Introducing Research for Practice 76
Ours is a world of information overload. To help navigate through it, this new addition to acmqueue curates a selection of research papers on a given topic. In this first installment, Justine Sherry looks at research on network functions virtualization, and Simon Peter focuses on data-center operating-system trends.
BY PETER BAILIS

The Flame Graph 48
With the help of a flame graph, the output of a software profiler turns into a simple visualization, making it faster and easier to see how resources are being consumed, and thus how performance could be improved.
BY BRENDAN GREGG

Debugging Distributed Systems 91
Distributed systems come with a unique set of challenges for software developers, requiring new approaches to debugging. One such approach is a visualization technique known as ShiViz, an open-source tool that helps address the challenges.
BY IVAN BESCHASTNIKH
PATTY WANG
YURIY BRUN
MICHAEL D. ERNST

Should You Upload or Ship Big Data to the Cloud? 111
The increasing availability of gigabit-speed Internet connections, along with the explosive growth in data-transfer speeds, opens up more possibilities for moving big data into the cloud.
BY SACHIN DATE
ESCAPING THE SINGULARITY
*Standing on Giant Distributed Shoulders*
There are parallels between the challenges of distributed computing and some of the world’s most important physics breakthroughs.
BY PAT HELLAND

EVERYTHING SYSADMIN
*The Small Batches Principle*
Working in small batches rather than taking big leaps leads to faster results, better quality, and less stress.
BY THOMAS A. LIMONCELLI

THE SOFT SIDE OF SOFTWARE
*Nine Things I Didn’t Know I Would Learn Being an Engineer Manager*
You need more than just technical prowess to be a successful dev lead.
BY KATE MATSUDAIRA

KODE VICIOUS
*What Are You Trying to Pull?*
Optimizing away a few instructions isn’t worth it if it leads to a lack of code clarity.
BY GEORGE NEVILLE-NEIL
ACM, the world’s largest educational and scientific computing society, delivers resources that advance computing as a science and profession. ACM provides the computing field’s premier Digital Library and serves its members and the computing profession with leading-edge publications, conferences, and career resources.
If you squint hard enough, many of the challenges of distributed computing appear similar to the work done by the great physicists. Dang, those fellows were smart! Here, we examine some of the most important physics breakthroughs and draw some whimsical parallels to phenomena in the world of computing... just for fun.

NEWTON THOUGHT HE KNEW WHAT TIME IT WAS
Isaac Newton (1642 – 1727) was a brilliant physicist who defined the foundations for classical mechanics, laws of motion, and universal gravitation. He also built the first refracting telescope, developed a theory of color, and much more. He was one bad dude.

Newton saw time as constant and consistent across the universe. Furthermore, he assumed that gravity operated instantaneously without regard to distance. Each object in the universe is exerting gravitational force at all times.

This is very much like what we see in a single computer or in a tightly coupled cluster of computers that perform consistent work in a shared transaction. Transactions have a clearly defined local notion of time. Each transaction sees its work as crisply following a set of transactions. Time marches forward unperturbed by distance.

When I was studying computer science (and Nixon was
president), we thought about only one computer. There was barely any network other than the one connecting terminals to the single computer. Sometimes, a tape would arrive from another computer and we had to figure out how to understand the data on it. We never thought much about time across computers. It would take a few years before we realized our perspective was too narrow.

EINSTEIN HAD MANY WATCHES
In 1905, Albert Einstein (1879 – 1955) proposed the special theory of relativity based on two principles. First, the laws of physics, including time, appear to be the same to all observers. Second, the speed of light is unchanging.

An implication of this theory is that there’s no notion of simultaneity. The notion of simultaneity is relative to the observer, and the march of time is also relative to the observer. Each of these frames of reference is separated by the speed of light as interpreted relative to their speed in space.

This concept has some interesting consequences. The sun might have blown up five minutes ago, and the next three minutes will be lovely. When stuff happens far away, it takes time to find out... potentially a long time.

In computing, you can’t know what’s happening “over there.” Interacting with another system always takes time. You can launch a message, but you always have to wait for the answer to come back to know the result. More and more, latency is becoming the major design point in systems.

The time horizon for knowledge propagation in a
The time horizon for knowledge propagation in a distributed system is unpredictable. This is even worse than in the physical Einstein-based universe. At least with our sun and the speed of light, we know that we can see what’s happening at the sun as of eight minutes ago. In a distributed system, we have a statistical understanding of how our knowledge propagates, but we simply cannot know with certainty. The other server, in its very own time domain, may be incommunicado for a heck of a long time.

Furthermore, in any distributed interaction, a message may or may not be delivered within bounded time. Higher-level applications don’t ever know if the protocol completed. Figure 1 shows how the last message delivery is not guaranteed and the sender never knows what the receiver knows. In any distributed protocol, the sender of the last message can’t tell whether it arrived. That would require another message.

Another problem is that servers and messages live in their very own time spaces. Messages sent and received across multiple servers may have surprising reorderings. Each server and each message lives in its own time, and they may be relative to each other but may offer surprises because they are not coordinated. Some appear slower, and some faster. This is annoying.

In figure 2, as work flows across different times in servers and messages, the time is disconnected and may be slower or faster than expected. In this case, the second message sent by A may arrive after work caused by the first message, traveling through C. These problems can make your head hurt.
in a similar fashion to how it hurts when contemplating twins where one travels close to the speed of light and appears to slow down while the other one stays home and ages.

You can’t do distributed agreement in bounded time. Messages get lost. You can retry them and they’ll probably get through. In a fixed period of time, however, there is a small (perhaps very small) chance they won’t arrive. For any fixed period of time, there’s a chance the partner server will be running *sloooooow* and not get back.

Two-phase commit cannot guarantee agreement in
bounded time. Similarly, Paxos,\textsuperscript{7} Raft,\textsuperscript{8} and the other cool agreement protocols cannot guarantee agreement in a bounded time. These protocols are very likely to reach agreement soon, but there’s no guarantee\textsuperscript{4}. Each lives in its own relative world and doesn’t know what’s happening over there... at least not yet.

According to the CAP Theorem\textsuperscript{15} (standing for consistency, availability, partition tolerance), if you tolerate failures of computers and/or networks, you can have either classic database consistency or database availability. To avoid application challenges, most systems choose consistency over availability.
Two-phase commit is the anti-availability protocol.

From where I stand, Einstein made a lot of sense. I’m not sure how you feel about him.

HUBBLE WAS INCREASINGLY FAR OUT
Edwin Hubble (1889–1953) was an astronomer who discovered that the farther away an object is, the faster it is receding from us. This, in turn, implies the universe is expanding. Basically, everything is getting farther away from everything else.

In computing, we have seen an ever-increasing amount of computation, bandwidth, and memory. It looks like this will continue for a while. Latency is not decreasing too much and is limited by the speed of light. There are no obvious signs that the speed of light will stop being a constraint anytime soon. The number of instruction opportunities lost to waiting while something is fetched is increasing inexorably.

Computing is like Hubble’s universe...
Everything is getting farther away from everything else.

Shared read-only data isn’t the biggest problem. With enough cache, you can pull the stuff you need into the sharing system. Sharing writeable stuff is a disaster. You frequently stall while pulling a cache line with the latest copy from a cohort’s cache. More and more instruction opportunities will be lost while waiting. This will only get worse as time moves on!
Shared memory works great... as long as you don’t share memory.

Either we figure out how to get around that pesky speed-of-light thing, or we’re going to need to work harder on asynchrony and concurrency.

HEISENBERG WASN’T SURE

Werner Heisenberg (1901–1976) defined the uncertainty principle, which states that the more you know about the location of a particle, the less you know about its movement. Basically, you can’t know everything about anything.

In a distributed system you have a gaggle of servers, each of which lives in various states of health, death, or garbage collection. The vast majority of the time you can chat with a server and get a crisp and timely result. Other times you don’t get a prompt answer and it’s tough to know if you should abandon the slacker or wait patiently. Furthermore, you don’t know if the server got the request, did the work, and just hasn’t answered. Any time a request goes to a single system, you don’t know when the request will be delayed.

In some distributed systems, it is essential to have an extremely consistent and fast response time for online users. To accomplish this, multiple requests must be issued, and the completion of a subset of the requests is accepted as happiness.

In a distributed system, you can know where the work is done or you can know when the work is done but you can’t know both.
To know *when* a request is done within a statistical SLA (service-level agreement), you need to accept that you don’t know *where* the work will be done. Retries of the request are the only option to get a timely answer often enough. Hence, the requests had better be idempotent.

**SCHRÖDINGER’S PUT**
Erwin Schrödinger (1887–1961) was a leading physicist of the early 20th century. While he made many substantial contributions to quantum field theory, he is most often remembered for a thought experiment designed to show the challenges of quantum physics.

In quantum physics the theory, the math, and the experimental observations show that pretty much everything remains in multiple states until it interacts with or is observed by the external world. This is known as a superposition of states that collapse when you actually look.

To show that this seems goofy, Schrödinger proposed that this quantum-level uncertainty could map to a macro-level uncertainty. Start by placing a tiny bit of uranium, a Geiger counter, a vial of cyanide and a cat into a steel box. Rig the Geiger counter to use a hammer to break the vial of cyanide if an atom of uranium has decayed. Since the quantum physics of uranium decay show it is both decayed and not decayed until you observe the state, it is clear that the cat is simultaneously both dead and alive. Turns out many contemporary physicists think that it’s not goofy... the cat would be in both states. Go figure!

New distributed systems such as Dynamo store their
data in unpredictable locations. This allows prompt and consistent latencies for **puts** as well as self-managing and self-balancing servers. Typically, the client issues a **put** to each of three servers, and when the cluster is automatically rebalancing, the destination servers may be sloshing data around. The set of servers used as destinations may be slippery. A subsequent **get** may need to try many servers to track down the new value. If a client dies during a **put**, it is possible that no servers received the new value or that only a single server received it. That single server may or may not die before sharing the news. That single server may die, not be around to answer a read, and then later pop back to life resurrecting the missing **put**.

Therefore, a subsequent **get** may find the **put**, or it may not. There is effectively no limit to the number of places it may be hiding. There’s no upper bound on the time taken for the new value to appear. If it does appear, it will be re-replicated to make it stick.

**While not yet observed,**  
**a put does not really exist...**  
**it’s likely to exist but you can’t be sure.**  
**Only after it is seen by a get will the put really exist.**

Furthermore, the failure to observe does not mean the **put** is really missing. It may be lurking in a dead or unresponsive machine. If you see the **put** and force its replication to multiple servers, it remains in existence with very high fidelity. Not seeing it tells you only that it’s likely it’s not there.
CONCLUSION

Wow! There have been lots of brilliant physicists, many of them not mentioned here. Much of their work has shown us the very counterintuitive ways the world works. Year after year, there are new understandings and many surprises.

In our nascent discipline of distributed systems, we would be wise to realize that there are subtleties, surprises, and bizarre uncertainties intrinsic in what we do. Understanding, bounding, and managing the tradeoffs inherent in these systems will be a source of great challenge for years to come. I think it’s a lot of fun!

References

web services. *ACM SIGACT News*, 33(2).


**LOVE IT, HATE IT? LET US KNOW** feedback@queue.acm.org

Pat Helland has been implementing transaction systems, databases, application platforms, distributed systems, fault-tolerant systems, and messaging systems since 1978. For recreation, he occasionally writes technical papers. He currently works at Salesforce.

Copyright © 2016 held by owner/author. Publication rights licensed to ACM.
Nine Things I Didn’t Know I Would Learn
Being an Engineer Manager

KATE MATSUDAIRA

When I moved from being an engineer to being a dev lead, I knew I had a lot to learn. My initial thinking was that I had to be able to do thorough code reviews, design and architect websites, see problems before they happened, and ask insightful technical questions. To me that meant learning the technology and becoming a better engineer. When I actually got into the role (and after doing it almost 15 years), the things I have learned—and that have mattered the most—weren’t those technical details. In fact, many of the skills I have built that made me a good engineer manager weren’t technical at all and, while unexpected lessons, have helped me in many other areas of my life.

What follows are some of these lessons, along with ideas for applying them in your life—whether you are a manager, want to be a manager, or just want to be a better person and employee.

1. Driving consensus
Technical people love to disagree. I’ve found there usually are no definitive answers to a problem. Instead,
there are different paths with different risks, and each solution has its own pros and cons. Being able to get people to agree (without being the dictator telling people what to do) means learning how to get everyone on the same page.

Since meetings with lots of people can be very contentious, one technique that has helped me wrangle those ideas is called multivoting. Multivoting is helpful for narrowing a wide range of ideas down to a few of the most important or appropriate, and it allows every idea to receive consideration.

You can do this by first brainstorming as a team while putting all the ideas on a whiteboard, along with the pros and cons of each. From there you go through a voting process until the group arrives at what it considers to be an appropriate number of ideas for further analysis. Organizational development consultant Ava S. Butler explains the multivoting process in wonderful detail if you would like more information.¹

2. Bringing out ideas (even from quiet people)
One of the challenges of working with introverts and shy people without strong communication skills is that it can be hard to surface their ideas. They tend to be quiet in meetings and keep their ideas (which can be very good!) to themselves. Below are a few techniques I’ve learned that help me bring these people out of their shells:
  - In meetings I call on people or do a round robin so everyone gets a chance to talk. This way, the shy team members are given the floor to speak where they might have otherwise remained silent.
In one-on-ones I have learned to use the power of silence. I ask a question and then refrain from speaking until the person answers—even if it is a minute later. I had to learn to get comfortable with uncomfortable silence, which has been a powerful technique in uncovering what people are thinking.

I often have everyone write their ideas on a Post-it note and put it on the whiteboard during team meetings. This allows everyone’s ideas to receive equal weight, and introverted people are therefore encouraged to share their thoughts.

3. Explaining tech to nontech
When you want to rewrite code that already works, you have to justify the change to management. Much of the time nontechnical people don’t care about the details. Their focus is on results. Therefore, I have learned to look at all my work, and the work my team does, in a business context. For example, does it save time, save money, or generate revenue—and then how do I best communicate that?

I frame my ideas in a context that matters to the specific audience I’m addressing. Using analogy is one technique I’ve found to be quite powerful. Explaining an idea through analogy allows you to consider your audience’s perspective and talk at their level, never above them.

4. Being a good listener
When you manage people you really have to learn to listen. And, by the way, listening goes way beyond paying attention to what is said. You should also be paying attention to body language and behavior.
I like to use the example of an employee who always arrives early to work. If that person suddenly makes a new habit of showing up late, this could be a cue that something is amiss. By listening to that person’s actions, and not just their words, you gain valuable insight and can manage with greater empathy and awareness.

5. Caring about appearance
When you are in a leadership role you often meet with people outside of your immediate co-workers who don’t know you as well. And they judge you. Plus, studies have shown that your appearance strongly influences other people’s perception of your intelligence, authority, trustworthiness, financial success and whether you should be hired or promoted.5

Growing up, I was taught by my grandfather how to dress for the job I wanted, not the job I currently had. As a new manager, I put more of an effort into my appearance, and it definitely had a positive effect, especially when interacting with customers and clients outside of the organization.

I recommend emulating the people in your organization whom you look up to. Look at how they dress. Study how they carry themselves. Watch how they conduct themselves in meetings, parties, and other events. This is where you can get your best ideas for how to dress and communicate success. You want your work and reputation to speak for themselves, but don’t let your appearance get in the way of that.

6. Caring about other disciplines
The more you know about other facets of the business, like
sales and marketing, the more capable you are of making strategic decisions. The higher up you go, the more important this is, because you aren’t just running software—you are running a business.

It’s also vital to understand the needs of your customers. You could build what you believe is an amazing product, but it could end up being useless to the customer if you never took the time to fully understand their needs. Even if you work in back-end development, caring about the end user will make you create better solutions.

7. Being the best technologist doesn’t make you a good leader
If you are managing enough people or products, you don’t have time to dive into the deep details of the technology. Moreover, you need to learn to trust the people on your team. It is better to let them be the experts and shine in meetings than to spend your time looking over their shoulders to know all the details.

The best skills you can have are these:

- Ask great questions that get to the root of the problem. This helps others think through their challenges, uncovering issues before they arise.
- Delegate and defer so that you are able to accomplish more while empowering those around you.
- Teach people to think for themselves. Instead of prescribing answers, ask people what they think you would say or tell them to do. I highly recommend David Marquet’s talk, “Greatness.” He reveals that while working as a captain on a military submarine he vowed never to give...
another order. Instead, he allowed his reports to make their own empowered decisions. This small shift in thinking brought about powerful change.

8. Being organized and having a system

When you are responsible for the work of others, you must have checks and balances. Practicing strong project-management skills is key. You need to have a way of keeping things organized and know what is going on, and be able to communicate it when things aren’t going as planned.

It’s also important to be strategic about your own time management. I start each week with at least 30 minutes dedicated to looking at my top priorities for the week, and then I carve out the time to make progress on these priorities. One time-management tool that has been successful for me is time blocking, where I plan my days in a way that optimizes my time for my productivity [for example, I am a much better writer in the mornings, so I make sure to do my writing then].

This helps me optimize my time and always know the best way to use a spare 15 minutes.

Similarly, I have a system for keeping track of my great ideas. I keep an Evernote where I save articles I love or interesting ideas I come across. This gives me a little vault of information I can go to when I need to get inspired, write a blog post, or come up with something worthwhile to post on social media.

The point here is to have systems in place. You need a way to do all the things that are important and keep your information and details organized.
9. Networking

If you think about it, every job offer, promotion, and raise wasn’t given to you because of the work you did. The quality of your work *may* have been a factor, but there was a person behind those decisions. It was *someone* who gave you those opportunities.

If you do great work and no one likes you, then you simply won’t be as successful. Be someone with whom people want to work. For example, helping others, listening intently, and caring about the lives of the people around you will help you profoundly. I am always looking for ways to expand my network, while also deepening the relationships I have with my mentors and friends.

I hope these ideas help you become a better leader or employee. Pick one or two to focus on each week, and see where it takes you—progress is a process! I would love to hear from you, especially if you have any other ideas to add to this list.

References


3. Marquet, D. 2013. Inno-versity presents:
greatness. YouTube; https://www.youtube.com/watch?v=OqmdLcyES_Q.


LOVE IT, HATE IT? LET US KNOW feedback@queue.acm.org

Kate Matsudaira is an experienced technology leader. She worked in big companies such as Microsoft and Amazon and three successful startups (Decide acquired by eBay, Moz, and Delve Networks acquired by Limelight) before starting her own company, Popforms (https://popforms.com/), which was acquired by Safari Books. Having spent her early career as a software engineer, she is deeply technical and has done leading work on distributed systems, cloud computing, and mobile. She has experience managing entire product teams and research scientists, and has built her own profitable business. She is a published author, keynote speaker, and has been honored with awards such as Seattle’s Top 40 under 40. She sits on the board of acmqueue and maintains a personal blog at katemats.com.

Copyright © 2016 held by owner/author. Publication rights licensed to ACM.
Q: What do DevOps people mean when they talk about small batches?
A: To answer that, let’s take a look at an unpublished chapter from the upcoming book *The Practice of System and Network Administration*, third edition, due out in October 2016.

One of the themes you will see in this book is the small batches principle: it is better to do work in small batches than big leaps. Small batches permit us to deliver results faster, with higher quality and less stress.

We begin with an example that has nothing to do with system administration in order to demonstrate the general idea. Then we focus on three IT-specific examples to show how the method applies and the benefits that follow.

The small batches principle is part of the DevOps methodology. It comes from the lean manufacturing movement, which is often called just-in-time manufacturing. It can be applied to just about any kind of process. It also enables the MVP (minimum viable product) methodology, which involves launching a small version of a service to get early feedback that informs the decisions made later in the project.

**THE CARPENTER ANALOGY**
Imagine a carpenter who needs 50 pieces of two-by-four lumber, all the same length. One could imagine sawing all 50
pieces then measuring them to verify they are all the correct size. It would be very disappointing to discover that the blade shifted while making piece 10, and pieces 11 through 50 are unusable. The carpenter would have to remake 40 pieces.

A better method would be to verify the length after each piece is made. If the blade had shifted, the carpenter would detect the problem soon after it happened, and there would be less waste.

These two approaches demonstrate big batches versus small batches. In the big-batch world the work is done in two large batches: the carpenter cut all the boards, then inspected all the boards. In the small-batch world, there are many iterations of the entire process: cut and inspect, cut and inspect, cut and inspect,...

The first benefit of the small-batch approach is less waste. Because an error or defect is caught immediately, the problem can be fixed before it affects other parts.

A less obvious benefit is latency. At the construction site there is a second team of carpenters who use the pieces to build a house. The pieces cannot be used until they are inspected. Using the first method, the second team cannot begin its work until all the pieces are cut and at least one piece is inspected. The chances are high that the pieces will be delivered in a big batch after they have all been inspected. In the small-batch example the new pieces are delivered without this delay.

The sections that follow relate the small-batch principle to system administration and show many benefits beyond reduced waste and improved latency.
IN-HOUSE SOFTWARE DEPLOYMENT

A company had a team of developers that produced a new release every six months. When that release shipped, the operations team stopped everything and deployed the release into production. The process took three or four weeks and was very stressful for all involved. Scheduling the maintenance window required complex negotiation. Testing the release was complex and required all hands on deck. The actual software installation never worked on the first try. Once deployed, a number of high-priority bugs would be discovered, and each would be fixed by various “hot patches” that would follow.

Even though the deployment process was labor intensive, there was no attempt to automate it. The team had many rationalizations that justified this. The production infrastructure changed significantly between releases, thus making it a moving target. It was believed that any automation would be useless by the next release because each release’s installation instructions were shockingly different. With the next release being so far away, there was always a more important “burning issue” that had to be worked on first. Thus, those who did want to automate the process were told to wait until tomorrow, and tomorrow never came. Lastly, everyone secretly hoped that maybe, just maybe, the next release cycle wouldn’t be so bad. Such optimism is a triumph of hope over experience.

Each release was a stressful, painful month for all involved. Soon it was known as hell month.

To make matters worse, the new software was usually late.
This made it impossible for the operations team to plan ahead. In particular, it was difficult to schedule any vacation time, which just created more stress.

Feeling compassion for the team’s woes, someone proposed that the release should be done less often, perhaps every 9 or 12 months. If something is painful, it is natural to want to do it less frequently.

To everyone’s surprise the operations team suggested going in the other direction: monthly releases.

This was a big-batch situation. To improve, the company didn’t need bigger batches, it needed smaller ones.

People were shocked! Were they proposing that every month be hell month?

No, by doing it more frequently, there would be pressure to automate the process. If something happens infrequently, there’s always an excuse to put off automating it. Also, there would be fewer changes to the infrastructure between releases. If an infrastructure change did break the release automation, it would be easier to fix the problem.

The change did not happen overnight. First the developers changed their methodology from mega releases with many new features, to small iterations, each with a few specific new features. This was a big change, and selling the idea to the team and management was a long battle.

Meanwhile, the operations team automated the testing and deployment processes. The automation could take the latest code, test it, and deploy it into the beta-test area in less than an hour. The push to production was still manual, but by reusing code for the beta rollouts it became increasingly less manual over time.
The result was that the beta area was updated multiple times a day. Since it was automated, there was little reason not to. This made the process continuous, instead of periodic. Each code change triggered the full testing suite, and problems were found in minutes rather than in months.

Pushes to the production area happened monthly because they required coordination among engineering, marketing, sales, customer support, and other groups. That said, all of these teams loved the transition from an unreliable mostly every-six-months schedule to a reliable monthly schedule. Soon these teams started initiatives to attempt weekly releases, with hopes of moving to daily releases. In the new small-batch world the following benefits were observed:

*Features arrived faster.* While in the past a new feature took up to six months to reach production, now it could go from idea to production in days.

*Hell month was eliminated.* After hundreds of trouble-free pushes to beta, pushing to production was easier than ever.

*The operations team could focus on higher-priority projects.* The team was no longer directly involved in software releases other than fixing the automation, which was rare. This freed up the team for more important projects.

*There were fewer impediments to fixing bugs.* The first step in fixing a bug is to identify which code change was responsible. Big-batch releases had hundreds or thousands of changes to sort through to identify the guilty party. With small batches, it was usually quite obvious where to find the bug.

*Bugs were fixed in less time.* Fixing a bug in code that was written six months ago is much more difficult than fixing a bug
in code while it is still fresh in your mind. Small batches meant bugs were reported soon after the code was written, which meant developers could fix them more expertly in a shorter amount of time.

*Developers experienced instant gratification.* Waiting six months to see the results of your efforts is demoralizing. Seeing your code help people shortly after it was written is addictive.

*Most importantly, the operations team could finally take long vacations,* the kind that require advance planning and scheduling, thus giving them a way to reset and live healthier lives.

While these technical benefits are worthwhile, the business benefits are even more exciting:

- **Their ability to compete improved.** Confidence in the ability to add features and fix bugs led to the company becoming more aggressive about new features and fine-tuning existing ones. Customers noticed and sales improved.

- **Fewer missed opportunities.** The sales team had been turning away business because of the company’s inability to strike fast and take advantage of opportunities as they arrived. Now the company could enter markets it hadn’t previously imagined.

- **Enabled a culture of automation and optimization.** Rapid releases removed common excuses not to automate. New automation brought consistency, repeatability, better error checking, and less manual labor. Plus, automation could run any time, not just when the operations team was available.
FAILOVER

Stack Overflow’s main website infrastructure is in a data center in New York City. If the data center fails or needs to be taken down for maintenance, duplicate equipment and software are running in Oregon.

The failover process is complex. Database masters need to be transitioned. Services need to be reconfigured. It takes a long time and requires skills from four different teams. Every time the process happens it fails in new and exciting ways, requiring ad-hoc solutions invented by whoever is doing the procedure.

In other words, the failover process is risky. When Tom was hired at Stack, his first thought was, “I hope I’m not on call when we have that kind of emergency.”

Drunk driving is risky, so we avoid doing it. Failovers are risky, so we should avoid them, too. Right?

Wrong. There is a difference between behavior and process. Risky behaviors are inherently risky; they cannot be made less risky. Drunk driving is a risky behavior. It cannot be done safely, only avoided.

A failover is a risky process. A risky process can be made less risky by doing it more often.

The next time a failover was attempted at Stack Overflow, it took 10 hours. The infrastructure in New York had diverged from Oregon significantly. Code that was supposed to seamlessly failover had been tested only in isolation and failed when used in a real environment. Unexpected dependencies were discovered, in some cases creating catch-22 situations that had to be resolved in the heat of the moment.
This 10-hour ordeal was the result of big batches. Because failovers happened rarely, there was an accumulation of infrastructure skew, dependencies, and stale code. There was also an accumulation of ignorance: new hires had never experienced the process; others had fallen out of practice.

To fix this problem the team decided to do more failovers. The batch size was the number of accumulated changes and other things that led to problems during a failover. Rather than let the batch size grow and grow, the team decided to keep it small. Rather than waiting for the next real disaster to exercise the failover process, they would introduce simulated disasters.

The concept of activating the failover procedure on a system that was working perfectly may seem odd, but it is better to discover bugs and other problems in a controlled situation than during an emergency. Discovering a bug during an emergency at 4 a.m. is troublesome because those who can fix it may be unavailable—and if they are available, they’re certainly unhappy to be awakened. In other words, it is better to discover a problem on Saturday at 10 a.m. when everyone is awake, available, and presumably sober.

If schoolchildren can do fire drills once a month, certainly system administrators can practice failovers a few times a year. The team began doing failover drills every two months until the process was perfected.

Each drill surfaced problems with code, documentation, and procedures. Each issue was filed as a bug and was fixed by the next drill. The next failover took five hours, then two hours, then eventually the drills could be done in an hour with zero user-visible downtime.
The process found infrastructure changes that had not been replicated in Oregon and code that didn’t failover properly. It identified new services that hadn’t been engineered for smooth failover. It discovered a process that could be done only by one particular engineer. If he was on vacation or unavailable, the company would be in trouble. He was a single point of failure.

Over the course of a year all these issues were fixed. Code was changed, better pretests were developed, and drills gave each member of the SRE [site reliability engineering] team a chance to learn the process. Eventually the overall process was simplified and easier to automate. The benefits Stack Overflow observed included:

- **Fewer surprises.** The more frequent the drills, the smoother the process became.
- **Reduced risk.** The procedure was more reliable because there were fewer hidden bugs waiting to bite.
- **Higher confidence.** The company had more confidence in the process, which meant the team could now focus on more important issues.
- **Bugs fixed faster.** The smaller accumulation of infrastructure and code changes meant each drill tested fewer changes. Bugs were easier to identify and faster to fix.
- **Bugs fixed during business hours.** Instead of having to find workarounds or implement fixes at odd hours when engineers were sleepy, they were worked on during the day when engineers were there to discuss and implement higher-quality fixes.

**Practice makes perfect.** Operations team members all had
a turn at doing the process in an environment where they had help readily available. No person was a single point of failure.

**Improved process documentation and automation.** Documentation improved while the drill was running. Automation was easier to write because the repetition helped the team see what could be automated or what pieces were most worth automating.

**New opportunities revealed.** The drills were a big source of inspiration for big-picture projects that would radically improve operations.

**Happier developers.** There was less chance of being woken up at odd hours.

**Happier operations team.** The fear of failovers was reduced, leading to less stress. More people trained in the failover procedure meant less stress on the people who had previously been single points of failure.

**Again, it became easier to schedule long vacations.**

**THE MONITORING PROJECT**

An IT department needed a monitoring system. The number of servers had grown to the point where situational awareness was no longer possible by manual means. The lack of visibility into the company’s own network meant that outages were often first reported by customers, and often after the outage had been going on for hours and sometimes days.

The system administration team had a big vision for what the new monitoring system would be like. All services and networks would be monitored, the monitoring system would run on a pair of big, beefy machines, and when problems were
detected a sophisticated on-call schedule would be used to determine whom to alert.

Six months into the project they had no monitoring system. The team was caught in endless debates over every design decision: monitoring strategy, how to monitor certain services, how the pager rotation would be handled, and so on. The hardware cost alone was high enough to require multiple levels of approval.

Logically the monitoring system couldn’t be built until the planning was done, but sadly it looked like the planning would never finish. The more the plans were discussed, the more issues were raised that needed to be discussed. The longer the planning lasted, the less likely the project would come to fruition.

Fundamentally they were having a big-batch problem. They wanted to build the perfect monitoring system in one big batch. This is unrealistic.

The team adopted a new strategy: small batches. Rather than building the perfect system, they would build a small system and evolve it.

At each step they would be able to show it to their co-workers and customers to get feedback. They could validate assumptions for real, finally putting a stop to the endless debates the requirements documents were producing. By monitoring something—anything—they would learn the reality of what worked best.

Small systems are more flexible and malleable; therefore, experiments are easier. Some experiments would work well, others wouldn’t. Because they would keep things small
and flexible, however, it would be easy to throw away the mistakes. This would enable the team to pivot, meaning they could change direction based on recent results. It is better to pivot early in the development process than to realize well into it that you’ve built something nobody likes.

Google calls this “launch early and often.” Launch as early as possible even if that means leaving out most of the features and launching to only a few select users. What you learn from the early launches informs the decisions later on and produces a better service in the end.

Launching early and often also gives you the opportunity to build operational infrastructure early. Some companies build a service for a year and then launch it, informing the operations team only a week prior. IT then has little time to develop operational practices such as backups, on-call playbooks, and so on. Therefore, those things are done badly. With the launch-early-and-often strategy, you gain operational experience early and you have enough time to do it right.

This is also known as the MVP strategy. As defined by Eric Ries in 2009, “The minimum viable product is that version of a new product which allows a team to collect the maximum amount of validated learning about customers with the least effort” (“Minimum Viable Product: a guide”; http://www.startuplessonslearned.com/2009/08/minimum-viable-product-guide.html). In other words, rather than focusing on new functionality in each release, focus on testing an assumption in each release.

The team building the monitoring system adopted the
launch-early-and-often strategy. They decided that each iteration, or small batch, would be one week long. At the end of the week they would release what was running in their beta environment to their production environment and ask for feedback from stakeholders.

For this to work they had to pick very small chunks of work. Taking a cue from Jason Punyon and Kevin Montrose (“Providence: Failure Is Always an Option”; http://jasonpunyon.com/blog/2015/02/12/providence-failure-is-always-an-option/), they called this “What can get done by Friday?”-driven development.

Iteration 1 had the goal of monitoring a few servers to get feedback from various stakeholders. The team installed an open-source monitoring system on a virtual machine. This was in sharp contrast to their original plan of a system that would be highly scalable. Virtual machines do not have the I/O and network performance that physical hardware has. Hardware could not be ordered in a one-week time frame, however. So the first iteration used virtual machines.

At the end of this iteration, the team didn’t have their dream monitoring system, but they had more monitoring capability than ever before.

In this iteration they learned that SNMP (Simple Network Management Protocol) was disabled on most of the organization’s networking equipment. They would have to coordinate with the network team if they were to collect network utilization and other statistics. It was better to learn this now than to have their major deployment scuttled by making this discovery during the final big deployment. To work
around this, the team decided to focus on monitoring other things, such as servers and services. This gave the network team time to create and implement a project to enable SNMP in a secure and tested way.

Iterations 2 and 3 proceeded well, adding more machines and testing other configuration options and features.

During iteration 4, however, the team noticed that the other system administrators and managers hadn’t been using the system much. This was worrisome. They paused to talk one-on-one with people to get some honest feedback.

What the team learned was that without the ability to have dashboards that displayed historical data, the system wasn’t very useful to its users. In all the past debates this issue had never been raised. Most confessed they hadn’t thought it would be important until they saw the system running; others hadn’t raised the issue because they simply assumed all monitoring systems had dashboards.

It was time to pivot.

The software package that had been the team’s second choice had very sophisticated dashboard capabilities. More importantly, dashboards could be configured and customized by individual users. They were self-service.

After much discussion, the team decided to pivot to the other software package. In the next iteration, they set up the new software and created an equivalent set of configurations. This went very quickly because a lot of work from the previous iterations could be reused: the decisions on what and how to monitor the previous SNMP work with the network team and so on.
By iteration 6, the entire team was actively using the new software. Managers were setting up dashboards to display key metrics that were important to them. People were enthusiastic about the new system.

Something interesting happened around this time: a major server crashed on Saturday morning. The monitoring system alerted the sysadmin team, who were able to fix the problem before staff arrived at the office on Monday. In the past there had been similar outages but repairs had not begun until the sysadmins arrived on Monday morning, well after most employees had arrived. This showed management, in a very tangible way, the value of the system.

Iteration 7 had the goal of writing a proposal to move the monitoring system to physical machines so that it would scale better. By this time the managers who would approve such a purchase were enthusiastically using the system; many had become quite expert at creating custom dashboards. The case was made to move the system to hardware for better scaling and performance, and to use a duplicate set of hardware for a hot spare site in another data center.

The plan was approved.

In future iterations the system became more valuable to the organization as the team implemented features such as

---

**Thanksgiving**

The U.S. Thanksgiving holiday involves a large feast. If you are not used to cooking a large meal for many people, this once-a-year event can be a stressful, scary time. Any mistakes are magnified by their visibility: all your relatives are there to see you fail. It is a big batch.

Some people turn this into a small-batch situation by attempting new recipes in the weeks ahead of time, or by making certain key elements in a large batch as a test run. These techniques reduce risk and stress from an otherwise-busy holiday.
as a more sophisticated on-call schedule, monitored more services, and so on. The benefits of small batches observed by the sysadmin team included:

*Testing assumptions early prevents wasted effort.* The ability to fail early and often means the team can pivot. Problems can be fixed sooner rather than later.

*Providing value earlier builds momentum.* People would rather have some features today than all the features tomorrow. Some monitoring is better than no monitoring. The naysayers see results and become advocates. Management has an easier time approving something that isn’t hypothetical.

*Experimentation is easier.* Often, people develop emotional attachment to code. With small batches they can be more agile because they have grown less attached to past decisions.

*Instant gratification.* The team saw the results of their work faster, which improved morale.

*Less stress.* There is no big, scary, due date, just a constant flow of new features.

*Big-batch debating is procrastination.* Much of the early debate had been about details and features that didn’t matter or didn’t get implemented.

The first few weeks were the hardest. The initial configuration required special skills. Once it was running, however, people with less technical skill or desire could add rules and make dashboards. In other words, by taking a lead and setting up the scaffolding, others can follow. This is an important point of technical leadership. Technical leadership means going first and making it easy for others to follow.
A benefit of using the MVP model is that the system is always working. This is called “always being in a shippable state.” The system is always working and providing benefit, even if not all the features are delivered. Therefore, if more urgent projects take the team away, the system is still usable and running. If the original big-batch plan had continued, the appearance of a more urgent project might have left the system half developed but unlaunched. The work done so far would have been for naught.

SUMMARY
Why are small batches better?
  Small batches result in happier customers. Features get delivered with less latency. Bugs are fixed faster.
  Small batches reduce risk. By testing assumptions, the prospect of future failure is reduced. More people get experience with procedures, which means our skills improve.
  Small batches reduce waste. They avoid endless debates and perfectionism that delay the team in getting started. Less time is spent implementing features that don’t get used. In the event that higher-priority projects come up, the team has already delivered a usable system.
  Small batches improve the ability to innovate. Because experimentation is encouraged, the team can test new ideas and keep the good ones. We can take risks. We are less attached to old pieces that must be thrown away.
  Small batches improve productivity. Bugs are fixed quicker and the process of fixing them is accelerated because the code is fresher in the mind.
Small batches encourage automation. When something must happen often, excuses not to automate go away.

Small batches encourage experimentation. The team can try new things—even crazy ideas, some of which turn into competition-killing features. We fear failure less because we can easily undo a small batch if the experiment fails. More importantly, experimentation allows the team to learn something that will help them make future improvements.

Small batches make system administrators happier. We get instant gratification, and hell month disappears. It is simply a better way to work.

Reprinted with permission. Volume 1: *The Practice of System and Network Administration*, 3rd Edition
By Thomas A. Limoncelli, Christina J. Hogan, Strata R. Chalup

**LOVE IT, HATE IT? LET US KNOW** feedback@queue.acm.org

Thomas A. Limoncelli is an author, speaker, and system administrator. He is a site reliability engineer at Stack Overflow Inc. in New York City. His books include The Practice of Cloud Administration ([http://the-cloud-book.com](http://the-cloud-book.com)), The Practice of System and Network Administration ([http://the-sysadmin-book.com](http://the-sysadmin-book.com)) and Time Management for System Administrators. He blogs at EverythingSysadmin.com and tweets at @YesThatTom.

Copyright © 2016 held by owner/author. Publication rights licensed to ACM.
Dear KV,

I’ve been reading some pull requests from a developer who has recently been working in code that I also have to look at from time to time. The code he has been submitting is full of strange changes that he claims are optimizations. Instead of simply returning a value such as 1, 0, or -1 for error conditions, he allocates a variable and then increments or decrements it, and then jumps to the return statement. I haven’t bothered to check whether or not this would save instructions, because I know from benchmarking the code that those instructions are not where the majority of the function spends its time. He has argued that any instruction we don’t execute saves us time, and my point is that his code is confusing and hard to read. If he could show a five or ten percent increase in speed, it might be worth considering, but he has not been able to show that in any type of test. I’ve blocked several of his commits, but I would prefer to have a usable argument against this type of optimization.

Pull the Other One

Dear Pull,

Saving instructions—how very 1990s of him. It’s always nice when people pay attention to details, but sometimes they simply don’t pay attention to the right ones. While KV would
never encourage developers to waste instructions, given the state of modern software, it does seem like someone already has. KV would, as you did, come out on the side of legibility over the saving of a few instructions.

It seems that no matter what advances are made in languages and compilers, there are always programmers who think they're smarter than their tools, and sometimes they're right about that, but mostly they're not. Reading the output of the assembler and counting the instructions may be satisfying for some, but there had better be a lot more proof than that to justify obfuscating code. I can only imagine a module full of code that looks like this:

```c
if (some condition) {
    retval++;
    goto out:
} else {
    retval--;  
    goto out:
}
...
out:
    return(retval)
```

and, honestly, I don’t really want to. Modern compilers, or even not-so-modern ones, play all the tricks that programmers used to have to play by hand—inlining, loop
unrolling, and many others—and yet there are still some programmers who insist on fighting their own tools.

When the choice is between code clarity and minor optimizations, clarity must, nearly always, win. A lack of clarity is the source of bugs, and it’s no good having code that’s fast and wrong. First the code must be right, then the code must perform; that is the priority that any sane programmer must obey. Insane programmers, well, they’re best avoided. Eventually they wind up moving to a Central American nation, mixing their own drugs in bathtubs, and claiming they can unlock iPhones.

The other significant problem with the suggested code is that it violates a common coding idiom. All languages, including computer languages, have idioms, as pointed out at length in *The Practice of Programming* by Brian W. Kernighan and Rob Pike (Addison-Wesley Professional, 1999), which I recommended to readers more than a decade ago. Let’s not think about the fact that that book is still relevant, and that I’ve been repeating myself every decade. No matter what you think of a computer language, you ought to respect its idioms for the same reason that one has to know idioms in a human language—they facilitate communication, which is the true purpose of all languages, programming or otherwise. A language idiom grows organically from the use of a language. Most C programmers, though not all of course, will write an infinite loop in this way:
for (;;) {
}

or as

while (1) {
}

with an appropriate break statement somewhere inside to handle exiting the loop when there is an error. In fact, checking the *Practice of Programming* book, I find that this is mentioned early on (in section 1.3). For the return case, you mention it is common to return using a value such as 1, 0, or -1 unless the return encodes more than true, false, or error. Allocating a stack variable and incrementing or decrementing it and adding a **goto** is not an idiom I’ve ever seen in code, anywhere—and now that you’re on the case, I hope that I never have to.

Moving from this concrete bit of code to the abstract question of when it makes sense to allow some forms of code trickery into the mix really depends on several factors, but mostly on how much speedup can be derived from twisting the code a bit to match the underlying machine a bit more closely. After all, most of the hand optimizations you see in low-level code, in particular C and its bloated cousin C++, exist because the compiler cannot recognize a good way to map what the programmer wants to do onto the way the underlying machine actually works. Leaving aside the fact that most software engineers really don’t know how a
computer works, and leaving aside that what most of them were taught—if they were taught—about computers hails from the 1970s and 1980s, before superscalar processors and deep pipelines were a standard feature of CPUs, it is still possible to find ways to speed up by playing tricks on the compiler.

The tricks themselves aren’t that important to this conversation; what’s important is knowing how to measure their effects on the software. This is a difficult and complicated task. It turns out that simply counting instructions as your co-worker has done doesn’t tell you very much about the runtime of the underlying code. In a modern CPU the most precious resource is no longer instructions, except in a very small number of compute-bound workloads. Modern systems don’t choke on instructions; they drown in data. The cache effects of processing data far outweigh the overhead of an extra instruction or two, or ten. A single cache miss is a 32-nanosecond penalty, or about 100 cycles on a 3-GHz processor. A simple MOV instruction, which puts a single, constant number into a CPU’s register, takes one-quarter of a cycle, according to Agner Fog at the Technical University of Denmark ([http://www.agner.org/optimize/instruction_tables.pdf](http://www.agner.org/optimize/instruction_tables.pdf)).

That someone has gone so far as to document this for quite a large number of processors is staggering, and those interested in the performance of their optimizations might well lose themselves in that site generally ([http://www.agner.org](http://www.agner.org)).

The point of the matter is that a single cache miss is more
expensive than many instructions, so optimizing away a few instructions isn’t really going to win your software any speed tests. To win speed tests you have to measure the system, see where the bottlenecks are, and clear them if you can. That, though, is a subject for another time.

KV

LOVE IT, HATE IT? LET US KNOW feedback@queue.acm.org

Kode Vicious, known to mere mortals as George V. Neville-Neil, works on networking and operating-system code for fun and profit. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking, operating systems, and rewriting your bad code (OK, maybe not that last one). He earned his bachelor’s degree in computer science at Northeastern University in Boston, Massachusetts, and is a member of ACM, the Usenix Association, and IEEE. Neville-Neil is the co-author with Marshall Kirk McKusick and Robert N. M. Watson of The Design and Implementation of the FreeBSD Operating System (second edition). He is an avid bicyclist and traveler who currently lives in New York City.

Copyright © 2016 held by owner/author. Publication rights licensed to ACM.
An everyday problem in our industry is understanding how software is consuming resources, particularly CPUs. What exactly is consuming how much, and how did this change since the last software version? These questions can be answered using software profilers, tools that help direct developers to optimize their code and operators to tune their environment. The output of profilers can be verbose, however, making it laborious to study and comprehend. The flame graph provides a new visualization for profiler output and can make for much faster comprehension, reducing the time for root cause analysis.

In environments where software changes rapidly, such as the Netflix cloud microservice architecture, it is especially important to understand profiles quickly. Faster comprehension can also make the study of foreign software more successful, where one’s skills, appetite, and time are strictly limited.

Flame graphs can be generated from the output of many different software profilers, including profiles for different resources and event types. Starting with CPU profiling, this
article describes how flame graphs work, then looks at the real-world problem that led to their creation.

CPU PROFILING
A common technique for CPU profiling is the sampling of stack traces, which can be performed using profilers such as Linux perf_events and DTrace. The stack trace is a list of function calls that show the code-path ancestry. For example, the following stack trace shows each function as a line, and the top-down ordering is child to parent:

```
SpinPause
StealTask::do_it
GCTaskThread::run
java_start
start_thread
```

Balancing considerations that include sampling overhead, profile size, and application variation, a typical CPU profile might be collected in the following way: stack traces are sampled at a rate of 99 times per second (not 100, to avoid lock-step sampling) for 30 seconds across all CPUs. For a 16-CPU system, the resulting profile would contain 47,520 stack-trace samples. As text, this would be hundreds of thousands of lines.

Fortunately, profilers have ways to condense their output. DTrace, for example, can measure and print unique stack traces, along with their occurrence count. This approach is more effective than it might sound: identical stack traces may
be repeated during loops or when CPUs are in the idle state. These are condensed into a single stack trace with a count.

Linux perf_events can condense profiler output even further: not only identical stack trace samples, but also subsets of stack traces can be coalesced. This is presented as a tree view with counts or percentages for each code-path branch, as shown in figure 1.

In practice, the output summary from either DTrace or perf_events is sufficient to solve the problem in many cases, but there are also cases where the output produces a wall of text, making it difficult or impractical to comprehend much of the profile.

**FIGURE 1: SAMPLE LINUX PERF EVENTS TREE VIEW**
THE PROBLEM
The problem that led to the creation of flame graphs was application performance on the Joyent public cloud. The application was a MySQL database that was consuming around 40 percent more CPU resources than expected.

DTrace was used to sample user-mode stack traces for the application at 997 Hz for 60 seconds. Even though DTrace printed only unique stack traces, the output was 591,622 lines long, including 27,053 unique stack traces. Fortunately, the last screenful—which included the most frequently sampled stack traces—looked promising, as shown in figure 2.

The most frequent stack trace included a MySQL calc_sum_of_all_status() function, indicating that it was processing a "show status" command. Perhaps the customer had enabled aggressive monitoring, explaining the higher CPU usage?

FIGURE 2: MYSQL DTRACE PROFILE SUBSET
To quantify this theory, the stack-trace count (5,530) was divided into the total samples in the captured profile (348,427), showing that it was responsible for only 1.6 percent of the CPU time. This alone could not explain the higher CPU usage. It was necessary to understand more of the profile.

Browsing more stack traces became an exercise in diminishing returns, as they progressed in order from most to least frequent. The scale of the problem is evident in figure 3, where the entire DTrace output becomes a featureless gray square.

With so much output to study, solving this problem within a reasonable time frame began to feel insurmountable. There had to be a better way.

I created a prototype of a visualization that leveraged the hierarchical nature of stack traces to combine common

FIGURE 3: FULL MYSQL DTRACE PROFILE OUTPUT
paths. The result is shown in figure 4, which visualizes the same output as in figure 3. Since the visualization explained why the CPUs were “hot” (busy), I thought it appropriate to choose a warm palette. With the warm colors and flame-like shapes, these visualizations became known as flame graphs. [An interactive version of figure 4, in SVG [scalable vector graphics] format, is available at http://queue.acm.org/downloads/2016/Gregg4.svg.]

The flame graph allowed the bulk of the profile to be understood very quickly. It showed that the earlier lead, the MySQL status command, was responsible for only 3.28 percent of the profile when all stacks were combined. The

**FIGURE 4: FULL MYSQL PROFILER OUTPUT AS A FLAME GRAPH**
bulk of the CPU time was consumed in MySQL `join`, which provided a clue to the real problem. The problem was located and fixed, and CPU usage was reduced by 40 percent.

**FLAME GRAPHS EXPLAINED**

A flame graph visualizes a collection of stack traces (a.k.a. call stacks), shown as an adjacency diagram with an inverted icicle layout. Flame graphs are commonly used to visualize CPU profiler output, where stack traces are collected using sampling.

A flame graph has the following characteristics:

- A stack trace is represented as a column of boxes, where each box represents a function (a stack frame).
- The y-axis shows the stack depth, ordered from root at the bottom to leaf at the top. The top box shows the function that was on-CPU when the stack trace was collected, and everything beneath that is its ancestry. The function beneath a function is its parent.
- The x-axis spans the stack trace collection. It does not show the passage of time, so the left-to-right ordering has no special meaning. The left-to-right ordering of stack traces is alphabetical on the function names, from the root to the leaf of each stack. This maximizes box merging: when identical function boxes are horizontally adjacent, they are merged.
- The width of each function box shows the frequency at which that function was present in the stack traces, or part of a stack trace ancestry. Functions with wide boxes were more frequent in the stack traces than those with narrow...
boxes, in proportion to their widths.

- If the box is wide enough, it displays the full function name. If not, either a truncated function name with an ellipsis is shown, or nothing.
- The background color for each box is not significant and is picked at random to be a warm hue. This randomness helps the eye differentiate boxes, especially for adjacent thin “towers.” Other color schemes are discussed later.
- The profile visualized may span a single thread, multiple threads, multiple applications, or multiple hosts. Separate flame graphs can be generated if desired, especially for studying individual threads.
- Stack traces may be collected from different profiler targets, and widths can reflect measures other than sample counts. For example, a profiler (or tracer) could measure the time a thread was blocked, along with its stack trace. This can be visualized as a flame graph, where the x-axis spans the total blocked time, and the flame graph shows the blocking code paths.

As the entire profiler output is visualized at once, the end user can navigate intuitively to areas of interest. The shapes and locations in the flame graphs become visual maps for the execution of software.

While flame graphs use interactivity to provide additional features, these characteristics are fulfilled by a static flame graph, which can be shared as an image (e.g., a PNG file or printed on paper). While only wide boxes have enough room to contain the function label text, they are also usually sufficient to show the bulk of the profile.
INTERACTIVITY
Flame graphs can support interactive features to reveal more detail, improve navigation, and perform calculations.

The original implementation of flame graphs creates an SVG image with embedded JavaScript for interactivity, which is then loaded in a browser. It supports three interactive features: Mouse-over for information, click to zoom, and search.

Mouse-over for information
On mouse-over of boxes, an informational line below the flame graph and a tooltip display the full function name, the number of samples present in the profile, and the corresponding percentage for those samples in the profile. For example, **Function: mysqld::JOIN::exec** (272,959 samples, 78.34 percent).

This is useful for revealing the function name from unlabeled boxes. The percentage also quantifies code paths in the profile, which helps the user prioritize leads and estimate improvements from proposed changes.

Click to zoom
When a box is clicked, the flame graph zooms horizontally. This reveals more detail, often including function names for the child functions. Ancestor frames below the clicked box are shown with a faded background as a visual clue that their widths are now only partially shown. A Reset Zoom button is included to return to the original full profile view. Clicking any box while zoomed will reset the zoom to focus on that new box.
Search

A search button or keystroke (Ctrl-F) prompts the user for a search term, which can include regular expressions. All function names in the profile are searched, and any matched boxes are highlighted with magenta backgrounds. The sum of matched stack traces is also shown on the flame graph as a percentage of the total profile, as in figure 5. (An interactive version of figure 5 in SVG format is available at http://queue.acm.org/downloads/2016/Gregg5.svg.)

This is useful not just for locating functions, but also for highlighting logical groups of functions—for example,

FIGURE 5: SEARCH HIGHLIGHTING
searching for \texttt{"ext4\_"} to find the Linux \texttt{ext4} functions.

For some flame graphs, many different code paths may end with a function of interest—for example, spin-lock functions. If this appeared in 20 or more locations, calculating their combined contribution to the profile would be a tedious task, involving finding then adding each percentage. The search function makes this trivial, as a combined percentage is calculated and shown on screen.

\textbf{INSTRUCTIONS}

There are several implementations of flame graphs so far.\footnote{The original implementation, FlameGraph, was written in the Perl programming language and released as open source. It makes the generation of flame graphs a three-step sequence, including the use of a profiler:}

1. Use a profiler to gather stack traces (e.g., Linux \texttt{perf\_events}, DTrace, Xperf).
2. Convert the profiler output into the “folded” intermediate format. Various programs are included with the FlameGraph software to handle different profilers; the program names begin with “stackcollapse”.
3. Generate the flame graph using flamegraph.pl. This reads the previous folded format and converts it to an SVG flame graph with embedded JavaScript.

The folded stack-trace format puts stack traces on a single line, with functions separated by semicolons, followed by a space and then a count. The name of the application, or the name and process ID separated by a dash, can be optionally included at the start of the folded stack trace,
followed by a semicolon. This groups the application’s code paths in the resulting flame graph.

For example, a profile containing the following three stack traces:

```
func_c
func_b
func_a
start_thread
```
```
func_d
func_a
start_thread
```
```
func_d
func_a
start_thread
```

becomes the following in the folded format:

```
start_thread;func_a;func_b;func_c 1
start_thread;func_a;func_d 2
```

If the application name is included—for example, “java”—it would then become:

```
java;start_thread;func_a;func_b;func_c 1
java;start_thread;func_a;func_d 2
```
This intermediate format has allowed others to contribute converters for other profilers. There are now stackcollapse programs for DTrace, Linux perf_events, FreeBSD pmcstat, Xperf, SystemTap, Xcode Instruments, Intel VTune, Lightweight Java Profiler, Java jstack, and gdb.4

The final flamegraph.pl program supports many options for customization, including changing the title of the flame graph.

As an example, the following steps fetch the FlameGraph software, gather a profile on Linux (99 Hz, all CPUs, 60 seconds), and then generate a flame graph from the profile:

```bash
# git clone https://github.com/brendangregg/FlameGraph
# cd FlameGraph
# perf record -F 99 -a -g -- sleep 60
# perf script | ./stackcollapse-perf.pl | ./flamegraph.pl > out.svg
```

Since the output of stackcollapse has single lines per record, it can be modified using grep/sed/awk if needed before generating a flame graph.

The online flame graph documentation includes instructions for using other profilers.4,5

**FLAME GRAPH INTERPRETATION**

Flame graphs can be interpreted as follows:

- The top edge of the flame graph shows the function that was running on the CPU when the stack trace was collected.
For CPU profiles, this is the function that is directly consuming CPU cycles. For other profile types, this is the function that led directly to the instrumented event.

- Look for large plateaus along the top edge, as these show a single stack trace was frequently present in the profile. For CPU profiles, this means a single function was frequently running on-CPU.
- Reading top down shows ancestry. A function was called by its parent, which is shown directly below it; the parent was called by its parent shown below it, and so on. A quick scan downward from a function identifies why it was called.
- Reading bottom up shows code flow and the bigger picture. A function calls any child functions shown above it, which, in turn, call functions shown above them. Reading bottom up also shows the big picture of code flow before various forks split execution into smaller towers.
- The widths of function boxes can be directly compared: wider boxes mean a greater presence in the profile and are the most important to understand first.
- For CPU profiles that employ timed sampling of stack traces, if a function box is wider than another, this may be because it consumes more CPU per function call or because the function was simply called more often. The function-call count is not shown or known via sampling.
- Major forks in the flame graph, spotted as two or more large towers atop a single function, can be useful to study. They can indicate a logical grouping of code, where a function processes work in stages, each with its own function.
They can also be caused by a conditional statement, which chooses which function to call.

INTERPRETATION EXAMPLE
As an example of interpreting a flame graph, consider the mock one shown in figure 6. Imagine this is visualizing a CPU profile, collected using timed samples of stack traces (as is typical).

The top edge shows that function $g()$ is on-CPU the most; $d()$ is wider, but its exposed top edge is on-CPU the least. Functions including $b()$ and $c()$ do not appear to have been sampled on-CPU directly; rather, their child functions were running.

Functions beneath $g()$ show its ancestry: $g()$ was called by $f()$, which was called by $d()$, and so on.

Visually comparing the widths of functions $b()$ and $h()$ shows that the $b()$ code path was on-CPU about four times as much as the $h()$ code path.
more than \( \text{h()} \). The actual functions on-CPU in each case were their children.

A major fork in the code paths is visible where \( \text{a()} \) calls \( \text{b()} \) and \( \text{h()} \). Understanding why the code does this may be a major clue to its logical organization. This may be the result of a conditional (if conditional, call \( \text{b()} \), else call \( \text{h()} \)) or a logical grouping of stages (where \( \text{a()} \) is processed in two parts: \( \text{b()} \) and \( \text{h()} \)).

OTHER CODE-PATH VISUALIZATIONS
As was shown in figure 1, Linux perf_events prints a tree of code paths with percentage annotations. This is another type of hierarchy visualization: an indented tree layout. Depending on the profile, this can sometimes sufficiently summarize the output, but not always. Unlike flame graphs, one cannot zoom out to see the entire profile and still make sense of this text-based visualization, especially after the percentages can no longer be read.

KCachegrind\textsuperscript{14} visualizes code paths from profile data using a directed acyclic graph. This involves representing functions as labeled boxes (where the width is scaled to fit the function name), parent-to-child relationships as arrows, and then profile data is annotated on the boxes and arrows as percentages with bar chart-like icons. Similar to the problem with perf_events, if the visualization is zoomed out to fit a complex profile, then the annotations may no longer be legible.

The sunburst layout is equivalent to the icicle layout used by flame graphs, but it uses polar coordinates.\textsuperscript{7} While this
can generate interesting shapes, there are some difficulties: function names are harder to draw and read from sunburst slices than they are in the rectangular flame-graph boxes. Also, comparing two functions becomes a matter of comparing two angles rather than two line lengths, which has been evaluated as a more difficult perceptual task.\textsuperscript{10}

Flame charts are a similar code-path visualization to flame graphs (and were inspired by flame graphs\textsuperscript{13}). On the x-axis, however, they show the passage of time instead of an alphabetical sort. This has its advantages: time-ordered issues can be identified. It can greatly reduce merging, however, a problem exacerbated when profiling multiple threads. It could be a useful option for understanding time order sequences when used with flame graphs for the bigger picture.

CHALLENGES
Challenges with flame graphs mostly involve system profilers and not flame graphs themselves. There are two typical problems with profilers:

- **Stack traces are incomplete.** Some system profilers truncate to a fixed stack depth (e.g., 10 frames), which must be increased to capture the full stack traces, or else frame merging can fail. A worse problem is when the software compiler reuses the frame pointer register as a compiler optimization, breaking the typical method of stack-trace collection. The fix requires either a different compiled binary (e.g., using gcc’s -fno-omit-frame-pointer) or a different stack-walking technique.

- **Function names are missing.** In this case, the stack trace
is complete, but many function names are missing and may be represented as hexadecimal addresses. This commonly happens with JIT (just-in-time) compiled code, which may not create a standard symbol table for profilers. Depending on the profiler and runtime, there are different fixes. For example, Linux perf_events supports supplemental symbol files, which the application can create.

At Netflix we encountered both problems when attempting to create flame graphs for Java. The first has been fixed by the addition of a JVM (Java Virtual Machine) option—XX:+PreserveFramePointer, which allows Linux perf_events to capture full stack traces. The second has been fixed using a Java agent, perf-map-agent, which creates a symbol table for Java methods.

One challenge with the Perl flame-graph implementation has been the resulting SVG file size. For a large profile with many thousands of unique code paths, the SVG file can be tens of megabytes in size, which becomes sluggish to load in a browser. The fix has been to elide code paths that are so thin they are normally invisible in the flame graph. This does not affect the big-picture view and has kept the SVG file smaller.

OTHER COLOR SCHEMES
Apart from a random warm palette, other flame-graph color schemes can be used, such as for differentiating code or including an extra dimension of data.

Various palettes can be selected in the Perl flame-graph version, including “java,” which uses different hues to highlight a Java mixed-mode flame graph: green for
Java methods, yellow for C++, red for all other user-mode functions, and orange for kernel-mode functions. An example is shown in figure 7. (An interactive version of figure 7 in SVG)

FIGURE 7: JAVA MIXED-MODE CPU FLAME GRAPH
format is available at http://queue.acm.org/downloads/2016/Gregg7.svg.

Another option is a hashing color scheme, which picks a color based on a hash of the function name. This keeps colors consistent, which is helpful when comparing multiple flame graphs from the same system.

Color can also be used for differential flame graphs, described in the next section.

DIFFERENTIAL FLAME GRAPHS
A differential flame graph shows the difference between two profiles, A and B. The Perl flame-graph software currently supports one method, where the B profile is displayed and then colored using the delta from A to B. Red shades indicate functions that increased, and blue shades indicate those that decreased. A problem with this approach is that some code paths present in the A profile may be missing entirely in the B profile, and so will be missing from the final visualization. This could be misleading.

Another implementation, flamegraphdiff, solves this problem by using three flame graphs. The first shows the A profile, the second shows the B profile, and the third shows only the delta between them. A mouse-over of one function in any flame graph also highlights the others to help navigation. Optionally, the flame graphs can also be colored using a red/blue scheme to indicate which code paths increased or decreased.
OTHER TARGETS
As previously mentioned, flame graphs can visualize any profiler output. This includes stack traces collected on CPU PMC (performance monitoring counter) overflow events, static tracing events, and dynamic tracing events. Following are some specific examples.

Stall Cycles
A stall-cycle flame graph shows code paths that commonly block on processor or hardware resources—typically memory I/O. The input stack traces can be collected using a PMC profiler, such as Linux perf_events. This can direct the developer to employ a different optimization technique for the identified code paths, one that aims to reduce memory I/O rather than reducing instructions.

CPI
CPI (cycles per instruction), or its inverse, IPC (instructions per cycle), is a measure that also helps explain the types of CPU cycles and can direct tuning effort. A CPI flame graph shows a CPU sample flame graph where widths correspond to CPU cycles, but it uses a color scale from red to blue to indicate each function’s CPI: red for a high CPI and blue for a low CPI. This can be accomplished by capturing two profiles—a CPU sample profile and an instruction count profile—and then using a differential flame graph to color the difference between them.
Memory
Flame graphs can shed light on memory growth by visualizing a number of different memory events.

A `malloc()` flame graph, created by tracing the `malloc()` function, visualizes code paths that allocated memory. This can be difficult in practice, as allocator functions can be called frequently, making the cost to trace them prohibitive in some scenarios.

Tracing the `brk()` and `mmap()` syscalls can show code paths that caused an expansion in virtual memory for a process, typically related to the allocation path, although this could also be an asynchronous expansion of the application’s memory. These are typically lower frequency, making them more suitable for tracing.

Tracing memory page faults shows code paths that caused an expansion in physical memory for a process. Unlike allocator code paths, this shows the code that populated the allocated memory. Page faults are also typically a lower-frequency activity.

I/O
The issuing of I/O, such as file system, storage device, and network, can usually be traced using system tracers. A flame graph of these profiles illustrates different application paths that synchronously issued I/O.

In practice, this has revealed types of I/O that were otherwise not known. For example, disk I/O may be issued: synchronously by the application, by a file system read-ahead routine, by an asynchronous flush of dirty data, or by
a kernel background scrub of disk blocks. An I/O flame graph identifies each of these types by illustrating the code paths that led to issuing disk I/O.

**Off-CPU**

Many performance issues are not visible using CPU flame graphs, as they involve time spent while the threads are blocked, not running on a CPU (off-CPU). Reasons for a thread to block include waiting on I/O, locks, timers, a turn on-CPU, and waiting for paging or swapping. These scenarios can be identified by the stack trace when the thread was descheduled. The time spent off-CPU can also be measured by tracing the time from when a thread left the CPU to when it returned. System profilers commonly use static trace points in the kernel to trace these events.

An off-CPU time flame graph can illustrate this off-CPU time by showing the blocked stack traces, where the width of a box is proportional to the time spent blocked.

**Wakes**

A problem found in practice with off-CPU time flame graphs is that they are inconclusive when a thread blocks waiting for a signal from another thread. We needed information on why the other thread took so long.

A wakeup time flame graph can be generated by tracing thread wakeup events. This includes stack traces from the waker threads, and so they shed light on why they were blocked. This flame-graph type can be studied along with an off-CPU time flame graph for more information on blocked threads.
Chain Graphs
One wakeup flame graph may not be enough. The thread that woke up a blocked thread may itself have been blocked by another thread. In practice, one thread may have been blocked on a second, which was blocked on a third, and a fourth.

A chain flame graph is an experimental visualization\(^3\) that begins with an off-CPU flame graph and then adds all wakeup stack traces to the top of each blocked stack. By reading bottom-up, you see the blocked off-CPU stack trace, and then the first stack trace that woke it, then the next stack trace that woke it, and so on. Widths correspond to the time that threads were off-CPU and the time taken for wakeups.

This can be accomplished by tracing all off-CPU and wakeup events with time stamps and stack traces, and post-processing. These events can be extremely frequent, however, and impractical to instrument in production using current tools.

FUTURE WORK
Much of the work related to flame graphs has involved getting different profilers to work with different runtimes so that the input for flame graphs can be captured correctly (e.g., for Node.js, Ruby, Perl, Lua, Erlang, Python, Java, golang, and with DTrace, perf_events, pmcstat, Xperf, Instruments, etc.). There is likely to be more of this type of work in the future.

Another in-progress differential flame graph, called a
white/black differential, uses the single flame-graph scheme described earlier plus an extra region on the right to show only the missing code paths. Differential flame graphs (of any type) should also see more adoption in the future; at Netflix, we are working to have these generated nightly for microservices to identify regressions and aid with performance-issue analysis.

Several other flame-graph implementations are in development, exploring different features. Netflix has been developing d3-flame-graph, which includes transitions when zooming. The hope is that this can provide new interactivity features, including a way to toggle the merge order from bottom-up to top-down, and also to merge around a given function. Changing the merge order has already proven useful for the original flamegraph.pl, which can optionally merge top-down and then show this as an icicle plot. A top-down merge groups together leaf paths, such as spin locks.

CONCLUSION

The flame graph is an effective visualization for collected stack traces and is suitable for CPU profiling, as well as many other profile types. It creates a visual map for the execution of software and allows the user to navigate to areas of interest. Unlike other code-path visualizations, flame graphs convey information intuitively using line lengths and can handle large-scale profiles, while usually remaining readable on one screen. The flame graph has become an essential tool for understanding profiles quickly and has been instrumental in countless performance wins.
Acknowledgments

Inspiration for the general layout, SVG output, and JavaScript interactivity came from Neelakanth Nadgir’s function_call_graph.rb time-ordered visualization for callstacks, which itself was inspired by Roch Bourbonnais’s CallStackAnalyzer and Jan Boerhout’s vftrace. Adrien Mahieux developed the horizontal zoom feature for flame graphs, and Thorsten Lorenz added a search feature to his implementation. Cor-Paul Bezemer researched differential flame graphs and developed the first solution. Off-CPU time flame graphs were first discussed and documented by Yichun Zhang.

Thanks to the many others who have documented case studies, contributed ideas and code, given talks, created new implementations, and fixed profilers to make this possible. See the updates section for a list of this work. Finally, thanks to Deirdré Straughan for editing and feedback.

References

   27th Large Installation System Administration Conference; 
   https://www.usenix.org/conference/lisa13/technical-sessions/plenary/gregg.

4. Gregg, B. FlameGraph. GitHub; https://github.com/brendangregg/FlameGraph.


LOVE IT, HATE IT? LET US KNOW feedback@queue.acm.org

Brendan Gregg is a senior performance architect at Netflix, where he does large-scale computer performance design, analysis, and tuning. He is the author of multiple technical books including Systems Performance (Prentice Hall, 2013). He received the Usenix LISA Award for Outstanding Achievement in System Administration. He was previously a performance lead and kernel engineer at Sun Microsystems, where he developed the ZFS L2ARC and worked on performance. He has also created numerous performance-analysis tools, which have been included in multiple operating systems. His recent work includes developing methodologies and visualizations for performance analysis.

Copyright © 2016 held by owner/author. Publication rights licensed to ACM.
Reading a great research paper is a joy. A team of experts deftly guides you, the reader, through the often complicated research landscape, noting the prior art, the current trends, the pressing issues at hand—and then, sometimes artfully, sometimes through seeming sheer force of will, expands the body of knowledge in a fell swoop of 12 or so pages of prose. A great paper contains a puzzle and a solution; these can be useful, enlightening, or both. A great paper is a small, structured quantum of human ingenuity, creativity, and labor, in service of a growing understanding of our world and the future worlds we may inhabit.

Unfortunately, information overload is a defining problem of our time, and computer science research is no exception. The volume of research produced each year in computer science is heartening, but it can be difficult to determine which papers are most deserving of our scarce time. This volume of papers is also at odds with many of the best elements of paper reading: distillation of work to its critical essence, thoughtful consideration of its nuances and the context in which the research was performed, and application of concepts to one’s own technical problems and experiences.

As a result, the past few years have seen a rise in interest
and organizations—such as Papers We Love and its many chapters—devoted to the joy and utility of reading computer science research: curated-paper discussions have escaped the traditionally academic “reading seminar” format and have been supplanted by groups of hundreds of participants meeting regularly, at startups and community centers, to discuss the latest and greatest computer science research. This is exciting. Why should the greatest of papers be enjoyed only in academia? As a public good, research should be read, discussed, digested, and enjoyed by all interested parties.

ACM has a particularly important role to play in this democratization of access to research. First, the ACM Digital Library is the largest collection of computer science research in the world, with hundreds of thousands of papers, articles, and manuscripts. Second, the ACM membership consists of world experts across all subfields of computer science, from Turing laureates to ACM Fellows, from upstart academics to engineers on the cutting edge of practice. Separately, these are unparalleled resources; put together, they are even more extraordinary.

Research for Practice is born from the potential of this combination. In every RfP column, two experts will introduce a short curated selection of papers on a concentrated, practically oriented topic. Want to learn about the latest and greatest developments in operating systems for data-center workloads? RfP will provide an essential crash course from a world authority by describing the trends in this space, selecting a handful of papers to read, and providing motivation and the critical insights behind each.
This approach is designed to allow you to become fluent in exciting topics in computer science research in a weekend afternoon. In addition, ACM has graciously agreed to provide open access to any Research for Practice paper citations available in the ACM Digital Library. Each installment will cover different topics from different volunteer experts, and we intend to cover the entire range of computer science subfields.

This issue of *acmqueue* magazine contains the first installment of Research for Practice. Were you curious about the data-center operating system trends I just mentioned? You’re in luck: Simon Peter has a fantastic selection on this topic, including papers on the interplay between emerging I/O subsystems and the kernel, principles for multicore scalability, and systems possibilities for new secure computing hardware. In addition, Justine Sherry has contributed an exciting selection on network functions virtualization: our networks are getting smarter, aided by increasingly complex in-network software. This allows functionality beyond traditional network “middlebox” operation, including complex routing and policy deployment and cryptographically secure and private packet processing. Both of these selections highlight practical yet principled research papers. We’re especially pleased by how accessible each of our experts has made these otherwise highly technical topics.

Research for Practice is itself an ongoing experiment. We’re inspired by the widespread and growing enthusiasm about computer science research as well as the role ACM,
its members, and the acmqueue readership can play in amplifying this excitement. We welcome your feedback, and please enjoy! —Peter Bailis

DATA CENTERS ARE CHANGING THE WAY WE DESIGN SERVER SYSTEMS

BY SIMON PETER

The growing number of cloud service users and volume of data are putting tremendous pressure on I/O, processing, and integrity. Hardware has kept pace: data-center networks allow servers to transmit and receive millions of requests per second with microsecond delivery latencies. An increasing number of processors multiplies server-processing capacities, and new technologies such as Intel’s Software Guard Extensions (SGX) help keep sensitive data confidential. As a result, operating systems need to provide these new technologies to applications scalably and efficiently.

The following papers introduce thought-provoking OS design paradigms that address each of these trends. First we attack the I/O performance problem. We then introduce a handy software-interface design rule that ensures that constructed software can scale with the number of processors present in data-center servers. Finally, we learn how to protect the integrity of sensitive data, even from access by the cloud operator. We conclude with an outlook on how these paradigms enable an ecosystem of execution
environments for data-center applications.

**Dealing with the data deluge**

https://www.usenix.org/conference/osdi14/technical-sessions/presentation/peter

https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay

These papers discuss the design of operating systems that provide high I/O performance to request-intensive server applications. The authors find that the complexity of monolithic OS kernels is the biggest barrier to server I/O performance and remedy the situation by introducing an I/O model that bypasses the kernel in the common case without losing any of its protection guarantees. Both papers split the OS into a control and a data plane: A kernel-level control plane carries out access control and resource management, while a user-level data plane is responsible for fast I/O mechanisms.

The papers differ in how network I/O policy is enforced. Arrakis reaches for utmost performance by relying on hardware to enforce per-application maximum I/O rates and
allowed communication peers. IX trades performance for software control over network I/O, thus allowing the precise enforcement of the I/O behavior of a particular network protocol, such as TCP congestion control.

Both OS models do extremely well supporting an emerging bit of cloud infrastructure: containers. Containers bundle all required components of an application into a manageable unit. Arrakis and IX empower containers to use all I/O capabilities of the underlying server hardware without the overhead of a monolithic OS kernel.

Keeping all processors busy

Many OS researchers have worked on the problem of using an increasing number of processor cores to handle growing workload demands. Manually identifying and working around scalability bottlenecks caused by shared resource contention in implementations has often been the answer. This paper asks a different question: Can APIs have an impact on software scalability? The surprising answer is that the impact is not only profound, but also fundamental.

The paper distills its insight into a simple yet effective software-development rule: whenever interface operations commute, they can be implemented in a way that scales. The authors provide a tool that helps developers apply the rule by generating test cases that find scalability bottlenecks.
in commutative API implementations. They use the tool to evaluate the POSIX API and point out where the API has the ability to scale but its OS implementation hits a bottleneck. They employ the results to develop a new OS that is practically free of scalability bottlenecks.

The scalable commutativity rule applies not just to the design of operating systems, but also to any multicore software system. It should thus be part of the toolkit of any multicore application developer.

Keeping sensitive data confidential

https://www.usenix.org/conference/osdi14/technical-sessions/presentation/baumann

Customers trust their cloud providers not to expose any of their data—a tall order, given the staggering complexity of the cloud hardware/software platform. Bugs may easily compromise sensitive data. This paper introduces Haven, a software system that protects the integrity of a program and its data from the entire cloud-execution platform, except for a small trusted block of firmware.

To achieve this, Haven uses the recently introduced Intel SGX technology to develop a non-hierarchical OS security model that allows applications to run in a secure region of memory that is protected from outside access, including privileged software such as OS kernels and hypervisors. To
support execution on top of an untrusted OS kernel, Haven introduces a mutually distrusting kernel interface that applications access via a user-level library that provides the Windows API.

Haven introduces a new way of protecting data confidentiality. While previous attempts use encryption techniques such as homomorphic encryption to compute on encrypted data in limited cases, Haven relies on hardware-protection technology to address the problem in a more general way.

**An ecosystem of application execution environments**

These papers establish a new baseline for data-center OS design. Not the traditional Unix model where processes run on top of a shared kernel invoked via POSIX system calls, but protected software containers using scalable library invocations that map directly to hardware mechanisms allow applications to break out of existing OS performance and protection limitations.

This new OS design has the potential to enable an ecosystem of library execution environments that support applications in various ways. For example, a fast library network stack may be linked to a web server to improve its webpage delivery latency and throughput. A Haven-like system call library may be linked to protect the integrity of confidential data held by the application. Finally, a scalable storage stack may be linked to a database to allow it to keep pace with the throughput offered by parallel flash memory. In many cases, these libraries can improve application
execution transparently. Together, these new execution environments have the potential to allow applications to match the performance and integrity demands of current and future data-center workloads.

NFV AND MIDDLEBOXES

BY JUSTINE SHERRY

We usually think of networks as performing only one task: delivering packets from sender to receiver. Today’s networks, however, do a lot more by deploying special-purpose middleboxes to inspect and transform packets, usually to improve performance or security. A middlebox may scan a connection for malicious behavior, compress data to provide better performance on low-resource mobile devices, or serve content from a cache inside the network to reduce bandwidth costs. Both industry and research sources have recently begun to refer to the features implemented by middleboxes as “network functions.” Popular open-source network functions include the Snort Intrusion Detection System$^3$ and the Squid Web Proxy.$^4$

To deploy a new network function, a network administrator traditionally purchases a specialized, fixed-function hardware device (the middlebox) implementing, for example, intrusion detection or caching, and physically installs the device at a chokepoint in the network such that all traffic entering or exiting the network must pass through
it. Alternatively, an administrator might use an off-the-shelf server as a middlebox, installing software such as Snort, Squid, or a proprietary software package, and then routing traffic through the server at a chokepoint in the network.

NFV (network functions virtualization) is a new movement in networking that takes the software-based approach to an extreme. The NFV ISG (industry specification group) envisions a future in which all middlebox functionality is implemented in software.² Network administrators will deploy a server or cluster of servers dedicated to network functions, and network virtualization software will automatically route traffic through various network functions.

NFV promises many benefits for network administrators. It reduces costs by moving from special-purpose to general-purpose hardware, makes upgrades as easy as a software patch, offers the opportunity to scale on demand, and promises more efficient installations with multiple network functions potentially sharing a single server, leaving few resources wasted. NFV has tremendous momentum in the networking community—the NFV working group has more than 200 industrial members¹—but is in its infancy and was founded only in late 2012.

Here we present three highlights from the research community on middleboxes and NFV, and conclude by discussing some of the challenges and opportunities that NFV presents for application developers.
What capabilities do network functions implement?

Though it predates NFV by about a decade, this article remains a nice summary of the features for which middleboxes are commonly deployed. The document could have gone into more depth about application-layer behaviors such as exfiltration detection or intrusion detection—increasingly common in today’s corporate networks—but these behaviors are more common today than they were in 2002 when the article was written. Nonetheless, it remains the most comprehensive survey of middlebox functionality to date, and most of the features it describes remain in common use.

What does an NFV-managed network look like?
http://dl.acm.org/citation.cfm?id=2815423

This article provides the cleanest vision for an NFV-managed cluster to date. The authors describe a system called E2, which automatically schedules and configures network functions on a cluster of general-purpose servers. E2 allows a network administrator to specify a “configuration” (e.g., all traffic on port 80 should be routed through this HTTP proxy, all traffic to this subnet should be processed by an IDS),
and the framework will automatically instantiate software instances and a routing configuration to ensure that the policy is met. E2 is conceptually similar to cloud frameworks such as OpenStack or RightScale but in practice involves many different technical challenges, including scheduling to ensure that bandwidth is not overutilized, ensuring low latency, and enabling efficient communication and “chaining” between network functions.

Can I control how network functions process my traffic?
http://queue.acm.org/rfp/vol14iss2.html
Sherry, J., et al. 2015. BlindBox: deep packet inspection over encrypted traffic. ACM SIGCOMM.
http://queue.acm.org/rfp/vol14iss2.html

Today, application developers have no way of controlling which network functions process their traffic, short of making a phone call to their network administrators. Nonetheless, developers may have concerns about inspection or modification of traffic sent by their applications—especially with regard to privacy. Hence, many developers choose to encrypt their entire connection (e.g., using SSL/TLS). While this preserves privacy, it also prevents all benefits of middlebox processing. These two articles propose new cryptographic protocols, mcTLS and BlindBox, that would let application developers allow certain middlebox operations but restrict others. The two articles
propose very different approaches to the same problem and are worth reading side by side.

**What does NFV mean for application developers?**

As NFV makes the deployment and configuration of network functions/middleboxes easier, application developers can expect to see increasingly complex behavior from their networks. While this capability for complex behavior retains some of the old challenges of middleboxes (e.g., privacy), it also introduces a huge new opportunity for application developers. NFV enables application developers to run and execute their code not only on end hosts they maintain, but also in the network itself.

For example, a developer who designs a custom load-balancing filter based on a unique service architecture might write the new code to run on the load balancer itself. A web service may implement a custom cache to serve encrypted content to its users, deploying the in-network cache within its customers’ ISPs within virtual machines hosted in the provider’s infrastructure. With the ability to execute *arbitrary* code in the network—and smart routing and scheduling to ensure that the right traffic receives such processing—NFV opens an entirely new programming platform for developers. The next big app store may be for features deployed within data-center networks, ISPs, or even on home routers.

**References**

4. Squid; http://www.squid-cache.org/

**LOVE IT, HATE IT? LET US KNOW** feedback@queue.acm.org

Peter Bailis will join Stanford University as an assistant professor of computer science, after spending the academic year visiting MIT CSAIL. He received a Ph.D. in computer science from UC Berkeley in 2015 and an A.B. in computer science from Harvard in 2011. His research in the Future Data Systems group (http://futuredata.stanford.edu/) focuses on the design and implementation of next-generation data-intensive systems.

Justine Sherry is a doctoral candidate at UC Berkeley. Her interests are in computer networking; her work includes middleboxes, networked systems, measurement, cloud computing, and congestion control. Sherry’s dissertation focuses on new opportunities and challenges arising from the deployment of middleboxes—such as firewalls and proxies—as services offered by clouds and ISPs. She received an M.S. from UC Berkeley in 2012 and a B.S. and B.A. from the University of Washington in 2010. She is a National Science Foundation Graduate Research Fellow, has won paper awards from both Usenix NSDI and ACM SIGCOMM, and is always on the lookout for a great cappuccino.
Simon Peter is an assistant professor at the University of Texas at Austin, where he leads research in operating systems and networks. He received a Ph.D. in computer science from ETH Zurich in 2012 and an M.S. in computer science from the Carl-von-Ossietzky University in Oldenburg, Germany, in 2006. Before joining UT Austin in 2016, he was a research associate at the University of Washington from 2012-2016. For his work on high-I/O-performance operating systems, he received the Jay Lepreau best paper award (2014) and the Madrona prize (2014). He has conducted further award-winning systems research at various locations, including MSR SVC and Cambridge, Intel Labs, and UC Riverside.

Copyright © 2016 held by owner/author. Publication rights licensed to ACM.
Distributed systems pose unique challenges for software developers. Reasoning about concurrent activities of system nodes and even understanding the system’s communication topology can be difficult. A standard approach to gaining insight into system activity is to analyze system logs. Unfortunately, this can be a tedious and complex process. This article looks at several key features and debugging challenges that differentiate distributed systems from other kinds of software. The article presents several promising tools and ongoing research to help resolve these challenges.

DISTRIBUTED-SYSTEM FEATURES AND CHALLENGES
Distributed systems differ from single-machine programs in ways that are simultaneously positive in providing systems with special capabilities, and negative in presenting software-development and operational challenges.

Heterogeneity
A distributed system’s nodes may include mobile phones, laptops, server-class machines, and more. This hardware and software diversity in node resources and network connectivity can make a distributed system more robust,
but this heterogeneity forces developers to manage compatibility during both development and debugging.

**Concurrency**

Simultaneous operation by multiple nodes leads to concurrency, which can make a distributed system outperform a centralized system. However, concurrency may introduce race conditions and deadlocks, which are notoriously difficult to diagnose and debug. Additionally, networks introduce packet delay and loss, exacerbating the issues of understanding and debugging concurrency.

**Distributed state**

Distributing system state across multiple nodes can remove a central point of failure and improve scalability, but distributed state requires intricate node coordination to synchronize state across nodes—for example, nodes must ensure their local states are consistent. Potential inconsistencies are prevented by distributed algorithms, such as those that guarantee a particular flavor of data consistency and cache coherence. Developers may find it difficult, or even impossible, to reconstruct the global state of the system when it is distributed on many nodes. This complicates bug diagnosis and validation.

**Partial failures**

The distribution of state and responsibility allows distributed systems to be robust and survive a variety of failures. For example, Google’s Spanner system can survive failures of
Achieving such fault tolerance, however, requires developers to reason through complex failure modes. For most distributed systems, fault tolerance cannot be an afterthought; the systems must be designed to deal with failures. Such failure resiliency is complex to design and difficult to test.

EXISTING APPROACHES

What follows is an overview of seven approaches designed to help software engineers validate and debug distributed systems.

Testing

A test suite exercises a specific set of executions to ensure that they behave properly. Most testing of distributed systems is done using manually written tests, typically introduced in response to failures and then minimized. Testing is an effective way to detect errors. However, since testing exercises a limited number of executions, it can never guarantee to reveal all errors.

Model checking

Model checking is exhaustive testing, typically up to a certain bound (number of messages or steps in an execution). Symbolic model checking represents and explores possible executions mathematically; explicit-state model checking is more practical because it actually runs the program, controlling its executions rather than attempting to abstract it. MoDist performs black-box model checking, permuting
message sequences and changing the execution speed of a process relative to other processes in the system. MaceMC is a white-box technique that achieves speedups by adding programming-language support for model checking. Common problems of all model-checking tools are scalability and environmental modeling, so they rarely achieve a guarantee.

Theorem proving
Theorem proving can, in principle, prove a distributed system to be free of defects. Amazon uses TLA+ to verify its distributed systems. Two recent systems can construct a verified distributed-system implementation. Verdi uses the Coq tool, whose expressive type system makes type checking equivalent to theorem proving, thanks to the Curry-Howard isomorphism; the Coq specification is then compiled into an OCaml implementation of the distributed system. In contrast, IronFleet uses TLA and Hoare-logic verification to similarly produce a verified implementation of a distributed system. The enormous effort needed to use these tools makes them most appropriate for new implementations of small, critical cores. Other techniques are needed for existing distributed systems.

Record and replay
Record and replay captures a single execution of the system so that this execution can be later replayed or analyzed. This is especially useful when debugging nondeterministic behaviors. A record-and-replay tool such as Friday or
D3S captures all nondeterministic events so that an execution can be reproduced exactly. Recording a complex execution, however, may be prohibitively expensive and may change the behavior of the underlying system.

**Tracing**

Tracing tracks the flow of data through a system, even across applications and protocols such as a database, web server, domain-name server, load balancer, or virtual private network protocol. For example, pivot tracing dynamically instruments Java-based systems to collect user-defined metrics at different points in the system and collates the resulting data to provide an inter-component view of the metrics over multiple executions. Dapper is a lower-level tracing system used at Google to trace infrastructure services. Tracing is more efficient than record and replay because it focuses on a specific subset of the data, but it requires instrumenting applications and protocols to properly forward, without consuming, the tracing metadata.

**Log analysis**

Log analysis is an even lighter-weight approach that works with systems that cannot be modified. It is a common black-box approach in which a system’s console logs, debug logs, and other log sources are used to understand the system. For example, Xu et al. applied machine learning to logs to detect anomalies in Google infrastructure services. Detailed logs from realistic systems contain a great deal of valuable detail, but they tend to be so large that they
are overwhelming to programmers, who as a result cannot directly benefit from them.

**Visualization**

The complexity of distributed systems has inspired work on visualization of such systems to make them more transparent to developers. For example, Theia displays a visual signature that summarizes various aspects of a Hadoop execution, such as the execution’s resource utilization. These signatures can be used to spot anomalies and to compare executions. Tools such as Theia provide high-level summaries of a system’s behavior. They do not, however, help a developer understand the underlying communication pattern in the system, including the distributed ordering of messages.

**VISUALIZING DISTRIBUTED-SYSTEM EXECUTIONS**

As noted above, the ability to visualize distributed-system executions can help developers understand and debug their distributed systems. ShiViz is such a visualization tool, displaying distributed-system executions as interactive time-space diagrams that explicitly capture distributed ordering of messages and events in the system. This diagram reproduces the events and interactions captured in the execution log, making the ordering information explicit through a concise visualization. A developer can expand, collapse, and hide parts of the diagram, as well as search for particular interaction patterns. ShiViz is freely available as a browser application; any developer can visualize a log, without installing software or sending the log over the network.
To provide a rich and accurate visualization of a distributed system’s execution, ShiViz displays the *happens-before relation*. Given event \( e \) at node \( n \), the happens-before relation indicates all the events that logically precede \( e \). Other events might have already occurred at other nodes according to wall-clock time, but node \( n \) cannot tell whether those other events happened before or after \( e \), and they do not affect the behavior of \( e \). This partial order can rule out which events do not cause others, identify concurrent events, and help developers mentally replay parts of the execution.

Figure 1 illustrates an execution of the two-phase commit protocol with one transaction manager and two replicas.\(^1\)

**FIGURE 1: TIME-SPACE DIAGRAM OF AN EXECUTION WITH THREE NODES**
This time-space diagram is a visualization of the underlying happens-before partial order, showing an execution with three nodes. Lines with arrows denote the partial ordering of events, each of which has an associated vector timestamp in brackets. [See timestamp sidebar on next page.]

Figure 2 shows a screenshot of ShiViz visualizing an execution of a distributed data-store system called Voldemort. In the middle of the screen is the time-space diagram, with time flowing from top to bottom. The colored boxes at the top represent nodes, and the vertical lines below them are the node timelines. Circles on each node’s timeline represent events executed by that node. Edges connect events, representing the recorded happens-before relation: an event that is higher in the graph happened before an event positioned lower in the graph that it is connected to.

**FIGURE 2: A SHIVIZ SCREENSHOT**
Distributed systems

to via a downward path. ShiViz augments the time-space diagram with operations to help developers explore distributed-system executions and corresponding logs. Figure 2 details some of these operations.

**Distributed timestamps**

A typical distributed-system log does not contain enough information to regenerate the happens-before relation, and this is one reason that distributed-system logs are so hard to interpret. ShiViz relies on logs that have been enhanced by another tool, ShiVector, to include *vector clock timestamps* that capture the happens-before relation between events. Each node $\alpha$ maintains a vector of logical clocks, one clock for each node in the distributed system, including itself. $\alpha$'s $i$th clock is a lower bound on the current logical time at node $i$. The node $\alpha$ increments the $\alpha$th component of its vector clock each time it performs a local action or sends or receives a message. Each message contains the sending node's current vector clock; upon message receipt, the receiving node updates its vector clock to the elementwise maximum of its local and received timestamps.

ShiVector is a lightweight instrumentation tool that augments the information already logged by a distributed system with the partial ordering information encoded as vector clocks. ShiVector interposes on communication and logging channels at each node in the system to add vector clock timestamps to every logged event.

ShiViz parses ShiVector-augmented logs to determine, for each event: (1) the node that executed the event; (2) the vector timestamp of the event; and (3) the event's description.

ShiViz permits a user to customize the parsing of logs using regular expressions, which can be used to associate additional information, or *fields*, with each event.
UNDERSTANDING DISTRIBUTED-SYSTEM EXECUTIONS

ShiViz helps developers [1] to understand the relative ordering of events and the likely chains of causality between events, which is important for debugging concurrent behavior; [2] to query for certain events and interaction patterns between hosts; and [3] to identify structural similarities and differences between pairs and groups of executions. The time-space diagram representation supports the first goal by visualizing event ordering and communication. The next section describes two search operations that support the second goal, and operations over multiple executions that correspond to the third goal.

**Keyword search and structured search operations**

ShiViz implements two kinds of search operations: keyword and structured. Both types are accessible to the developer through the top search bar (see figure 2).

Keyword search allows a developer to highlight all events in the diagram that contain a field matching a query. For example, searching for `send` will highlight all events in the diagram that have a field whose value is `send`. The results can be further constrained with field identifiers and regular expressions. For example, the query `node=alice && priority=CRITICAL*` will highlight only events at the `alice` node with a priority field matching the regular expression `CRITICAL*`.

In a structured search, a user queries ShiViz for any set of events related through a particular ordering pattern, and ShiViz highlights the sections of the diagram (events
and their interconnections) that match this pattern. ShiViz includes several predefined patterns:

- **Request-response.** A source node sends a request and the destination node sends back a response.
- **Broadcast.** A node sends a message to most other nodes in the system.
- **Gather.** A node receives a message from most other nodes.

A user can also compose a custom pattern consisting of nodes, node events, and connections between events representing a partial order. Figure 3 shows such a custom pattern, depicting three nodes communicating in a ring: node 1 communicates only with node 2; node 2 with node 3; and node 3 with node 1. Drawing this pattern allows the user to search for all instances of this three-node ring communication in the execution. ShiViz automatically translates the drawn pattern into a textual representation (see search bar at the top), and it is possible to edit.

**FIGURE 3:** STRUCTURED SEARCH FEATURE

[Image of the structured search feature]

- **Search for a pre-defined structure:** Select one of the options below to find the specified structure.
  - REQUEST-RESPONSE
  - BROADCAST
  - GATHER

- **Search for a custom structure:** Draw a graph structure below (add processes, events, and edges to add inter-event edges).
copy, and paste the textual representation directly. The structured search feature allows users to express custom communication patterns between events and to query an execution for instances of the specified pattern. The presence or absence of queried subgraphs at particular points in an execution can help users detect anomalous behavior, aiding them in their debugging efforts.

**Comparing executions**

ShiViz can help users understand multiple executions of a system. When ShiViz parses multiple executions, the user can choose between viewing executions individually or pairwise.

In the pairwise view, a user can compare the two executions further by highlighting their differences. When enabled, the nodes are compared by name. For nodes present in both executions, ShiViz compares their events one by one by comparing the corresponding event descriptions. Nodes or events in one execution that do not appear in the other are redrawn as rhombuses.

Figure 4 illustrates this pairwise comparison on a log of the two-phase commit protocol. The two selected events in the figure explain the difference between these two executions: the two-phase commit successfully commits a transaction in the left execution, but aborts a transaction in the right execution.

The explicit highlighting of differences provides users with fast detection of anomalous events or points where
the two executions diverge. The search features described earlier can be applied in the pairwise view to help developers detect specific unifying or distinguishing features across traces, allowing them to design and test their systems more effectively.

**Clustering executions**

To help manage many executions, ShiViz supports grouping executions into clusters. A user can cluster by the number of nodes or by comparison to a base execution, using as a distance metric the differencing mechanism described earlier. Cluster results are presented as distinct groups of listed execution names.

Execution clusters aid in the inspection and comparison of multiple executions by providing an overview of all
executions at once. Users can quickly scan through cluster results to see how executions are alike or different, based on the groups into which they are sorted. Clustering also helps users pinpoint executions of interest by allowing them to inspect a subset of executions matching a desired measure. This subset can be further narrowed by performing a keyword search or a structured search on top of the clustering results. Execution names among clusters are highlighted if their corresponding graphs contain instances matching the user’s search query.

ShiViz helps developers visualize the event order, search for communication patterns, and identify potential event causality. This can help developers reason about the concurrency of events in an execution’s distributed system state, and distributed failure modes, as well as formulate hypotheses about system behavior and verify them via execution visualizations. Meanwhile, the generality of logging makes ShiVector and ShiViz broadly applicable to systems deployed on a wide range of devices.

ShiViz has some limitations. ShiViz surfaces low-level ordering information, which makes it a poor choice for understanding high-level system behavior. The ShiViz visualization is based on logical and not realtime ordering, and cannot be used to study certain performance characteristics. The ShiViz tool is implemented as a client-side-only browser application, making it portable and appropriate for analyzing sensitive log data. This design choice, however, also limits its scalability.

ShiViz is an open-source tool with an online deployment
Watch a video demonstrating key ShiViz features at http://bestchai.bitbucket.org/shiviz-demo.

Acknowledgments
We thank Perry Liu and Albert Xing, who helped develop ShiViz; Jenny Abrahamson, who developed the initial ShiVector and ShiViz prototypes; and Donald Acton and Colin Scott, who helped evaluate ShiViz. This work is supported by NSERC USRA, the NSERC Discovery grant, and the National Science Foundation under grants CCF-1453474 and CNS-1513055. This material is based on research sponsored by DARPA under agreement number FA8750-12-2-0107. The U.S. government is authorized to reproduce and distribute reprints for governmental purposes, notwithstanding any copyright notices thereon.

References


11. Project Voldemort; http://www.project-voldemort.com/voldemort/


Ivan Beschastnikh works on improving the design, implementation, and operation of complex systems. He is an assistant professor in the department of computer science at the University of British Columbia, where he leads a team of students on projects that span distributed systems, software engineering, security, and networks, with a particular focus on program analysis. More information is available at his homepage: http://www.cs.ubc.ca/~bestchail.

Patty Wang earned her bachelor’s degree in computer science and mathematics from the University of British Columbia. She has explored approaches to helping developers understand and compare multiple distributed executions, focusing on summarizing similarities and differences across traces.

Yuriy Brun works on automating system building and creating self-adaptive systems. He is an assistant professor at the University of Massachusetts, Amherst. He received his PhD from the University of Southern California in 2008, and then spent three years as a postdoctoral fellow at the University of Washington. He has been recognized with a CAREER award from the National Science Foundation, a Microsoft Research SEIF (Software Engineering Innovation Foundation) award, a Google Faculty Research award, and an IEEE TCSC (Technical Committee on Scalable Computing) Young Achiever in Scalable Computing award. More information is available at his homepage: http://people.cs.umass.edu/~brun/.
Michael D. Ernst researches ways to make software more reliable, more secure, and easier (and more fun!) to produce. His primary technical interests are in software engineering, programming languages, type theory, security, program analysis, bug prediction, testing, and verification. Ernst is an ACM Fellow and received the inaugural John Backus Award, the NSF CAREER Award, 10 best-paper awards, and other honors. More information is available at his homepage: http://homes.cs.washington.edu/~mernst/.

Copyright © 2016 held by owner/author. Publication rights licensed to ACM.

Applicative 2016
New York City | June 1 - 2, 2016

Applicative 2016 will bring together practitioners and researchers to share the latest emerging technologies and trends in software development, presented in two tracks:

Application Development
reactive programming, micro-services, single-page application frameworks, and other approaches

Systems Software
for systems-level practitioners involved in the design, implementation and support of novel technologies and low-level software

applicative.acm.org
Should You Upload or Ship Big Data to the Cloud?

THE ACCEPTED WISDOM DOES NOT ALWAYS HOLD TRUE

It is accepted wisdom that when the data you wish to move into the cloud is at terabyte scale and beyond, you are better off shipping it to the cloud provider, rather than uploading it. This article takes an analytical look at how shipping and uploading strategies compare, the various factors on which they depend, and under what circumstances you are better off shipping rather than uploading data, and vice versa. Such an analytical determination is important to make, given the increasing availability of gigabit-speed Internet connections, along with the explosive growth in data-transfer speeds supported by newer editions of drive interfaces such as SAS and PCI Express. As this article reveals, the aforementioned “accepted wisdom” does not always hold true, and there are well-reasoned, practical recommendations for uploading versus shipping data to the cloud.

SACHIN DATE, E-EMPHASYS TECHNOLOGIES
Here are a few key insights to consider when deciding whether to upload or ship:

- A direct upload of big data to the cloud can require an unacceptable amount of time, even over Internet connections of 100-Mbps (megabits per second) and faster. A convenient workaround has been to copy the data to storage tapes or hard drives and ship it to the cloud data center.

- With the increasing availability of affordable, optical fiber-based Internet connections, however, shipping the data via drives becomes quickly unattractive from the point of view of both cost and speed of transfer.

- Shipping big data is realistic only if you can copy the data into (and out of) the storage appliance at very high speeds and you have a high-capacity, reusable storage appliance at your disposal. In this case, the shipping strategy can easily beat even optical fiber-based data upload on speed, provided the size of the data is above a certain threshold value.

- For a given value of drive-to-drive data-transfer speed, this threshold data size (beyond which shipping the data to the cloud becomes faster than uploading it) grows with every Mbps increase in the available upload speed. This growth continues up to a certain threshold upload speed. If your ISP provides an upload speed greater or equal to this threshold speed, uploading the data will always be faster than shipping it to the cloud, no matter how big the data is.

Suppose you want to upload your video collection into the public cloud; or let’s say your company wishes to migrate its data from a private data center to a public cloud, or move it from one data center to another. In a way it doesn’t matter
what your profile is. Given the explosion in the amount of digital information that both individuals and enterprises have to deal with, the prospect of moving big data from one place to another over the Internet is closer than you might think.

To illustrate, let’s say you have 1 TB of business data to migrate to cloud storage from your self-managed data center. You are signed up with a business plan with your ISP that guarantees you an upload speed of 50 Mbps and a download speed of 10 times as much. All you need to do is announce a short system-downtime window and begin hauling your data up to the cloud. Right?

Not quite.

For starters, you will need a whopping 47 hours to finish uploading 1 TB of data at a speed of 50 Mbps—and that’s assuming your connection never drops or slows down.

If you upgrade to a faster—say, 100 Mbps—upload plan, you can finish the job in one day. But what if you have 2 TB of content to upload, or 4 TB, or 10 TB? Even at a 100-Mbps sustained data-transfer rate, you will need a mind-boggling 233 hours to move 10 TB of content!

As you can see, conventional wisdom breaks down at terabyte and petabyte scales. It’s necessary to look at alternative, nonobvious ways of dealing with data of this magnitude.

Here are two such alternatives available today for moving big data:

- Copy the data locally to a storage appliance such as LTO (linear tape open) tape, HDD (hard-disk drive), or SSD (solid-state drive), and ship it to the cloud provider. For
convenience, let’s call this strategy “Ship It!”

- Perform a cloud-to-cloud transfer of content over the Internet using APIs (application programming interfaces) available from both the source and destination cloud providers. Let’s call this strategy “Transfer It!”

This article compares these alternatives, with respect to time and cost, to the baseline technique of uploading the data to the cloud server using an Internet connection. This baseline technique is called “Upload It!” for short.

A REAL-LIFE SCENARIO
Suppose you want to upload your content into, purely for the sake of illustration, the Amazon S3 (Simple Storage Service) cloud, specifically its data center in Oregon. This could well be any other cloud-storage service provided by players in this space such as (but not limited to) Microsoft, Google, Rackspace, and IBM. Also, let’s assume that your private data center is located in Kansas City, Missouri, which happens to be roughly geographically equidistant from Amazon’s data centers located in the eastern and western United States.

Kansas City is also one of the few places where a gigabit-speed optical-fiber service is available in the United States. In this case, it’s offered by Google Fiber.

As of November 2015, Google Fiber offers one of the highest speeds that an ISP can provide in the United States: 1 Gbps (gigabit per second), for both upload and download. Short of having access to a leased Gigabit Ethernet line, an optical fiber-based Internet service is a really, really fast way
to shove bits up and down Internet pipes anywhere in the world.

Assuming an average sustained upload speed of 800 Mbps on such a fiber-based connection,\(^{13}\) (i.e., 80 percent of its advertised theoretical maximum speed of 1 Gbps), 1 TB of data will require almost three hours to upload from Kansas City to S3 storage in Oregon. This is actually pretty quick (assuming, of course, your connection never slows down). Moreover, as the size of the data increases, the upload time increases in the same ratio: 20 TB requires 2½ days to upload, 50 TB requires almost a week to upload, and 100 TB requires twice that long. At the other end of the scale, a half a petabyte of data requires two months to upload. Uploading one petabyte at 800 Mbps should keep you going for four months.

It’s time to consider an alternative.

SHIP IT!
That alternative is copying the data to a storage appliance and shipping the appliance to the data center, at which end the data is copied to cloud storage. This is the Ship It! strategy. Under what circumstances is this a viable alternative to uploading the data directly into the cloud?

The Mathematics of Shipping Data
When data is read out from a drive, it travels from the physical drive hardware (e.g., the HDD platter) to the onboard disk controller (the electronic circuitry on the drive). From there the data travels to the host controller (a.k.a. the host bus adapter, a.k.a. the interface card) and finally to
the host system (e.g., the computer with which the drive is interfaced). When data is written to the drive, it follows the reverse route.

When data is copied from a server to a storage appliance (or vice versa), the data has to travel through an additional physical layer, such as an Ethernet or USB connection existing between the server and the storage appliance.

Figure 1 is a simplified view of the data flow when copying data to a storage appliance. The direction of data flow shown in the figure is conceptually reversed when the data is copied out from the storage appliance to the cloud server.

Note that often the storage appliance may be nothing more than a single hard drive, in which case the data flow from the server to this drive is basically along the dotted line in the figure.

Given this data flow, a simple way to express the time needed to transfer the data to the cloud using the Ship It! strategy is shown in equation 1:

\[
(\text{Transfer Time})_{\text{hours}} = \frac{V_{\text{content}}}{3600 \times \text{Speed}_{\text{copyIn}}} + T_{\text{transit}} + \frac{V_{\text{content}}}{3600 \times \text{Speed}_{\text{copyOut}}} + T_{\text{overhead}}
\]

...(Eq 1)

Where:

- \(V_{\text{content}}\) is the volume of data to be transferred in megabytes (MB).
- \(\text{Speed}_{\text{copyIn}}\) is the \textit{sustained} rate in MBps (megabytes per second) at which data is copied from the source drives to the
storage appliance. This speed is essentially the minimum of three speeds: (1) the speed at which the controller reads data out of the source drive and transfers it to the host computer with which it interfaces; (2) the speed at which the storage
Appliance’s controller receives data from its interfaced host and writes it into the storage appliance; and (3) the speed of data transfer between the two hosts. For example, if the two hosts are connected over a Gigabit Ethernet or a Fibre Channel connection, and the storage appliance is capable of writing data at 600 MBps, but the source drive and its controller can emit data at only 20 MBps, then the effective copy-in speed can be at most 20 MBps.

\[ \text{Speed}_{\text{copyOut}} \] is similarly the sustained rate in MBps at which data is copied out of the storage appliance and written into cloud storage.

\[ T_{\text{transit}} \] is the transit time for the shipment via the courier service from source to destination in hours.

\[ T_{\text{overhead}} \] is the overhead time in hours. This can include the time required to buy the storage devices (e.g., tapes), set them up for data transfer, pack and create the shipment, and drop it off at the shipper’s location. At the receiving end, it includes the time needed to process the shipment received from the shipper, store it temporarily, unpack it, and set it up for data transfer.

The use of sustained data-transfer rates

Storage devices come in a variety of types such as HDD, SSD, and LTO. Each type is available in different configurations such as a RAID (redundant array of independent disks) of HDDs or SSDs, or an HDD-SSD combination where one or more SSDs are used as a fast read-ahead cache for the HDD array. There are also many different data-transfer interfaces such as SCSI (Small Computer System Interface),
SATA (Serial AT Attachment), SAS (Serial Attached SCSI), USB (Universal Serial Bus), PCI (Peripheral Component Interconnect) Express, Thunderbolt, etc. Each of these interfaces supports a different theoretical maximum data-transfer speed.

Figure 2 lists the data-transfer speeds supported by recent versions of some of these controller interfaces.

The effective copy-in/copy-out speed while copying data to/from a storage appliance depends on a number of factors:

- **Type of drive.** For example, SSDs are usually faster than HDDs partly because of the absence of any moving parts. Among HDDs, higher-RPM drives can exhibit lower seek times than lower-RPM drives. Similarly, higher areal-density (bits per surface area) drives can lead to higher data-transfer rates.

- **Configuration of the drive.** Speeds are affected by, for example, single disk versus an array of redundant disks, and the presence or absence of read-ahead caches on the drive.

- **Location of the data on the drive.** If the drive is fragmented (particularly applicable to HDDs), it can take

---

### Figure 2: Data Transfer Speeds Supported by Various Interfaces

<table>
<thead>
<tr>
<th>Interface Type</th>
<th>Data Transfer Speed (GBPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SATA Revision 3</td>
<td>6[^7]</td>
</tr>
<tr>
<td>SAS-3</td>
<td>12[^8]</td>
</tr>
<tr>
<td>SuperSpeed USB (USB 3.0)</td>
<td>10[^9]</td>
</tr>
<tr>
<td>PCI Express version 4</td>
<td>15.754 (single data lane), to 252.064 (16 data lanes)^[^14]</td>
</tr>
<tr>
<td>Thunderbolt 2</td>
<td>20[^1]</td>
</tr>
</tbody>
</table>

[^7]: Reference to speed
[^8]: Reference to speed
[^9]: Reference to speed
[^10]: Reference to speed
[^11]: Reference to speed
[^12]: Reference to speed
[^13]: Reference to speed
[^14]: Reference to speed
longer to read data from and write data to it. Similarly, on HDD platters, data located near the periphery of the platter will be read faster than data located near the spindle. This is because the linear speed of the platter near the periphery is much higher than near the spindle.

- **Type of data-transfer interface.** SAS-3 versus SATA Revision 3, for example, can make a difference in speeds.
- **Compression and encryption.** Compression and/or encryption at source and decompression and/or decryption at the destination reduce the effective data-transfer rate.

Because of these factors, the effective *sustained* copy-in or copy-out rate is likely to be much different (usually much less) than the burst read/write rate of either the source drive and its interface or the destination drive and its controller interface.

With these considerations in mind, let’s run some numbers through equation 1, considering the following scenario. You decide to use LTO-6 tapes for copying data. An LTO-6 cartridge can store 2.5 TB of data in uncompressed form.\(^1\)\(^8\) LTO-6 supports an uncompressed read/write data speed of 160 MBps.\(^1\)\(^9\) Let’s make an important simplifying assumption that both the source drive and the destination cloud storage can match the 160-MBps transfer speed of the LTO-6 tape drive (i.e., \(\text{Speed}_{\text{copyIn}} = \text{Speed}_{\text{copyOut}} = 160 \text{ MBps}\)). You choose the overnight shipping option and the shipper requires 16 hours to deliver the shipment (i.e., \(T_{\text{transit}} = 16 \text{ hours}\)). Finally, let’s factor in 48 hours of overhead time (i.e., \(T_{\text{overhead}} = 48 \text{ hours}\)).
Plugging these values into equation 1 and plotting the data-transfer time versus data size using the Ship It! strategy produces the maroon line in figure 3. For the sake of comparison, the blue line shows the data-transfer time of the Upload It! strategy using a fiber-based Internet connection running at 800-Mbps sustained upload rate. The figure shows comparative growth in data-transfer time between uploading at 800 Mbps versus copying it to LTO-6 tapes and shipping it overnight.

Equation 1 shows that a significant amount of time in the Ship It! strategy is spent copying data into and out of the storage appliance. The shipping time is comparatively small and constant (even if you are shipping internationally), while

**FIGURE 3: GROWTH IN DATA TRANSFER TIME, 800 MBPS VS. TAPES**
the drive-to-drive copy-in/copy-out time increases to a very large value as the size of the content being transferred grows. Given this fact, it’s hard to beat a fiber-based connection on raw data-transfer speed, especially when the competing strategy involves copy in/copy out using an LTO-6 tape drive running at 160 MBps.

Often, however, you may not be so lucky as to have access to a 1-Gbps upload link. In most regions of the world, you may get no more than 100 Mbps, if that much, and rarely on a sustained basis. For example, at 100 Mbps, Ship It! has a definite advantage for large data volumes, as in figure 4, which shows comparative growth in data-transfer time.

**FIGURE 4: GROWTH IN DATA TRANSFER TIME, 100 MBPS VS. TAPES**
between uploading at 100 Mbps versus copying the data to LTO-6 tapes and shipping it overnight.

The maroon line in figure 4 represents the transfer time of the Ship It! strategy using LTO-6 tapes, while this time the blue line represents the transfer time of the Upload It! strategy using a 100-Mbps upload link. Shipping the data using LTO-6 tapes is a faster means of getting the data to the cloud than uploading it at 100 Mbps for data volumes as low as four terabytes.

What if you have a much faster means of copying data in and out of the storage appliance? How would that compete with a fiber-based Internet link running at 800 Mbps? With all other parameter values staying the same, and assuming a drive-to-drive copy-in/copy-out speed of 240 MBps (50 percent faster than what LTO-6 can support), the inflection point (i.e., the content size at which the Ship It! strategy becomes faster than the Upload It! strategy at 800 Mbps) is around 132 terabytes. For an even faster drive-to-drive copy-in/copy-out speed of 320 MBps, the inflection point drops sharply to 59 terabytes. That means if the content size is 59 TB or higher, it will be quicker just to ship the data to the cloud provider than to upload it using a fiber-based ISP running at 800 Mbps.

Figure 5 shows the comparative growth in data-transfer time between uploading it at 800 Mbps versus copying it at a 320-MBps transfer rate and shipping it overnight.

Two Key Questions
This analysis brings up the following two questions:
If you know how much data you wish to upload, what is the minimum sustained upload speed your ISP must provide, below which you would be better off shipping the data via overnight courier?

If your ISP has promised you a certain sustained upload speed, beyond what data size will shipping the data be a quicker way of hauling it up to the cloud than uploading it?

Equation 1 can help answer these questions by estimating how long it will take to ship your data to the data center. This quantity is \((\text{Transfer Time})_{\text{hours}}\). Now imagine uploading the same volume of data \((V_{\text{content}} \text{ Megabytes})\), in parallel, over a network link. The question is, what is the

![Figure 5: Growth in data transfer time, 800 Mbps vs. 320 Mbps](image-url)
minimum sustained upload speed needed to finish uploading everything to the data center in the same amount of time as shipping it there. Thus, you just have to express equation 1’s left-hand side (i.e., \((\text{Transfer Time})_{\text{hours}}\)) in terms of [a] the volume of data \(V_{\text{content}} \text{ Megabytes}\) and [b] the required minimum Internet connection speed \((\text{Speed}_{\text{upload}} \text{ Mbps})\).

In other words: 
\[ (\text{Transfer Time})_{\text{hours}} = 8 \times V_{\text{content}} / \text{Speed}_{\text{upload}}. \]

Having made this substitution, let’s continue with the scenario: LTO-6-based data transfer running at 160 MBps, overnight shipping of 16 hours, and 48 hours of overhead time. Also assume there is 1 TB of data to be transferred to the cloud.

The aforementioned substitution reveals that unless the ISP provides a sustained upload speed \(\text{Speed}_{\text{upload}}\) of at least 34.45 Mbps, the data can be transferred faster using a Ship It! strategy that involves an LTO-6 tape-based data transfer running at 160 MBps and a shipping and handling overhead of 64 hours.

Figure 6 shows the relationship between the volume of data to be transferred (in TB) and the minimum sustained ISP upload speed (in Mbps) that is needed to make uploading the data as fast as shipping it to the data center. For very large data sizes, the threshold ISP upload speed becomes less sensitive to the data size and more sensitive to the drive-to-drive copy-in/copy-out speeds with which it is competing.

Now let’s attempt to answer the second question. This time, assume \(\text{Speed}_{\text{upload}}\) (in Mbps) is the maximum sustained upload speed that the ISP can provide. What is the maximum
data size beyond which it will be quicker to ship the data to the data center? Once again, recall that equation 1 helps estimate the time required \((\text{Transfer Time})_{\text{hours}}\) to ship the data to the data center for a given data size \((V_{\text{content}}\text{ MB})\) and drive-to-drive copy-in/copy-out speeds. If you were instead to upload \(V_{\text{content}}\text{ MB} \text{ at } Speed_{\text{upload}}\text{ Mbps}\) over a network link, you would need \(8 \times \frac{V_{\text{content}}}{Speed_{\text{upload}}}\) hours. At a certain threshold value of \(V_{\text{content}}\), these two transfer times (shipping versus upload) will become equal. Equation 1 can be rearranged to express this threshold data size:
Figure 7 shows the relationship between this threshold data size and the available sustained upload speed from the ISP for different values of drive-to-drive copy-in/copy-out speeds. Equation 2 also shows that, for a given value of drive-to-drive copy-in/copy-out speed, the upward trend in $V_{\text{content}}$ continues up to a point where $\text{Speed}_{\text{upload}} = \frac{8}{\Delta T_{\text{data, copy}}}$.

**Figure 7: Maximum Possible Data Size for Faster Uploading**

![Graph showing the relationship between data size and ISP upload speed](image)
beyond which $V_{\text{content}}$ becomes infinite, meaning that it is no longer possible to ship the data more quickly than simply uploading it to the cloud, no matter how gargantuan the data size. In this case, unless you switch to a faster means of copying data in and out of the storage appliance, you are better off simply uploading it to the destination cloud.

Again, in the scenario of LTO-6 tape-based data transfer running at 160-MBps transfer speed, overnight shipping of 16 hours, and 48 hours of overhead time, the upload speed beyond which it’s always faster to upload than to ship your data is 640 Mbps. If you have access to a faster means of drive-to-drive data copying—say, running at 320 MBps—your ISP will need to offer a sustained upload speed of more than 1,280 Mbps to make it speedier for you to upload the data than to copy and ship it.

CLOUD-TO-CLOUD DATA TRANSFER

Another strategy is to transfer data directly from the source cloud to the destination cloud. This is usually done using APIs from the source and destination cloud providers. Data can be transferred at various levels of granularity such as logical objects, buckets, byte blobs, files, or simply a byte stream. You can also schedule large data transfers as batch jobs that can run unattended and alert you on completion or failure.

Consider cloud-to-cloud data transfer particularly when:

- Your data is already in one such cloud-storage provider and you wish to move it to another cloud-storage provider.
- Both the source and destination cloud-storage providers offer data egress and ingress APIs.
You wish to take advantage of the data copying and scheduling infrastructure and services already offered by the cloud providers.

Note that cloud-to-cloud transfer is conceptually the same as uploading data to the cloud in that the data moves over an Internet connection. Hence, the same speed considerations apply to it as explained previously while comparing it with the strategy of shipping data to the data center. Also note that the Internet connection speed from the source to the destination cloud may not be the same as the upload speed provided by the ISP.

**COST OF DATA TRANSFER**

LTO-6 tapes, at 0.013 cents per GB, provide one of the lowest cost-to-storage ratios, compared with other options such as HDD or SSD storage. It’s easy to see, however, that the total cost of tape cartridges becomes prohibitive for storing terabyte and beyond content sizes. One option is to store data in a compressed form. LTO-6, for example, can store up to 6.25 TB per tape in compressed format, thereby leading to fewer tape cartridges. Compressing the data at the source and uncompressing it at the destination, however, further reduces the effective copy-in/copy-out speed of LTO tapes, or for that matter of any other storage medium. As explained earlier, a low copy-in/copy-out speed can make shipping the data less attractive than uploading it over a fiber-based ISP link.

But what if the cloud-storage provider loaned the storage appliance to you? This way, the provider can potentially...
afford to use higher-end options such as high-end SSDs or a combination HDD-SSD array in the storage appliance, which would otherwise be prohibitively expensive to purchase just for the purpose of transferring data. In fact, that is exactly the approach that Amazon appears to have taken with its AWS [Amazon Web Services] Snowball.\(^3\) Amazon claims that up to 50 TB of data can be copied from your data source into the Snowball storage appliance in less than one day. This performance characteristic translates into a sustained data-transfer rate of at least 600 MBps. This kind of data-transfer rate is possible only with very high-end SSD/HDD drive arrays with read-ahead caches operating over a fast interface such as SATA Revision 3, SAS-3, or PCI Express, and a Gigabit Ethernet link out of the storage appliance.

In fact, the performance characteristics of AWS Snowball closely resemble those of a high-performance NAS (network-attached storage) device, complete with a CPU, on-board RAM, built-in data encryption services, Gigabit Ethernet network interface, and a built-in control program—not to mention a ruggedized, tamper-proof construction. The utility of services such as Snowball comes from the cloud provider making a very high-performance (and expensive) NAS-like device available to users for “rent” to copy-in/copy-out files to the provider’s cloud. Other major cloud providers such as Google and Microsoft aren’t far behind in offering such capabilities. Microsoft requires you to ship SATA II/III internal HDDs for importing or exporting data into/from the Azure cloud and provides the software needed to prepare the drives for import or export.\(^6\) Google, on the other hand,
appears to have outsourced the data-copy service to a third-party provider.\textsuperscript{8}

One final point on the cost: unless your data is in a self-managed data center, usually the source cloud provider will charge you for data egress,\textsuperscript{4,5,12,15} whether you do a disk-based copying out of data or cloud-to-cloud data transfer. These charges are usually levied on a per-GB, per-TB, or per-request basis. There is usually no data ingress charge levied by the destination cloud provider.

CONCLUSION

If you wish to move big data from one location to another over the Internet, there are a few options available—namely, uploading it directly using an ISP-provided network connection, copying it into a storage appliance and shipping the appliance to the new storage provider, and, finally, cloud-to-cloud data transfer.

Which technique you choose depends on a number of factors: the amount of data to be transferred, the sustained Internet connection speed between the source and destination servers, the sustained drive-to-drive copy-in/copy-out speeds supported by the storage appliance and the source and destination drives, the monetary cost of data transfer, and to a smaller extent, the shipment cost and transit time. Some of these factors result in the emergence of threshold upload speeds and threshold data sizes that fundamentally influence which strategy you should choose. Drive-to-drive copy-in/copy-out times have enormous influence on whether it is attractive to copy and ship data, as
opposed to uploading it over the Internet, especially when competing with an optical fiber-based Internet link.

References

11. IEEE. 802.3: Ethernet standards; http://standards.ieee.org/about/get/802/802.3.html.

**LOVE IT, HATE IT? LET US KNOW** feedback@queue.acm.org

Sachin Date ([https://in.linkedin.com/in/sachindate](https://in.linkedin.com/in/sachindate)) looks after the Microsoft and cloud applications portfolio of e-Emphasys Technologies ([www.e-emphasys.com](http://www.e-emphasys.com)). In his past lives, Date has worked as a practice head for mobile technologies, an enterprise software architect, and a researcher in autonomous software agents. He blogs at [https://sachinsdate.wordpress.com](https://sachinsdate.wordpress.com). He holds a master’s degree in computer science from the University of Massachusetts at Amherst.

Copyright © 2016 held by owner/author. Publication rights licensed to ACM.
We're more than computational theorists, database managers, UX mavens, coders and developers. We’re on a mission to solve tomorrow. ACM gives us the resources, the access and the tools to invent the future. Join ACM today and receive 25% off your first year of membership.

BE CREATIVE. STAY CONNECTED. KEEP INVENTING.

ACM.org/KeepInventing