure they have the right values during record-and-replay. And, trust me, this can prove to be very nontrivial. One register lets you know whether your process uses one of the X87 floating-point instructions from the 1980s. For our record-and-replay purposes, we need the value in that register to remain constant. If it also happens to be correct, so much the better. There are lots of weird instructions like that, which can be really eye-opening—and kind of scary.

KH There definitely is a lot of complexity in the X86 space submerged just below the surface.

TC As I listen to you talk about this, I’m thinking to myself, “Oh my God, this sounds like software-developer hell.” You’ve got some super-complicated user software that relies on a whole bunch of other super-complicated submerged bits of software. Does this ever get to you? Have you had to invest superhuman amounts of effort into testing? Basically, how do you manage to cope with so

much complexity?

KH Well, we used to work on web browsers, so this just seems simple by comparison.

RO It's true that the stuff we're interfacing with now is kind of crazy and low level. On the other hand, it doesn't change all that rapidly. It's not as if Intel's architecture revs are coming at some incredible pace—especially not now. The kernel is also evolving pretty slowly at this point.

Because `rr` is purely a user-space tool (which was one of our design goals from the start), we depend on the kernel behavior of a bunch of APIs, but we don't really care about how those APIs are implemented so long as they don't break stuff. The kernel is also open source, so we can always see exactly what's going on there. The hardware, of course, tends to be much more opaque, but it's also more fixed.

Still, I have to admit it takes a lot of effort just to make sure things work consistently across architecture revs. It's also scary to depend on so many things, sometimes in some very subtle ways, and have absolutely no control over those things. But we actually invested a fair amount of energy in talking to people at Intel to nudge them in the right direction or at least discourage them from doing things that are sure to break stuff. For the most part, they've actually been quite helpful.

KH I would add that we have done some amount of testing on the hardware. Of course, what we can do is fairly limited since, by the time we get access to any new Intel silicon, we're already essentially committed to it. Thankfully, there hasn't been a new microarchitecture released in a while, so that hasn't been a huge deal.

We also periodically run our regression test suite

against the current kernel version, and we end up finding things there on a semi-regular basis—maybe once a quarter. Usually it's something relatively benign, but every now and then we'll find something more troubling: Maybe they've just broken the performance counter interface in a way that's problematic for us.

A recurring theme with rr is that one capability just seems to beget another. As an example, once the ability of rr to handle nondeterminism had been clearly demonstrated, it occurred to O'Callahan and Huey that even more nondeterminism might be employed to discover weird edge cases. In this way, rr has come to be used as not only a debugger, but also a tool for improving software reliability.

That is, by combining record-and-replay with the ability to harness nondeterminism, it became apparent that a program could be tested just by unleashing a torrent of nondeterminism on it to see what might come up as a consequence. Any problems bobbing to the surface then could be dealt with proactively, instead of waiting for failures to show up in production.

TC You indicated the original impetus for creating rr was to capture hard-to-reproduce errors that surface during automated testing. But now, is it also being used in some other ways?

RO Actually, the funny thing is, while we designed rr to capture test and automation, it's *mostly* being used now for other things—primarily for ordinary debugging, in fact. On top of record-and-replay, you can simulate reverse

execution by taking checkpoints of the program as it executes forward, and then rolling back to a previous checkpoint and executing forward to any desired state. When you combine that with hardware data watchpoints, you can see where some state in your program isn't correct and then roll back to look at the code responsible for that. Especially with C and C++ code, I think that almost amounts to a programming superpower.

Once people discovered that, they started using it and found there were some other benefits. That's when rr really started catching on. At this point, there's a fair number of people, both at Mozilla and elsewhere, who have taken to using it for basically all their debugging.

One of the more appealing benefits—of both rr and record-and-replay in general—is the clean separation you get between the act of reproducing a bug and the act of debugging it. You run your program, you record it, and—in doing so—you reproduce the bug. Then you can just continue supplying fresh input and debugging what comes just as often as you like. That could lead to a session that lasts a day, or even weeks—if you're up for that. But, with each pass, you'll learn a bit more about what happened in that test case until you've finally got the bug figured out. That means you end up with a workflow that looks a little different from the traditional debugging workflow. People take to that, I think, since it's instinctive.

TC Isn't there also another mode that gives you complete control over thread scheduling? How has that been employed as part of your debugging efforts?

KH Assuming you care about the reliability of your software, once you have a useful tool for dealing with

nondeterminism, one relatively obvious thing to do is to throw as much nondeterminism at it as you can. Basically, that means searching the execution space to find weird edge cases, which is just what Rob has attempted to do with chaos mode.

RO Right. We've already talked about how rr operates on a single core and the sorts of things that can affect program execution there. One issue is that there are certain kinds of bugs we've had trouble reproducing. In fact, we had some Mozilla Firefox test failures that would show up only intermittently—and, in some cases, only on certain platforms. To get to the bottom of that, we'd sometimes run as many as 100,000 test iterations and still not be able to reproduce a failure. That's when we started to think we probably needed to do another pass where we injected some more nondeterminism. The most obvious way to go about that was to randomize the scheduler, or at least randomize the decisions it was making.

That allowed us to study what sorts of schedules, for example, would be required to reproduce some particular bug. It also let us explore why rr wasn't producing those schedules. With a number of iterations like this, we actually managed to implement an improved form of chaos mode.

While we value these benefits of running in chaos mode, the overhead is high. We tried to limit that but couldn't eliminate all of it, so we decided to make chaos mode purely an option. Still, by turning on chaos mode to run your tests, you're likely to find some of your more interesting failures.

Anecdotally, a lot of people have reported they've found chaos mode to be useful for reproducing bugs that

generally are very hard to reproduce. Something else we've discovered about using chaos mode that I consider to be particularly interesting is that many Mozilla tests that fail intermittently actually turn out to fail on only one platform—say, either Android or macOS. Yet when you debug them, you find it's actually a cross-platform bug that shows up on only one platform since it's either particularly slow or has some type of thread scheduler that makes it possible for the bug to be reproduced there. I should add that rr chaos mode often makes it possible to take a bug that was failing only on Android and then reproduce it on desktop Linux. This turns out to be rather useful.

TC Why not always run things in chaos mode in the first place and, thus, surface failures more readily?

RO Chaos mode is clearly recommended if you're trying to reproduce an intermittent failure. But pairing chaos mode with the record-and-replay tool is also advisable. I mean, if you stripped out all the record-and-replay stuff, you'd still be able to reproduce failures with some kind of controlled scheduling. But then what would you do with them?

Work to add new functionality to rr itself now seems to have concluded. But the story of renewal and extension will carry on with efforts to build new tools on top of the existing record-and-replay technology to deliver a “new debugging experience” in which developers can move beyond examining one state in one program space at a time to representing all of the program space within a database such that it can then be queried by a reworked debugger to obtain information across time. A

project to build this new debugger is already underway.

TC Looking back over your effort to build rr, is there anything you would do differently?

RO Yes. But I should first say I think we generally made good design decisions, as most of our bets seem to have paid off pretty well. That includes our choice to focus on a single-core approach to replay even though that decision has been criticized, given that it essentially locks rr out of a large and growing class of highly parallel applications. The main problem is that we still don't know how to even *handle* those applications all that well. I don't think anybody has a clue when it comes to recording highly parallel applications with data races with low overhead using existing hardware and software.

By keeping our focus mostly on single-core applications, we've managed to do a pretty good job with those. It's important to have a tool that works well for a large set of users, even if it doesn't work as well for some other set of users. It's better to narrow your focus and be really good at something than to make the compromises a broader focus might require. I have no regrets about that.

I'm also fine with the decision not to do program-code instrumentation. I continue to be grateful for that every single time Intel releases a thousand new instructions and we don't have to care since we're not faced with having to add all of those to our instrumentation engine.

But since you asked about what we'd do differently... I suppose we'd probably write rr in Rust rather than C++ if we were starting today. Beyond that, though, I'm pretty satisfied with the core design decisions we've made. Kyle, do you see this differently?

KH Not really. The things I'd want to do differently would be possible only in a different universe where saner hardware and saner kernels can be found.

TC What do you see happening with rr as you move forward?

KH We'd like to add support for AMD and ARM architectures, but that's going to require silicon improvements from both of them.

RO We also need to improve our support for GPUs since rr doesn't work when you have sessions that directly access the GPU. To fix that, we'll need to get to where we better understand the interactions between CPU user space and GPU hardware so we can figure out how to get rr to record and replay across that boundary—especially when it comes to recording any GPU hardware effects on CPU user space—if that's even possible.

I think there's room to build alternative implementations of record-and-replay using different approaches, but as for rr itself, it's basically there already. I just don't see adding a large number of new features. Instead, I think the future will involve building on top of the record-and-replay technology, which is something we're already exploring. The basic idea is that, once you can record and replay an execution, you have access to all program states. Traditional debuggers don't really leverage that because they're limited to looking at one state at a time.

The future lies with this new idea called *omniscient debugging* that lets you represent all program space in a database and then rework your debugger such that it can make queries to obtain information across time. In theory, developers ought to be able to use this to obtain results

instantaneously. That's where the next frontier lies in terms of improving the user experience.

A debugger like rr actually is an important stepping stone in this direction since what you really want is the ability to record some test failure with low overhead and then parallelize the analysis by farming it out to many different machines and combining the results. The effect of this would be essentially to deliver a precomputed analysis that enables a faster, more satisfying debugging experience for the developer.

TC A new debugging user experience? Apart from delivering the results faster, what exactly do you have in mind?

RO One of the things you typically do with traditional debuggers is single-stepping, right? Basically, you want to trace out the control flow in functions. So, you step, step, step, step, step. You find yourself staring at a very narrow window that lets you look at the current state, which you then can manipulate through time as you try to build a picture of the control flow. This is something you have to do by manually aggregating the data points.

Instead, what we're looking for is something that lets us observe a function execution and then—assuming we've already recorded and stored the control flow for the entire program—look at a function and immediately see which lines of the function have been executed. Which is to say, you should be able to view the control flow in an intuitive way. This, of course, will require a lot more debugger implementation.

DO It sounds like work on this might already be underway.

RO Yes, we've implemented a lot of this already. It's not out

there yet, but we're working on it.

DO On a related UX [user experience] note, do you foresee some way to make it easier for people to map mental models of protocols to code?

Related articles

➡ Debugging in an Asynchronous World
Hard-to-track bugs can emerge when you can't guarantee sequential execution. The right tools and the right techniques can help.
Michael Donat

<https://queue.acm.org/detail.cfm?id=945134>

➡ Advances and Challenges in Log Analysis
Logs contain a wealth of information for help in managing systems.

Adam Oliner, Archana Ganapathi, and Wei Xu

<https://queue.acm.org/detail.cfm?id=2082137>

➡ Reveling in Constraints

The Google Web Toolkit is an end-run around Web development obstacles.

Bruce Johnson

<https://queue.acm.org/detail.cfm?id=1572457>

RO Much could be done to present dynamic program execution to developers in a more intuitive way. For example, one thing we're doing with our new product is to make it much easier to explore the dynamic execution tree. That way, you can look at a function invocation and request a full list of the functions that are dynamically called from that. That maps pretty nicely to the mental models people have for programs and the ways they work. It also provides a great way to explore a program.

A lot more could be done here. If we were to build information about functions such as `malloc`

and `free` into our debugger, that would also prove really powerful in terms of helping people understand exactly what their programs are doing. I think that would be pretty exciting.

Copyright © 2020 held by owner/author. Publication rights licensed to ACM.