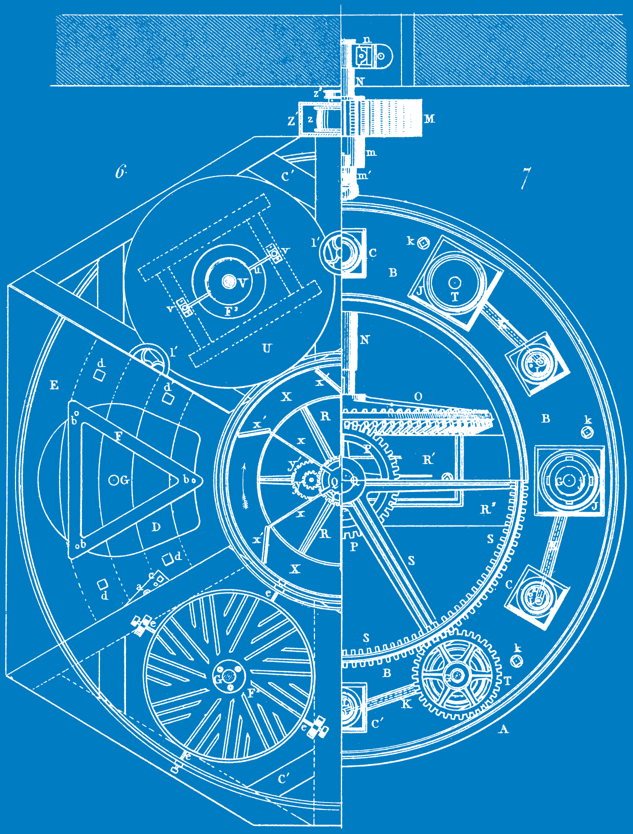
APRIL 2009 VOLUME 34 NUMBER 2

;login:

THE USENIX MAGAZINE



USENIX
The Advanced Computing Systems Association

2009 USENIX ANNUAL TECHNICAL CONFERENCE

June 14–19, 2009 • SAN DIEGO, CA

Join us in San Diego, June 14–19, 2009, for the 2009 USENIX Annual Technical Conference. USENIX Annual Tech has always been the place to present groundbreaking research and cutting-edge practices in a wide variety of technologies and environments. USENIX '09 will be no exception.

REGISTER WITH PRIORITY CODE ATCLOG09 AND SAVE $100!

USENIX '09 will feature:

- An extensive Training Program, covering crucial topics and led by highly respected instructors
- Technical Sessions, featuring the Refereed Papers Track, Invited Talks, and a Poster/Demo Session
- Workshop on Cloud Computing
- And more!

Join the community of programmers, developers, and systems professionals in sharing solutions and fresh ideas.

USENIX

Register by Monday, June 1, and save! **http://www.usenix.org/usenix09/loa**

contents

;login:

VOL. 34, #2, APRIL 2009

EDITOR
Rik Farrow
rik@usenix.org

MANAGING EDITOR
Jane-Ellen Long
jel@usenix.org

COPY EDITOR
David Couzens
proofshop@usenix.org

PRODUCTION
Casey Henderson
Jane-Ellen Long
Michele Nelson
Jennifer Peterson

TYPESETTER
Star Type
startype@comcast.net

USENIX ASSOCIATION
2560 Ninth Street,
Suite 215, Berkeley,
California 94710
Phone: (510) 528-8649
FAX: (510) 548-5738

http://www.usenix.org
http://www.sage.org

;login: is the official magazine of the USENIX Association.

;login: (ISSN 1044-6397) is published bi-monthly by the USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710.

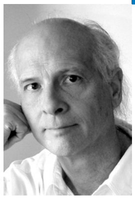
$90 of each member's annual dues is for an annual subscription to *;login:*. Subscriptions for nonmembers are $125 per year.

Periodicals postage paid at Berkeley, CA, and additional offices.

POSTMASTER: Send address changes to *;login:*, USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710.

RIK FARROW

musings



Rik is the Editor of *;login:*.

rik@usenix.org

WHERE WAS I? OH YEAH, IN SAN DIEGO again, but this time for OSDI. OSDI and SOSP are the two big operating system conferences in the US, and I've particularly enjoyed being able to attend OSDI. I sat in the second row during all sessions, listening intently. In this issue, I've worked with the authors for two OSDI papers to share some of what I learned, and I asked researchers at the University of Rochester to write a survey article about transactional memory (TM).

You may not have heard of TM (unless you are old enough to remember Transcendental Meditation), but TM may become very important in both hardware and programming languages in the near future. As Dave Patterson said during his keynote at USENIX Annual Tech in 2008, multicore is here. Multicore processors have not just arrived, they are the clear path forward in increasing processor performance, and TM looks like a good way to make parallel programming techniques accessible to people besides database and systems programmers.

Locks

Mutual exclusion (mutex) locks have been the technique of choice for protecting critical sections of code. You may recall past *;login:* articles about removing the "big lock" from Linux or FreeBSD kernels. The big lock refers to having one lock that insures that only one thread can be running within the locked code at a time. Having one big lock is inefficient, as it blocks parallel execution of key kernel code.

Over time, system programmers refined locking by replacing one big lock with fine-grained locks—locks that protect much smaller code segments. Programmers have to work carefully, because having multiple locks can result in code that deadlocks when one locked section requires access to a mutex for another already locked resource, which in turn requires access to the first locked section.

TM replaces locking with transactions, where the results of an operation are either committed atomically (all at once) or aborted. Within a processor's ABI there are precious few atomic operations, as these play havoc with instruction pipelines. These operations are useful for implementing mutexes, but not for handling transactions that will span the much larger blocks of code found in critical sections of locked code.

In Shriraman et al. you will learn of the various techniques in hardware, in software, and in mixed approaches to support TM. Hardware approaches are faster but inflexible and limited in scope. Software approaches are painfully slow, as they must emulate hardware features, and that also adds considerably to the amount of memory involved in a transaction.

As I was reading this article, I found myself wanting to reread Hennesey and Patterson [1] about caches and cache coherence. If you don't have access to this book, Wikipedia has a very decent entry on caches [2]. Many TM approaches rely on tags added to caches for their operation, and the tags themselves are related to cache coherency.

Recall that only registers can access data within a single processor clock cycle. Level 1 (L1) caches provide more memory than registers, but accessing the data requires multiple clock cycles. As the caches become larger (L2 and L3 caches), the number of clock cycles increases because of the hardware involved in determining whether a particular cache contains valid data for the desired memory address.

In single-threaded programs and programs that do not explicitly share memory, coherency issues do not arise. Only one thread has access to each memory location. But in multi-threaded programs and programs that share memory, the caches associated with a core, such as L1 cache, will contain data that needs to be consistent with the data found in the rest of the memory system. Cache coherency systems handle this by tagging each cache block with status, such as *exclusive*, *shared*, *modified*, and *invalid*. Processor-level hardware then manages coherency between the different caches by updating an invalided cache block when it is accessed, for example.

Shriraman et al.'s favored solution involves extending coherency mechanisms to support flexible TM. I like this article, as it is a thorough survey of the approaches to TM, as well as a clear statement of the issues, such as how TM can be an easier mechanism for programmers to use that avoids deadlock and approaches the performance of fine-grained locks.

Although parallel programming is largely the domain of systems, databases, and some gaming programmers, the wider use of multicore processors suggests that more programmers who require high performance will be looking to add parallelism to their code. TM appears to be a workable approach to writing efficient and easy-to-debug parallel code.

Memory and Syscalls

The next two articles don't go as deeply into the use of processor features. Gupta et al. consider the use of sharing portions of pages of memory as well as compressing memory. VMware ESX can share identical pages of memory, something that occurs as much as 40% of the time when homogeneous guests are running within VMs. By extending sharing to partial pages and by compressing rarely used pages, Difference Engine can save much more memory, allowing more VMs to run on the same system with an increase in throughput.

Xax, described in "Leveraging Legacy Code for Web Browsers," relies on the system call interface for isolating a process. The system call interface is the gateway linking kernel services such as file system access, allocating memory, and communications with the network. Only the operating system has access to these hardware-mediated services, so it is possible to isolate a process effectively by interposing on system calls. During the paper presentation, I found myself wondering about this, but further reflection and a few words with the paper's presenter, Jon Howell, helped me recall that the system call mechanism is inviolable because of hardware features—not just the trap instruction, but also memory management.

Dave Beazley gets into the nitty-gritty of Python 3. I had heard Guido van Rossum talk about this new version of Python back in the summer of 2007, and even then he was talking about how older Python programs will not run unmodified under the new version. Dave explains, with examples, some of the reasons for the departure from backward compatibility while also showing exactly what pitfalls await those who venture unprepared into the new version.

Rudi van Drunen begins a series of articles on hardware, starting with the basics of electricity. If you find yourself wondering just how much voltage will drop along a run of 12-gauge wire or what exactly is meant by three-phase power, you will want to read this article. A must-read for anyone designing or overseeing machine rooms or even just racks of systems.

To go back to the beginning of this issue, immediately following these Musings, Mark Burgess expresses his misgivings (putting it mildly) about the "new" rage, cloud computing. Perhaps I should be writing "a new buzzword," as the cloud really isn't all the new, nor particularly shiny white, either.

We also have the usual array of columns, and I am not going to attempt to introduce them this issue. There are many more book reviews than usual this time around, including reviews of several programming books.

Finally, we have the reports on OSDI and some co-located workshops.

When it is time to write this column, I often go back and read past columns to get myself into the "write" mood (pun intended). I noticed how often I have written about operating systems and security (or lack thereof), wondering what a secure yet usable operating system might look like. As you will have noticed, we still don't have secure systems, and that goal appears as elusive as ever. But we do have steps that may lead us to more flexible systems that will include some steps toward better security, such as the isolation mechanism seen in Xax, as well as clever hacks, such as Difference Engine and forward-thinking designs, such as FlexTM. I find that I like computer systems research as much as ever, and I am proud to be part of the community that does this work.

REFERENCES

[1] J.L. Hennesey and D.A. Patterson, *Computer Architecture*, Fourth Edition (San Francisco: Morgan Kaufman, 2006), Section 4.2.

[2] http://en.wikipedia.org/wiki/CPU_cache.

MARK BURGESS

the cloud minders

Mark Burgess is professor of network and system administration at Oslo University College, Norway. He is the author of Cfengine, co-author of the SAGE Short Topics Booklet *A System Engineer's Guide to Host Configuration and Maintenance Using Cfengine,* and author of many books and research papers on system administration.

Mark.Burgess@iu.hio.no

EVERYTHING OLD IS NEW AGAIN. SO goes the famous adage, and never more so than in computing.

Distributed computing has conjured many mirages on its broad horizons over the years: from distributed computing environments to fully distributed operating systems, service oriented architecture, the grid, and now, lately, Cloud Computing. Should we be impressed? As a researcher and technologist, I am not. As a consumer, there is more to be said, but first let's look at the technology.

Each time one of these new manifestations of "wishful innovation" comes up at a conference I attend, I wince a little and wonder whether it will be worth committing to memory before the next one takes over. With so few ideas in IT management and so much palpable desperation to come up with something innovative in both research and industry, even the research community seems to have become blindly complacent about these magical phrases and may even see them as a godsend to fund one more round of paper recycling. But perhaps I do protest too much: We are all stuck in the same mess, cheered on by broken funding politics and commercial exuberance; after all, this is no more than a sign that information technology has truly entered the marketplace. And there is something to be said for the hype that provokes us into thinking about the upsides and downsides of computing economics, especially given current events in money markets.

This column is a comment on what is currently being called Cloud Computing. Normally, I would not bat much of an eyelid to anything so plainly construed, but on this occasion the name Cloud Computing itself is only a distraction. Of greater importance is what is being offered: the idea that computing as a rentable service is preferable to owning your own—and this in itself warrants some remarks.

The name Cloud Computing seems appropriate for something so vaporous, in which people see the shapes they want to. What is it really? Last year it seemed as insubstantial as a wisp of cirrus, often mentioned in connection with Web 2.0. Is it a web? Is it a cloud? No, it's really a kind of utility computing, and the Web is just its presumed application of choice.

Sale of online computing resources is not new, of course. It has been going on for some time, from shared tenancy computing in the 1960s [1], to email accounts with Hotmail, and then Gmail, Yahoo!, etc. Then came the rise of social networking sites such as Facebook, online photography, and YouTube. The list goes on. At several junctures, there were briefly held notions of Application Service Providers (ASPs) changing the face of computing by running all of our software for us in centralized factories on the Internet, freeing ordinary companies from the burdens of coping with ever-changing technology. However, this only ever met with limited success. It still exists in a few forms, and indeed it has now brought on the idea of Cloud Computing, but it did not eradicate the stand-alone PC in favor of lighter, smaller "thin clients" as originally suggested, and this alone should be an omen to prevent us crying Hallelujah!

The Web-related hype has been snowballing into a silly idea: that not only will all computational resources be consolidated into mass production sweatshops of on-demand servers, but that the current flora of technologies would all be collapsed into a single kind of technological packaging—namely, that ever halting darling of ad hoc innovation, Web services. Supposedly, all of our applications might one day be provided by giant providers like Amazon, Google, and Sun, all willing to sell us storage or unlimited mileage accounts for such things as email, and all via the browser.

Where will our data be then? They will be all "out there," as Captain Kirk might have said expansively with a twinkle in his eye, in the unknown folds of the global Internet, in no known location—just in "cyberspace." Hmmm.

The idea is of course inevitable, just as everything else about the globalization of the economy has been inevitable. But what is disturbing is the lack of thought in presenting this as "The Big Step Forward." It makes me think of the alacrity with which people threw themselves into the economic bubbles of the past decade. I suggest that "Cloud Computing" is far from "The Future of Distributed Computing," powering us on to the next generation, but likely only a footnote to a broader view of global services that will find a moderate market share in the commercial future of IT systems alongside a variety of other models. Why? Because of risk.

What Is Cloud Computing?

You might imagine from the sheer size of its Wikipedia entry that Cloud Computing was really something quite innovative and special. It is of course no different from any other kind of computing; it offers no new functions and no special features, and it is not necessarily any cheaper to provide on an hourly basis than any previous model for computing service, despite what is often claimed. In fact, on balance, it might even result in more carbon emissions, given the kinds of customers who are likely to use it. It certainly does nothing to improve the security of users, who still need a PC and a Web browser to access it, with all of those attendant flaws. It exists chiefly for convenience to a certain segment of the market for computer services.

Price and ease are the main driving forces for the online services being offered—large amounts of "cheap" storage, "cheap" applications to replace existing commercial ones, etc. For startups and hobbyists who have neither the expertise nor the resources to run their own servers, the idea of rentable virtual computers is an amazing convenience. Suddenly resources are available without searching for rackspace, network providers, or hosting companies, and without any investment in infrastructure. Fantastic!

But this could be misleading. We'll look at the economics of this in a minute, but for now consider why someone would want to use a cloud service rather than running software on a PC as everyone has done in the past.

Let us imagine two computer users, Alice and Bob (as they are often cryptographically named), who want to send email to one another, or perhaps simply use an application such as a spreadsheet. Alice, having practiced in front of her looking glass, is an expert computer user and owns her own Custom Classic Computer, complete with air-brushed bodywork and a V8 processor and custom grown software built from spare parts she finds on the Net. Every day she tunes the engine a little to maximize performance or even just for enjoyment, and there are few problems that a little amateur tweaking cannot fix.

Bob, by way of contrast, is just floating along, going with the flow. His heart is not really in the mechanics of computing, but he enjoys a comfortable ride through the spreadsheets and word processors from time to time when he needs them—his needs vary so he doesn't want to tie himself to just one thing. His attitude is that he "just wants it to work" and he is willing to pay a rental company to fix this computing for him because it is cheaper than owning his own. That way he knows he will always have the latest and it will be pumped up with the latest hot air and checked by experts. So he normally rents a cheap service from MegaHertz or CloudAvis and they even throw in a built-in MP3 system and air conditioning for the servers (at the data center, naturally) for their frequent flyers.

Bob can never get the kind of souped-up custom experience that Alice enjoys from her personal computer system, but he wants neither the hassle of her infatuation nor the responsibility of owning a depreciating pile of capital expenditure. Alice, for her part, would be mortally offended by the mere suggestion that she should plump for anything as degradingly generic as a cloud service. She saves a bundle by doing it herself; after all, how hard could it be to add a hard disk, remember to do a backup, or install a new program—things that would make Bob shudder?

You get the idea. Cloud computing is much like the idea of car rental, ski rental, or any other kind of pay-as-you-go service. A kind of online Internet cafe for application services. You will pay more than you strictly need to get something quite generic, possibly with selectable levels of service quality (basic, super, or ultra) or one of a limited number of special-needs solutions (hatchback, van, or snowboard). It would be cheaper on an hour-by-hour basis to have your own, but you haven't; moreover, you don't have to pay for the thing, store it, and maintain it when you don't need it, so in the end it could be a lot cheaper for the occasional fair-weather user. When we compare this to renting a car, it is not so mysterious.

So cloud computing is about providing computing service (a lot of the hardware and all of the software) as a commodity without the need for a large and risky personal investment. The cloud provider will take that risk and investment, which of course is a lower risk if you know you will have sufficient customers—and in the case of the chief providers today, you actually know they can use the machines for their primary business if no one is buying the cloud service, so they are not losing anything. Some authors have likened this to making information technology run like water from a tap or flow like electricity from a wall socket, but, as we shall see, this analogy is not the right one. In fact it is more like a bank account with a credit card, with all of the risks that entails.

Why Is Cloud Computing Not Like Electricity?

The argument for cloud computing is an economic one. The argument is not that it is cheaper for everyone, only that it is a service that some will find useful and that can be cheaply provided by some giants who do it as a kind of sideline, using their spare capacity to subsidize the sale. Cloud computing is not going to replace other forms of computing any more than car rentals have taken over the transport sector, because the model does not fit everyone's needs, but it could be quite useful to occasional computer users. Certainly the idea that companies might want to set up their own "local cloud" to make effective use of resource virtualization seems faintly ridiculous—you mean set up their own computer infrastructure, the way they've been doing for . . . how many years?

The real issues lurking for inexperienced consumers are the risks.

True, the cloud companies bear the risk of initial investment and they carry the cost of maintenance. But what shall we make of the subsidies they provide? If this is a sideline propped up by the cloud providers' core business model, then we should look rather carefully at whether that model is rock-solid and is likely to survive. Worse, cloud services are not like electricity or car rentals, because those services are "disposable" transactions. You consume these services once and then they are gone; nothing is stored or saved for the future. If they go under one day, you might be inconvenienced but you will not lose any savings. The economics are also easy to understand. There is a big pool of resources that can be shared by a lot of customers. With many customers a single provider can own an efficient fleet of cars or a flexible farm of servers and pay for them with a profit because there is always a sufficiency of customers coming back for more.

A bank is a more comparable service. Banks aspire to make money flow like electricity when needed, but with an important difference: Clients own their savings. A bank provides various services (perhaps for a fee, though these are mostly gone in Scandinavia) and they do it for the privilege of having your money for their use while you don't want it back. As long as there are many customers with enough money, the pool allows the bank to smooth over the inequities of individuals' financial details. The bank even pays you a nominal interest rate to cover the depreciation of the money due to inflation. There is no reason why people could not stuff money into mattresses or have their own private vault for storing money, but banks are successful because they provide certain conveniences. The key difference between banks and car rental is that banks provide safekeeping for something that matters to you: your money. But this is a risk we are usually willing to take. After all, banks don't go under, now do they? If you are in the black, the risk is yours. If you are in the red, the risk is theirs. For them, this evens out, but for you it doesn't, as thousands of pensioners and savers around the world learned over the past decade.

The key to pooling and sharing resources is that the fee for occasional use does not necessarily have to directly cover the actual cost of making the service available, as long as there are enough individuals to balance the incoming payments somehow. Alternatively, the whole thing can be subsidized by external funding. This makes the rental look cheap and stable to casual individuals. However, if the supply of money ever starts to get too low to smooth over the inequities, the transactions will grind to a halt, confidence in the model can be lost, and people will take their money elsewhere, causing a collapse. The model goes into a "recession" and the ones who remain could lose everything.

Cloud computing is much like a bank, because it will contain people's personal data and valuables. But for how long? If one of these services should suddenly stop working, that data would all be gone forever and you would never be able to rescue it, because it would still be "out there," lost in space.

The risk lies in the stability of the collective. If it does not attract enough individuals to maintain its services, or if it grows too many to service and maintain, or if there is not enough money to smooth over the imbalances in the pricing, then confidence in the system can be lost and it can collapse, meaning that all of its users will potentially lose everything stored there. Now, this doesn't matter for electricity or car rentals; you use them and then they are gone anyway. But with a bank you do care. And there is no central bank to bail out cloud computing.

The Stability of the Commons

No one seriously looks at Amazon or Google and thinks that these companies, the very knights of modern marketing savviness, will go under—but take care. Cloud companies do not have cloud computing as their primary business, at least for the moment, so they can effectively subsidize these collectives, making them seem artificially cheap. What happens when too many providers enter the market and prices rise? This could also cause a mass exodus from the providers.

Unlikely? No one expected the present banking crisis to emerge, either. All the funds were guaranteed by someone, weren't they? Unfortunately, when you are playing with margins there is not always an outside source that can come to the rescue in a sudden shortage market fluctuation. The problem with all stochastic systems (systems with fluctuations) is that there is pretty much always a freak wave out there that can wreak mass destruction in the system, one against which it simply doesn't pay to try to protect oneself, because it is so unlikely.

There is actually precedent for this kind of precipitated collapse of a collective commons in the Internet world already. The Internet has its own exchanges for trading spare capacity and pooling its resources (i.e., the Internet Exchanges). There major and minor network providers can trade their capacity either for money or, more often, for "Brownie points" or goodwill. Indeed, studies of these exchanges show that exchange agreements are based more on visibility than on material profit [2]. This means that the larger providers often do more than their fair share of giving away their spare capacity, and on occasion this has led to a major provider withdrawing from the exchange, leading to a crisis for the others, forcing them to pay real money for those Brownie points. This happened in Norway only a few years ago, placing confidence in the system in jeopardy [3].

But even if the possibility of collapse seems small, there are several causes for concern in cloud computing. One is security and privacy (who can see my data, and how do I verify the claim?); another is the question of geography. How about backup? If you need to have a backup for your data locally, then you either need some local infrastructure or you have to diversify your data investment over multiple providers that are not likely to go away all at the same time. What if there is a take-over? Will one copy go away? Will you know the physical location of the data and avoid the next big earthquake or flood?

What are the terms and conditions for the services? Does the (remote) provider retain the right to mine your data for marketing buzzwords? Will it be forced to reveal your private data to someone else under duress? Will it adequately destroy sensitive data when you ask it to, including all of the backup copies? Will its backups be properly secured? What about geography? Where precisely is your data stored? Is the data illegal in the country of storage? Will you always be able to access it? Is there political (or tectonic) stability in the location of the data? The potential problem is that there is practically no way to assess these risks. It's all just "out there."

For these reasons, cloud computing is not going to be for everyone. The Alices of the world are never going to find Wonderland in outsourcing. They live off the ability to customize nonstandard systems, and they have a heavy weapon against it: competence. Competence and technology actually make it cheap for individuals to manage their own concerns. There is no single recipe for solving the problem of scale, as we have discovered in our research into systems in Oslo. Centralization is but one approach to resource management [4].

Self-Reliance: The Counterpoint of Cloud Computing

There is another weapon in the computing arsenal that could play a role. It has come increasingly to the fore of late. It began in the 1990s with artificial immune systems or computer immunology, and today it is often called "self-healing" technology. By contrast with cloud computing, which is mainly a brute-force cost reduction, self-healing is a set of more subtle technologies. The idea is (as with smart modern cars) to get experts to program the requirements and safe working conditions for computer systems in advance and then equip the units with smart technology that allows them to maintain this condition for the greatest possible time, ultimately eliminating the need for human intervention until an unexpected decision has to be made. Automation is a technology that can level the playing field again, removing some of the benefits of cloud convenience.

Whereas brute-force cost-cutting would try to make everything absolutely identical in order to keep down costs, the self-healing configuration approach actually tries to improve the technology itself to make a more effective system manage itself cheaply. Futurist Alvin Toffler wrote about this phenomenon in manufacturing at the end of the 1960s and concluded, "As technology becomes more sophisticated, the cost of introducing variations declines."

The differentiating self-healing technologies such as Cfengine, and to some extent IBM's autonomic initiative and HP's work in the area, are taking a different path to the idea of the cloud (and the Cloud Minders too can benefit from it), namely, bringing computer expert systems back to support complexity cheaply rather than offering only vanilla and strawberry flavored services to potential buyers (i.e., any color as long as it's black).

Self-healing, then, could be the thorn in the side of naive cloud computing, making resource flexibility easy at home. Do Hertz and Avis outsource cars to specialist companies, or do companies buy their own car pools? Of course both models exist, just as "cloud computing" is likely to coexist with in-house expertise, enabled with powerful self-healing systems in the years to come. Consolidation did not capture the market and change the world before, so why should it now? Pretty much every development in personal technology, starting with the motor car, has been about the opposite of pooling resources: mobile phones, microwave ovens, PDAs, Blackberries, iPods are all about personal enablement, making oneself independent of ties.

Consolidation is a strategy for the non-resource-wealthy that pokes its head up and dives down again like the Loch Ness Monster, at reasonably regular intervals in computing. When a resource becomes scarce, it encourages pooling of those resources through consolidation. Sometimes it was the need for processor capacity, sometimes it was memory, sometimes fast communication. The scarce resource today is *competence*, specifically in the areas of management and maintenance, but self-healing will take away much of the need for this too.

The technological phenomenon is the growth of computer virtualization for effective resource management. This is a healthy reality check, as power requirements force us to reconsider vulgar excesses. The role of the Internet will only come into play if resources can be moved dynamically around the globe to optimize time zones and traffic burst in a dynamic and secure fashion. That would be a true technology to propel us into the stratosphere.

Unlike some, I am not bowled over by cloud computing, any more than I was impressed by grid services or any other special packaging for distributed computing. Yes, of course there are arguments for it. It has its place where expertise is lacking or temporary, throw-away resources are required on short notice, but this is not a fundamental shift, only a commercial opportunity, and it does not free users from their responsibilities for thinking about backup and security. More convincing are the benefits of renting software as a service: paying a smaller regular fee for continuous updates, in which we keep our own data privately and safely. This is a model for nearly all regular computer users.

Marketing is a powerful force that is sometimes genuinely creative. I only wish that as much effort could be expended in educating competent specialists to solve the technical challenges of resource management as is put into the manufacturing of media hype to merely suggest overcoming them. Brute-force mediocrity is almost a standard for computing today, even in research. Personally, I am holding out for the next level: self-healing computers, with self-scaling automation, that can be deployed anywhere, not just in vast datacenters. This has nothing to do with the Web or the Internet but has everything to do with intelligent configuration.

As Droxine, the lovely daughter of the Cloud Minder, said in the memorable Star Trek story, "I shall go to the mines; I no longer wish to be limited to the clouds" [5].

ACKNOWLEDGMENTS

I would like to acknowledge helpful discussions and input from Khanan Grauer, Rik Farrow, and Helge Skrivervik. None of them necessarily shares any of the opinions expressed here.

REFERENCES

[1] For example, http://en.wikipedia.org/wiki/Automatic_Data_Processing.

[2] W.B. Norton, "The Art of Peering: The Peering Playbook," Equinix.com, 2001.

[3] For example, http://www.nettavisen.no/it/article1245673.ece and http://www.dagbladet.no/kultur/2007/06/20/504054.html. (Search Google for "Oslo telenor trekker seg internet exchange" for references in Norwegian.)

[4] For example, see research at http://research.iu.hio.no/promises.php.

[5] Paramount Pictures, *Star Trek* (Original Series 3), "The Cloud Minders."

ARRVINDH SHRIRAMAN,
SANDHYA DWARKADAS, AND
MICHAEL L. SCOTT

tapping into parallelism with transactional memory

Arrvindh Shriraman is a graduate student in computer science at the University of Rochester. Arrvindh received his B.E. from the University of Madras, India, and his M.S. from the University of Rochester. His research interests include multiprocessor system design, hardware-software interface, and parallel programming models.

ashriram@cs.rochester.edu

Sandhya Dwarkadas is a professor of computer science and of electrical and computer engineering at the University of Rochester. Her research lies at the interface of hardware and software with a particular focus on concurrency, resulting in numerous publications that cross areas within systems. She is currently an associate editor for IEEE Computer Architecture Letters (and has been an associate editor for IEEE *Transactions on Parallel and Distributed Systems*).

sandhya@cs.rochester.edu

Michael Scott is a professor and past Chair of the Computer Science Department at the University of Rochester. He is an ACM Fellow, a recipient of the Dijkstra Prize in Distributed Computing, and author of the textbook *Programming Language Pragmatics* (3d edition, Morgan Kaufmann, 2009). He was recently Program Chair of TRANSACT '07 and of PPoPP '08.

scott@cs.rochester.edu

MULTICORE SYSTEMS PROMISE TO deliver increasing performance only if programmers make thread-level parallelism visible in software. Unfortunately, multithreaded programs are difficult to write, largely because of the complexity of synchronization. Transactional memory (TM) aims to hide this complexity by raising the level of abstraction. Several software, hardware, and hybrid implementations of TM have been proposed and evaluated, and hardware support has begun to appear in commercial processors. In this article we provide an overview of TM from a systems perspective, with a focus on implementations that leverage hardware support. We describe the principal hardware alternatives, discuss performance and implementation tradeoffs, and argue that a classic "policy-in-software, mechanism-in-hardware" strategy can combine excellent performance with the flexibility to accommodate different system goals and workload characteristics.

For more than 40 years, Moore's Law has packed twice as many transistors on a chip every 18 months. Between 1974 and 2004, hardware vendors used those extra transistors to equip their processors with ever-deeper pipelines, multi-way issue, aggressive branch prediction, and out-of-order execution, all of which served to harvest more instruction-level parallelism (ILP). Because the transistors were smaller, vendors were also able to dramatically increase the clock rate. All of that ended about four years ago, when microarchitects ran out of independent things to do while waiting for data from memory, and when the heat generated by faster clocks reached the limits of fan-based cooling. Future performance improvements must now come from multicore processors, which depend on explicit, thread-level parallelism. Four-core chips are common today, and if programmers can figure out how to use them, vendors will deliver hundreds of cores within a decade. The implications for software are profound: Historically only the most talented programmers have been able to write good parallel code; now everyone must do it.

Sadly, parallel programming is *hard*. Historically it has been limited mainly to servers, with "embarrassingly parallel" workloads, and to high-end scientific applications, with enormous data sets and enormous budgets. Even given a good division of labor among threads (something that's often difficult to find), mainstream applications are plagued by the need to synchronize access to shared state. For this, programmers have traditionally relied on mutual exclusion locks, but these suffer from a host of problems, including the lack of composability (one can't nest two lock-based operations inside a new critical section without introducing the possibility of deadlock) and the tension between concurrency and clarity: Coarse-grain lock-based algorithms are relatively easy to understand (grab the One Big Lock, do what needs doing, and release it) but they preclude any significant parallel speedup; fine-grained lock-based algorithms allow independent operations to proceed in parallel, but they are notoriously difficult to design, debug, maintain, and understand.

Transactional Memory (TM) aims to simplify synchronization by raising the level of abstraction. As in the database world, the programmer or compiler simply marks a block of code as "atomic"; the underlying system then promises to execute the block in an "all-or-nothing" manner isolated from similar blocks (transactions) in other threads. The implementation is typically based on *speculation*: It guesses that transactions will be independent and executes them in parallel, but watches their memory accesses just in case. If a conflict arises (two concurrent transactions access the same location, and at least one of them tries to write it), the implementation *aborts* one of the contenders, rolls back its execution, and restarts it at a later time. In some cases it may suffice to *delay* one of the contending transactions, but this does not work if, for example, each transaction tries to write something that the other has already read.

TM can be implemented in hardware, in software, or in some combination of the two. Software-only implementations have the advantage of running on legacy machines, but it is widely acknowledged that performance competitive with fine-grain locks will require hardware support. This article aims to describe what the hardware might look like and what its impacts might be on system software. We begin with a bit more detail on the TM programming model and a quick introduction to software TM. We then describe several ways in which brief, small-footprint transactions can be implemented entirely in hardware. Extension to transactions that overflow hardware tables or must survive a context switch are considered next. Finally, we describe our approach to hardware-accelerated software-controlled transactions, in which we carefully separate policy (in software) from mechanism (in hardware).

Transactional Memory in a Nutshell

Although TM systems vary in how they handle various subtle semantic issues, all are based on the notion of *serializability*: Regardless of implementation, transactions *appear* to execute in some global serial order. Writes by transaction A must never become visible to other transactions until A commits, at which time *all* of its writes must be visible. Moreover, writes by other transactions must never become visible to A partway through its own execution, even if A is doomed to abort (for otherwise A might perform some logically impossible operation with externally visible effects). Some TM systems relax the latter requirement by sandboxing A so that any erroneous operations it may perform do no harm to the rest of the program.

The principal motivation for TM is to simplify the parallel programming model. In some cases (e.g., if transactions are used in lieu of coarse-grain locks), it may also lead to performance improvements. An example appears in Fig. 1: If X ≠ Y, it is likely that the critical sections of Threads 1 and 2 can execute safely in parallel. Because locks are a low-level mechanism, they preclude such execution. TM, however, allows it. If we replace the lock...unlock pairs with atomic{...} blocks, the typical TM implementation will execute the two transactions concurrently, aborting and retrying one of the transactions only if they actually conflict.

```
Thread 1                               Thread 2
  lock(hash_tab.mutex)                   lock(hash_tab.mutex)
    var = hash_tab.lookup(X);              var = hash_tab.lookup(Y);
    if(!var)                               if(!var)
      hash_tab.insert(X);                    hash_tab.insert(Y);
  unlock(hash_tab.mutex)                 unlock(hash_tab.mutex)
```

FIGURE 1: LOSS OF PARALLELISM AS A RESULT OF LOCKS [13]

IMPLEMENTATION

Any TM implementation based on speculation must perform at least three tasks: It must (1) detect and resolve conflicts between transactions executing in parallel; (2) keep track of both old and new versions of data that are modified speculatively; and (3) ensure that running transactions never perform erroneous, externally visible actions as a result of an inconsistent view of memory.

Conflict resolution may be *eager* or *lazy*. An eager system detects and resolves conflicts as soon as a pair of transactions have performed (or are about to perform) operations that preclude committing them both. A lazy system delays conflict resolution (and possibly detection as well) until one of the transactions is ready to commit. The losing transaction L may abort immediately or, if it is only about to perform its conflicting operation (and hasn't done so yet), it can wait for the winning transaction W to either abort (in which case L can proceed) or commit (in which case L may be able to occur after W in logical order).

Lazy conflict resolution exposes more concurrency by permitting both transactions in a pair of concurrent R-W conflicting transactions to commit so long as the reader commits (serializes) before the writer. Lazy conflict resolution also helps in ensuring that the conflict winner is likely to commit: If we defer to a transaction that is ready to commit, it will generally do so, and the system will make forward progress. Eager conflict resolution avoids investing effort in a transaction L that is doomed to abort, but it may waste the work performed so far if it aborts L in favor of W and W subsequently fails to commit owing to conflict with some third transaction T. Recent work [17, 22] suggests that eager management is inherently more performance-brittle and livelock-prone than lazy management. The performance of eager systems can be highly dependent on the choice of *contention management* (arbitration) policy used to pick winners and losers, and the right choice can be application-dependent.

Version management typically employs either *direct update*, in which speculative values are written to shared data immediately and are *undone* on abort, or *deferred update*, in which speculative values are written to a log and *redone* (written to shared data) on commit. Direct update may be somewhat cheaper if—as we hope—transactions commit more often than they abort.

Systems that perform lazy conflict resolution, however, must generally use deferred update, to enable parallel execution of (i.e., speculation by) conflicting writers.

A BRIEF LOOK AT SOFTWARE TM

To track conflicts in the absence of special hardware, a software TM (STM) system must augment a program with instructions that read and write some sort of metadata. If program data are read more often than written (as is often the case), it is generally undesirable for readers to modify metadata, since that tends to introduce performance-sapping cache misses. As a result, readers are invisible to writers in most STM systems and bear full responsibility for detecting conflicts with writers. This task is commonly rolled into the problem of *validation*—ensuring that the data read so far are mutually consistent.

State-of-the-art STM systems perform validation on every nonredundant read. The supporting metadata varies greatly: In some systems, a reader inspects a modification timestamp or writer (owner) id associated with the location it is reading. In other systems, the reader inspects a list of Bloom filters that capture the write sets of recently committed transactions [21]. In the former case, metadata may be located in object headers or in a hash table indexed by virtual address.

Figure 2 shows the overhead of an STM system patterned after TL2 [5], running the STAMP benchmark suite [12]. This overhead is embedded in every thread, cannot be amortized with parallelism, and in fact tends to increase with processor count, owing to contention for metadata access. Here, versioning adds 2%–150% to program run time, while conflict detection and validation add 10%–290%. Static analysis may, in some cases, be able to eliminate significant amounts of redundant or unnecessary validation, logging, and memory fence overhead. Still, it seems reasonable to expect slowdowns on the order of factors of 2–3 in STM-based code, relative to well-tuned locks, reducing the potential for their adoption in practice.

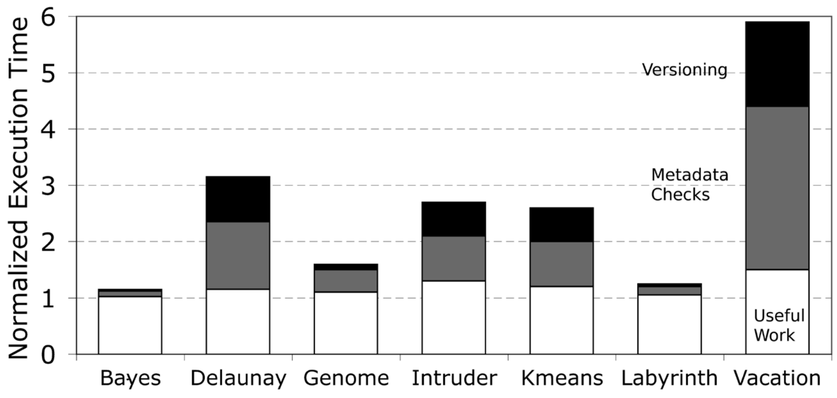


FIGURE 2: EXECUTION TIME BREAKDOWN FOR SINGLE-THREAD RUNS OF A TL2-LIKE STM SYSTEM ON APPLICATIONS FROM STAMP, UNINSTRUMENTED CODE RUN TIME = 1

Hardware for Small Transactions

On modern processors, locks and other synchronization mechanisms tend to be implemented using compare-and-swap (CAS) or load-linked/store-conditional (LL/SC) instructions. Both of these options provide the ability to read a single memory word, compute a new value, and update the word, atomically. Transactional memory was originally conceived as a way to extend this capability to multiple locations.

HERLIHY AND MOSS

The term “transactional memory” was coined by Herlihy and Moss in 1993 [9]. In their proposal (“H&M TM”), a small “transactional cache” holds speculatively accessed locations, including both old and new values of locations that have been written. Conflicts between transactions appear as an attempt to invalidate a speculatively accessed line within the normal coherence protocol and cause the requesting transaction to abort. A transaction commits if it reaches the end of its execution while still in possession of all speculatively accessed locations. A transaction will always abort if it accesses more locations than will fit in the special cache, or if its thread loses the processor as a result of preemption or other interrupts.

OKLAHOMA UPDATE

In modern terminology, H&M TM called for eager conflict resolution. A contemporaneous proposal by Stone et al. [23] envisioned lazy resolution, with a conflict detection and resolution protocol based on two-phase commit. Dubbed the “Oklahoma Update” (after the Rogers and Hammerstein song “All er Nuthin’ ”), the proposal included a novel solution to the doomed transaction problem: As part of the commit protocol, an Oklahoma Update system would immediately restart any aborted competing transactions by branching back to a previously saved address. By contrast, H&M TM required that a transaction explicitly poll its status (to see if it was doomed) prior to performing any operation that might not be safe in the wake of inconsistent reads.

AMD ASF

Recently, researchers at AMD have proposed a multiword atomic update mechanism as an extension to the x86-64 instruction set [6]. Their Advanced Synchronization Facility (ASF), although not a part of any current processor roadmap, has been specified in considerable detail. As H&M TM does, it uses eager conflict resolution, but with a different contention management strategy: Whereas H&M TM resolves conflicts in favor of the transaction that accessed the conflicting location first, ASF resolves it in favor of the one that accessed it last. This “requester wins” strategy fits more easily into standard invalidation-based cache coherence protocols, but it may be somewhat more prone to livelock. As Oklahoma Update does, ASF includes a provision for immediate abort.

SUN ROCK

Sun’s next-generation UltraSPARC processor, expected to ship in 2009 [7], includes a thread-level speculation (TLS) mechanism that can be used to implement transactional memory. As do H&M TM and ASF, Rock [24] uses eager conflict management; as does ASF, it resolves conflicts in favor of the requester. As do Oklahoma Update and ASF, it provides immediate abort. In a significant advance over these systems, however, it implements true processor checkpointing: On abort, all processor registers revert to the values they held when the transaction began. Moreover, all memory accesses within the transaction (not just those identified by special load and store instructions) are considered speculative.

STANFORD TCC

Although still limited (in its original form) to small transactions, the Transactional Coherence and Consistency (TCC) proposal of Hammond et al. [8] represented a major break with traditional concepts of memory access and communication. Whereas traditional threads (and processors) interact via individual loads and stores, TCC expresses all interaction in terms of transactions.

Like the multi-location commits of Oklahoma Update, TCC transactions are lazy. Individual writes within the transaction are delayed (buffered) and propagated to the rest of the system in bulk at commit time. Commit-time conflict detection and resolution employ either a central hardware arbiter or a distributed two-phase protocol. As in Rock, doomed transactions suffer an immediate abort and roll back to a processor checkpoint.

DISCUSSION

A common feature of the systems described in this section is the careful leveraging of existing hardware mechanisms. Eager systems (H&M TM, ASF, and Rock) leverage existing coherence protocol actions to detect transaction conflicts. In all five systems, hardware avoids most of the overhead of both conflict detection and versioning. At the same time, transactions in all five can abort simply because they access too much data (overflowing hardware resources) or take too long to execute (suffering a context switch). Also, although the systems differ in both the eagerness of conflict detection and resolution and the choice of winning transaction, in all cases these policy choices are embedded in the hardware; they cannot be changed in response to programmer preference or workload characteristics.

Unbounded Transactions

Small transactions are not sufficient if TM becomes a generic programming construct that can interact with other system modules (e.g., file systems and middleware) that have much more state than the typical critical section. It also seems unreasonable to expect programmers to choose transaction boundaries based on hardware resources. What is needed are low-overhead "unbounded" transactions that hide hardware resource limits and persist across system events (e.g., context switches, system calls, and device interrupts).

To support unbounded transactions, a TM system must virtualize both conflict detection and versioning. In both cases, the obvious strategy is to mimic STM and move transactional state from hardware to a metadata structure in virtual memory. Concrete realizations of this strategy vary in hardware complexity, degree of software intervention, and flexibility of conflict detection and contention management policy. In this section, we focus on implementation tradeoffs, dividing our attention between hardware-centric and hybrid hardware-software schemes. Later, we will turn to hardware-accelerated schemes that are fundamentally controlled by software, thereby affording policy freedom.

HARDWARE-CENTRIC SYSTEMS

Several systems have extended simple hardware TM (HTM) systems with hardware controllers that iterate through data structures housed in virtual memory. For example, the first unbounded HTM proposal, UTM [1], called for both an in-memory log of transactional writes and an in-memory descriptor for every fixed-size block of program data (to hold read-write permission bits). The descriptors (metadata) constituted an unbounded extension of the access tracking structures found in bounded (small-transaction) HTM. The log constituted an unbounded extension of bounded HTM versioning. Although located in virtual memory, both structures were to be maintained by a hardware controller active on every transactional read and write.

Subsequent unbounded HTM proposals have typically employed a two-level strategy in which a hardware controller implements small transactions in the same way as bounded HTM, but invokes firmware (or low-level software) handlers when space or time resources are exhausted. VTM [14], for example, uses deferred update and buffers speculative writes in the L1 cache as long as they fit. If a speculative line must be evicted owing to limited capacity or associativity, firmware (microcode) moves the line and its metadata to a data structure in virtual memory and maintains both a count of such lines and summary metadata (counting Bloom filters) for all evicted lines. On a context switch, a handler iterates through the entire cache and moves all speculative lines to this data structure. Subsequent accesses (when the count is nonzero) trigger firmware handlers that perform lookup operations of the in-memory data structures and summary metadata in order to detect conflicts (or fetch prior updates within the same transaction). Unfortunately, the cost of lookups is nontrivial.

Bloom-filter–based access-set tracking has also been used in direct-update systems. In LogTM-SE [25], a hardware controller buffers old values in an undo log residing in virtual memory, while speculative values update the original locations (which requires eager conflict resolution in order to avoid atomicity violations). Bloom filters are easy to implement in hardware and can be small enough to virtualize (save and restore) easily. Their drawback is imprecision. Although erroneous indications of conflict are not a correctness issue (since in the worst case, transactions can still execute one at a time), they may lead to lower performance [3].

Hardware-centric systems such as VTM and LogTM-SE hide most of the complexity of virtualization from the system programmer, resulting in a relatively simple run-time system. This simplicity, however, gives rise to semantic rigidity. Special instructions are needed, for example, to “leak” information from aborted transactions (e.g., for performance analysis). Similarly, policies that have first-order effects on performance (e.g., conflict resolution time, contention management policy) are fixed at system design time.

HYBRID APPROACHES

Hardware-centric approaches to unbounded TM demand significant investment from hardware vendors. Hybrid TM systems [4, 10] reduce this investment by adopting a two-level strategy in which the second level is in software. They begin with a “best-effort” implementation of bounded HTM; that is, they attempt to execute transactions in hardware, but the attempt can simply fail owing to implementation limitations. Software is then expected to pick up the pieces and ensure that all transactions are supported. The key idea is to generate two code sequences for transactions: an STM-compatible version that can run on stock processors and a second version that invokes the best-effort HTM. To ensure high performance, the STM is deployed only when HTM fails. The challenge is to ensure that HTM and STM transactions interoperate correctly. This is achieved by instrumenting the HTM transactions so that every memory operation also checks for concurrent, conflicting STM transactions. If one exists, then the HTM transaction fails, since it lacks the ability to perform conflict resolution with respect to the STM transaction.

Although hybrid systems keep the hardware simple, the instrumentation for interoperability may add significant overhead to HTM transactions. More ambitious hybrid systems [2] may improve performance by implementing conflict detection entirely in hardware (using extra bits associated with main memory), while performing versioning in software. As did hardware-centric unbounded TM, hybrid TM suffers from policy inflexibility inherited from the all-hardware case, and from significant overhead whenever overflow occurs.

Hardware-Accelerated Software-Controlled Transactions

Experimental evidence suggests that although eager conflict management may avoid wasted work, lazy systems may exploit more parallelism, avoid performance pathologies, and eliminate the need for sophisticated (and potentially costly) contention management [11, 17, 22]. Intermediate strategies (e.g., *mixed* conflict management, which resolves write-write conflicts eagerly and read-write conflicts lazily) may also be desirable for certain applications. Unfortunately, the hardware-centric and hybrid TM systems that we have discussed so far embed the choice of both conflict resolution time and contention management policy in silicon.

Hardware-accelerated but software-controlled TM systems [15, 16, 20] strive to leave such policy decisions under software control, while using hardware mechanisms to accelerate both bounded and unbounded transactions. This strategy allows the choice of policy to be tuned to the current workload. It also allows the TM system to reflect system-level concerns such as thread priority. As in the designs covered earlier, existing hardware mechanisms must be carefully leveraged to avoid potential impact on common-case non-transactional code.

The key insight that enables policy flexibility is that information gathering and decision making can be decoupled. In particular, data versioning, access tracking, and conflict detection can be supported as decoupled/separable mechanisms that do not embed policy. Conflict resolution time and contention management policy can then be decided dynamically by the application or TM runtime system.

DECOUPLED VERSIONING

To support lazy conflict resolution, we proposed a deferred-update versioning mechanism we call Programmable Data Isolation (PDI) [15]. PDI allows selective use of processor-private caches as a buffer for speculative writes or for reading/caching the current version of locations being speculatively written remotely. PDI lines are tracked by augmenting the coherence protocol with a pair of additional states. Data associated with speculative writes is not propagated to the rest of the system, allowing multiple transactions to speculatively read or write the same location. However, coherence actions *are* propagated, allowing remote caches to track the information necessary to return them to a coherent state, without resolving the detected conflict immediately.

To support cache overflow of speculative state, a hardware-based overflow table (akin to a software-managed translation lookaside buffer) is added to the miss path of the L1 cache. Replacement of a speculatively modified cache line results in it being written back to a different (software-specified) region of the process's virtual memory space. A miss in the overflow table results in a trap to software, which can then set up the necessary mapping. In other words, software controls where and how the speculative modifications are maintained while hardware performs the common case (in the critical path) operation of copying data into and out of the cache.

DECOUPLED CONFLICT DETECTION AND RESOLUTION

Access tracking can be performed in hardware by adding extra bits in the private cache to indicate a speculatively modified copy. However, this tracking is bounded by the size of the cache. Alternative forms of tracking for an unbounded amount of metadata include Bloom-filter signatures [3] and ECC bits in memory [2]. Our hardware [16] provides one set of Bloom filters on each processor to represent the read and write sets of the running thread and another to summarize the speculative read and write sets of all currently preempted threads. These signatures and, in some cases, the PDI state bits are checked on coherence protocol transitions in order to detect conflicts among concurrently executing transactions.

To decouple conflict detection from resolution time, we provide *conflict summary tables* (CSTs) that record the occurrence of conflicts without necessarily forcing immediate resolution. More specifically, CSTs indicate the *transactions* that conflict, rather than the locations on which they conflict. This information concisely captures what a TM system needs to know in order to resolve conflicts at some potentially future time. Software can choose when to examine the tables and can use whatever other information it desires (e.g., priorities) to drive its resolution policy.

When a transaction commits, its speculative state is made visible to the rest of the system. To avoid the doomed transaction problem without software polling or sandboxing, conflicting transactions must be alerted and aborted immediately. We enable such aborts with a mechanism known as *alert-on-update* (AOU). This mechanism adds one extra bit, set under software control, to each tag in the cache. When the cache controller detects a remote write of a line whose bit is set, it notifies the local processor, effecting an immediate branch to a previously registered handler. This mechanism can be very lightweight, since the handler invocation is entirely at the user level. By ensuring immediate aborts, AOU avoids the need for validation, thereby eliminating a large fraction of the cost for the metadata checks shown in Figure 2. By choosing what (data, metadata, or transaction status word) and when (at access or commit time) cache lines are tagged as AOU, software can choose between object-based and block-based granularity and among eager, mixed, and lazy conflict resolution.

Using AOU, PDI, signatures, and CSTs, we have developed a series of software-controlled, hardware-accelerated TM systems. RTM-Lite [15, 20] uses AOU alone for validation and conflict detection in a software TM framework (RSTM [18]). RTM-Lite is able to achieve up to a 5x speedup over RSTM on a single thread. RTM [15] uses both AOU and PDI to eliminate validation and versioning/copy overhead for transactions that fit in the cache. RTM is able to achieve up to an 8.7x speedup over RSTM. At the same time, it achieves only 35%–50% of the single-thread throughput of coarse-grain locks. The remaining overhead is due to software metadata updates and to the indirection needed for compatibility with transactions that fall back to software after overflowing the cache space available to PDI.

FlexTM [16] uses all four mechanisms to achieve flexible policy control without the need for software-managed metadata. The resulting single-thread performance is close to that of coarse-grain locks, demonstrating that eliminating per-access software overheads is essential to realizing the full potential of TM. Scalability is also improved relative to RTM-Lite and RTM. In contrast to other systems supporting lazy conflict resolution (e.g., TCC), FlexTM avoids the need for commit-time conflict detection: A processor's CSTs, which are purely local structures, identify the transactions with which the running transaction conflicts. Software can easily iterate through those transactions, aborting each. Experimental results [15–17, 22] confirm the ability to improve throughput by tailoring conflict resolution time and contention management policy based on application access patterns and overall system goals. The decoupled nature of the various hardware mechanisms also allows them to be used for a variety of non–TM-related tasks, including debugging, security, fast locks, and active messages.

Conclusion

The goal of Transactional Memory is to simplify synchronization in shared-memory parallel programs. Pure software approaches to implementing TM systems suffer from performance limitations. In this article, we presented an overview of emerging hardware support for TM that enhances performance, but with some limitations. The technology is still in its infancy, and widespread adoption will depend on the ability to support a wide spectrum of application behaviors and system requirements. Enforcing a single policy choice at design time precludes this flexibility. Hence, we advocate hardware acceleration of TM systems that leave policy in software. We described a set of mutually independent (decoupled) hardware mechanisms consistent with this approach and presented a series of systems that use this hardware to eliminate successive sources of software TM overhead. Decoupling facilitates incremental development by hardware vendors and leads to mechanisms useful not only for TM, but for various other purposes as well [15, 16, 19].

Several challenges remain. We need developers to integrate TM with existing systems, introduce new language constructs, and develop the necessary toolchains. We also need to support composability and allow existing libraries to coexist with TM. Finally, we need to resolve a variety of challenging semantic issues, through a combination of formalization and experience with realistic applications. We hope this article will help to foster that process by stimulating broader interest in the promise of transactional memory.

ACKNOWLEDGMENTS

This work was supported in part by NSF Grant Nos. CCF-0702505, CNS-0411127, CNS-0615139, CNS-0834451, and CNS-0509270 and by NIH Grant Nos. 5 R21 GM079259-02 and 1 R21 HG004648-01.

REFERENCES

[1] C.S. Ananian, K. Asanovic, B.C. Kuszmaul, C.E. Leiserson, and S. Lie, "Unbounded Transactional Memory," *Proc. of the 11th Int'l Symp. on High Performance Computer Architecture*, San Francisco, CA, Feb. 2005.

[2] L. Baugh, N. Neelakantan, and C. Zilles, "Using Hardware Memory Protection to Build a High-Performance, Strongly Atomic Hybrid Transactional Memory," *Proc. of the 35th Int'l Symp. on Computer Architecture*, Beijing, China, June 2008.

[3] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas, "Bulk Disambiguation of Speculative Threads in Multiprocessors," *Proc. of the 33rd Int'l Symp. on Computer Architecture*, Boston, MA, June 2006.

[4] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, "Hybrid Transactional Memory," *Proc. of the 12th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 2006.

[5] D. Dice, O. Shalev, and N. Shavit, "Transactional Locking II," *Proc. of the 20th Int'l Symp. on Distributed Computing*, Stockholm, Sweden, Sept. 2006.

[6] S. Diestelhorst and M. Hohmuth, "Hardware Acceleration for Lock-Free Data Structures and Software-Transactional Memory," presented at Workshop on Exploiting Parallelism with Transactional Memory and Other Hardware Assisted Methods (EPHAM), Boston, MA, Apr. 2008 (in conjunction with CGO).

[7] A. Gonsalves, "Sun Delays Rock Processor by a Year," *EE Times*, 7 Feb. 2008: http://www.eetimes.com/rss/showArticle.jhtml?articleID=206106243.

[8] L. Hammond, V. Wong, M. Chen, B. Hertzberg, B. Carlstrom, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional Memory Coherence and Consistency," *Proc. of the 31st Int'l Symp. on Computer Architecture*, Munich, Germany, June 2004.

[9] M. Herlihy and J.E. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," *Proc. of the 20th Int. Symp. on Computer Architecture*, San Diego, CA, May 1993. Expanded version available as CRL 92/07, DEC Cambridge Research Laboratory, Dec. 1992.

[10] S. Kumar, M. Chu, C.J. Hughes, P. Kundu, and A. Nguyen, "Hybrid Transactional Memory," *Proc. of the 11th ACM Symp. on Principles and Practice of Parallel Programming*, New York, March 2006.

[11] V.J. Marathe, W.N. Scherer III, and M.L. Scott, "Adaptive Software Transactional Memory," *Proc. of the 19th Int'l Symp. on Distributed Computing*, Cracow, Poland, Sept. 2005.

[12] C.C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford Transactional Applications for Multi-Processing," *Proc. of the 2007 IEEE Int'l Symp. on Workload Characterization*, Seattle, WA, Sept. 2008.

[13] R. Rajwar and J. R. Goodman, "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution," *Proc. of the 34th Int'l Symp. on Microarchitecture*, Austin, TX, Dec. 2001.

[14] R. Rajwar, M. Herlihy, and K. Lai, "Virtualizing Transactional Memory," *Proc. of the 32nd Int'l Symp. on Computer Architecture*, Madison, WI, June 2005.

[15] A. Shriraman, M.F. Spear, H. Hossain, S. Dwarkadas, and M.L. Scott, "An Integrated Hardware-Software Approach to Flexible Transactional Memory," *Proc. of the 34th Int'l Symp. on Computer Architecture*, San Diego, CA, June

2007. Earlier but expanded version available as TR 910, Dept. of Computer Science, Univ. of Rochester, Dec. 2006.

[16] A. Shriraman, S. Dwarkadas, and M.L. Scott, “Flexible Decoupled Transactional Memory Support,” *Proc. of the 25th Int’l Symp. on Computer Architecture*, Beijing, China, June 2008. Earlier version available as TR 925, Dept. of Computer Science, Univ. of Rochester, Nov. 2007.

[17] A. Shriraman and S. Dwarkadas, TR 939, Dept. of Computer Science, Univ. of Rochester, Sept. 2008.

[18] M.F. Spear, V.J. Marathe, W.N. Scherer III, and M.L. Scott, “Conflict Detection and Validation Strategies for Software Transactional Memory,” *Proc. of the 20th Int’l Symp. on Distributed Computing*, Stockholm, Sweden, Sept. 2006.

[19] M.F. Spear, A. Shriraman, H. Hossain, S. Dwarkadas, and M.L. Scott, “Alert-on-Update: A Communication Aid for Shared Memory Multiprocessors” (poster paper), *Proc. of the 12th ACM Symp. on Principles and Practice of Parallel Programming*, San Jose, CA, Mar. 2007.

[20] M.F. Spear, A. Shriraman, L. Dalessandro, S. Dwarkadas, and M.L. Scott, “Nonblocking Transactions without Indirection Using Alert-on-Update,” *Proc. of the 19th Annual ACM Symp. on Parallelism in Algorithms and Architectures*, San Diego, CA, June 2007.

[21] M.F. Spear, M.M. Michael, and C. von Praun, “RingSTM: Scalable Transactions with a Single Atomic Instruction,” *Proc. of the 20th Annual ACM Symp. on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[22] M.F. Spear, L. Dalessandro, V.J. Marathe, and M.L. Scott, “Fair Contention Management for Software Transactional Memory,” *Proc. of the 14th ACM Symp. on Principles and Practice of Parallel Programming*, Raleigh, NC, Feb. 2009.

[23] J. M. Stone, H.S. Stone, P. Heidelberger, and J. Turek, “Multiple Reservations and the Oklahoma Update,” *IEEE Parallel and Distributed Technology*, 1(4):58–71, Nov. 1993.

[24] M. Tremblay and S. Chaudhry, “A Third-Generation 65 nm 16-Core 32-Thread Plus 32-Scout-Thread CMT,” *Proc. of the Int’l Solid State Circuits Conf.*, San Francisco, CA, Feb. 2008.

[25] L. Yen, J. Bobba, M.R. Marty, K.E. Moore, H. Valos, M.D. Hill, M.M. Swift, and D.A. Wood, “LogTM-SE: Decoupling Hardware Transactional Memory from Caches,” *Proc. of the 13th Int’l Symp. on High Performance Computer Architecture*, Phoenix, AZ, Feb. 2007.

DIWAKER GUPTA, SANGMIN LEE, MICHAEL VRABLE, STEFAN SAVAGE, ALEX C. SNOEREN, GEORGE VARGHESE, GEOFFREY M. VOELKER, AND AMIN VAHDAT

Difference Engine

Diwaker Gupta's Ph.D research focused on virtualization, network emulation, and large-scale system testing. He is also interested in cloud computing and Web applications. He is currently employed at Aster Data Systems.

diwaker@asterdata.com

Sangmin Lee is a Ph.D student in the Department of Computer Sciences at the University of Texas, Austin. His research interests include distributed computing and operating systems.

sangmin@cs.utexas.edu

Michael Vrable is pursuing a Ph.D. in computer science at the University of California, San Diego, and is advised by professors Stefan Savage and Geoffrey Voelker. He received an M.S. from UCSD and a B.S. from Harvey Mudd College.

mvrable@cs.ucsd.edu

Stefan Savage is an associate professor of computer science at the University of California, San Diego. He has a B.S. in history and reminds his colleagues of this fact any time the technical issues get too complicated.

savage@cs.ucsd.edu

Alex C. Snoeren is an associate professor in the Computer Science and Engineering Department at the University of California, San Diego. His research interests include operating systems, distributed computing, and mobile and wide-area networking.

snoeren@cs.ucsd.edu

George Varghese is a professor of computer science at the University of California, San Diego. Several algorithms he has helped develop have found their way into commercial systems including Linux (timing wheels), the Cisco GSR (DRR), and Microsoft Windows (IP lookups).

varghese@cs.ucsd.edu

Geoffrey M. Voelker is an associate professor of computer science and engineering at the University of California, San Diego. He works in computer systems and networking.

voelker@cs.ucsd.edu

Amin Vahdat is a professor at the University of California, San Diego. His research focuses broadly on computer systems, including distributed systems, networks, and operating systems.

vahdat@cs.ucsd.edu

VIRTUALIZATION TECHNOLOGY HAS improved dramatically over the past decade and has now become pervasive within the service-delivery industry. Virtual machines are particularly attractive for server consolidation. Their strong resource and fault-isolation guarantees allow multiplexing of hardware among individual services, each configured with a custom operating system. Although physical CPUs are frequently amenable to multiplexing, main memory is not. Thus, memory is often the primary bottleneck to increasing the degree of multiplexing in enterprise and data center settings. Difference Engine [1] enables virtual machine (VM) monitors to allocate more machine memory for VMs than is present in the system, by using aggressive memory sharing techniques. As with VMware ESX server, Difference Engine shares identical memory pages. In addition, Difference Engine also shares pages with only partial content overlap and compresses infrequently used pages, enabling it to further improve memory savings by up to a factor of 2.5 compared to identical page sharing alone in VMware ESX server.

With main memory as a consolidation bottleneck, researchers and commercial VM software vendors have developed techniques to decrease the memory requirements for virtual machines. The VMware ESX server implements content-based page sharing, in which virtual pages in different VMs have identical content and therefore can share the same machine page copy-on-write. Identical page sharing has been shown to reduce the memory footprint of multiple, homogeneous virtual machines by 10%–40% [2]. We found, however, that the benefits of identical page sharing decline rapidly when more heterogeneous guest VMs are used.

The premise of this work is that there are significant additional benefits from sharing at a sub-page granularity (i.e., there are many pages that are *nearly* identical). We show that it is possible to efficiently find such similar pages and to coalesce them into a much smaller memory footprint. Among the set of similar pages, we are able to store most as patches relative to a single baseline page.

We also compress those pages that are unlikely to be accessed in the near future. In both patching and compression, Difference Engine relies on finding pages that are less frequently used to offset the cost of recovering these pages. To support these techniques, we added a swapping service so that even when memory has been oversubscribed (by allocating more memory than exists), all VM guests will have access to the memory their OS was configured to use by leveraging disk as secondary storage.

Difference Engine provides these benefits without negatively impacting application performance: in our experiments across a variety of workloads, Difference Engine imposes less than 7% execution time overhead. In return, we further show that Difference Engine can take advantage of the improved memory efficiency to increase aggregate system performance by utilizing the free memory to create additional virtual machines in support of a target workload. Thus, for a prototypical Internet service workload, Difference Engine is able to use the additional memory to increase maximum request throughput by nearly 40%.

Architecture

Difference Engine uses three distinct mechanisms that work together to realize the benefits of memory sharing, as shown in Figure 1. In this example, two VMs have allocated five pages total, each initially backed by distinct pages in machine memory (Figure 1a). For brevity, we only show how the mapping from guest physical memory to machine memory changes; the guest virtual to guest physical mapping remains unaffected. First, for identical pages across the VMs, we store a single copy and create references that point to the original. In Figure 1b, one page in VM-2 is identical to one in VM-1. For pages that are similar but not identical, we store a

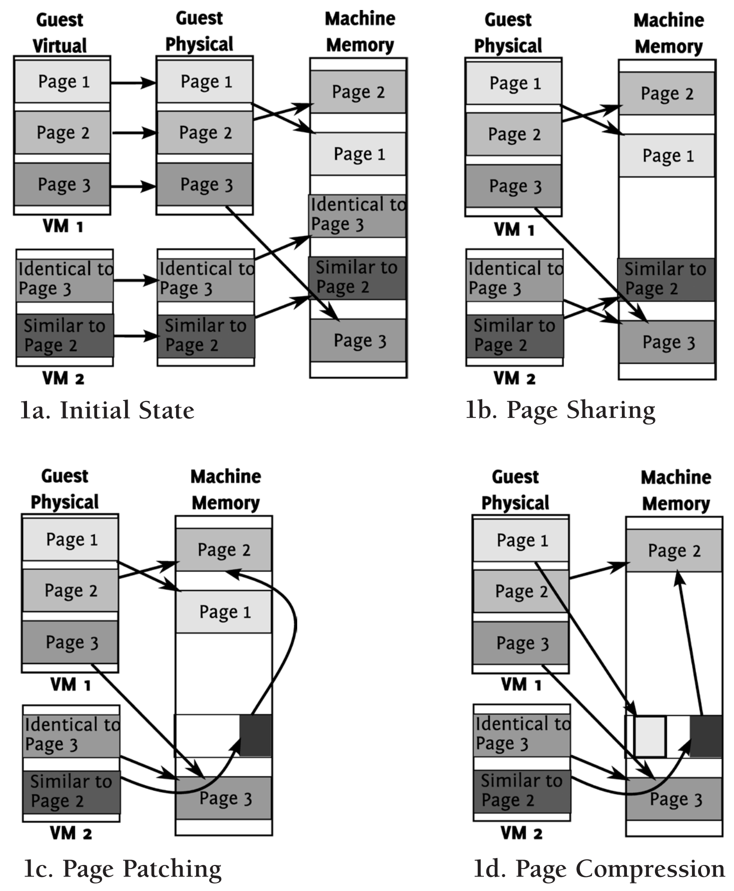


FIGURE 1: THE INITIAL STATE AND THE THREE DIFFERENT MEMORY CONSERVATION TECHNIQUES EMPLOYED BY DIFFERENCE ENGINE: PAGE SHARING, PAGE PATCHING, AND COMPRESSION. IN THIS EXAMPLE, FIVE PHYSICAL PAGES ARE STORED IN LESS THAN THREE MACHINE MEMORY PAGES FOR A SAVINGS OF ROUGHLY 50%.

patch against a reference page and discard the redundant copy. In Figure 1c, the second page of VM-2 is stored as a patch to the second page of VM-1. Finally, for pages that are unique and infrequently accessed, we compress them in memory to save space. In Figure 1d, the remaining private page in VM-1 is compressed. The actual machine memory footprint is now less than three pages, down from five pages originally.

In all three cases, efficiency concerns require us to select candidate pages that are unlikely to be accessed in the near future. We employ a global clock that scans memory in the background, identifying pages that have not been recently used. In addition, reference pages for sharing or patching must be found quickly without introducing performance overhead. Difference Engine uses full-page hashes and hash-based fingerprints to identify good candidates. Finally, we implement a demand paging mechanism that supplements main memory by writing VM pages to disk to support overcommitment, allowing the total memory required for all VMs to temporarily exceed the machine memory capacity.

Page Sharing

Difference Engine's implementation of content-based page sharing is similar to those in earlier systems. We walk through memory looking for identical pages. As we scan memory, we hash each page and index it based on its hash value. Identical pages hash to the same value and a collision indicates that a potential matching page has been found. We perform a byte-by-byte comparison to ensure that the pages are indeed identical before sharing them.

Upon identifying target pages for sharing, we reclaim one of the pages and update the virtual memory to point at the shared copy. Both mappings are marked read-only, so that writes to a shared page cause a page fault that will be trapped by the virtual machine monitor (VMM). The VMM returns a private copy of the shared page to the faulting VM and updates the virtual memory mappings appropriately. If no VM refers to a shared page, the VMM reclaims it and returns it to the free memory pool.

Patching

Traditionally, the goal of page sharing has been to eliminate redundant copies of identical pages. Difference Engine considers further reducing the memory required to store similar pages by constructing patches that represent a page as the difference relative to a reference page.

One of the principal complications with subpage sharing is identifying candidate reference pages. Difference Engine uses a parametrized scheme to identify similar pages based upon the hashes of several 64-byte portions of each page. In particular, HashSimilarityDetector(k,s) hashes the contents of ($k \times s$) 64-byte blocks at randomly chosen locations on the page and then groups these hashes together into k groups of s hashes each. We use each group as an index into a hash table.

Higher values of s capture local similarity, whereas higher k values incorporate global similarity. Hence, HashSimilarityDetector(1,1) will choose one block on a page and index that block; pages are considered similar if that block of data matches. HashSimilarityDetector(1,2) combines the hashes from two different locations in the page into one index of length two. HashSimilarityDetector(2,1) instead indexes each page twice: once based on the contents of a first block, and again based on the contents of a second block.

Pages that match at least one of the two blocks are chosen as candidates. Through experimentation, we discovered that HashSimilarityDetector(2,1) with one candidate does surprisingly well. There is a substantial gain from hashing two distinct blocks in the page separately, but little additional gain by hashing more blocks.

Difference Engine indexes a page by hashing 64-byte blocks at two fixed locations in the page (chosen at random) and uses each hash value as a separate index to store the page in the hash table. To find a candidate similar page, the system computes hashes at the same two locations, looks up those hash table entries, and calculates the page patch to determine memory savings if it finds a match for either of the indexed blocks.

Our current implementation uses 18-bit hashes to keep the hash table small to cope with the limited size of the Xen heap. In general, though, larger hashes might be used for improved savings and fewer collisions. Our analysis suggests, however, that the benefits from increasing the hash size are modest.

Compression

Finally, for pages that are not significantly similar to other pages in memory, we consider compressing them to reduce the memory footprint. Compression is useful only if the compression ratio is reasonably high and, like patching, if selected pages are accessed infrequently. Otherwise, the overhead of compression/decompression will outweigh the benefits. We identify candidate pages for compression using a global clock algorithm (see "Clock," below), assuming that pages that have not been recently accessed are unlikely to be accessed in the near future.

Difference Engine supports multiple compression algorithms, currently LZO and WKdm as described in Wilson et al. [3]; we invalidate compressed pages in the VM and save them in a dynamically allocated storage area in machine memory. When a VM accesses a compressed page, Difference Engine decompresses the page and returns it to the VM uncompressed. It remains there until it is again considered for compression.

Paging Machine Memory

Although Difference Engine will deliver some (typically high) level of memory savings, in the worst case all VMs might actually require all of their allocated memory. Setting aside sufficient physical memory to account for this prevents Difference Engine from using the memory to create additional VMs. Not doing so, however, may result in temporarily overshooting the physical memory capacity of the machine and causing a system crash. We therefore require a demand-paging mechanism to supplement main memory by writing pages out to disk in such cases.

A good candidate page for swapping out should not be accessed in the near future—the same requirement as compressed/patched pages. In fact, Difference Engine also considers compressed and patched pages as candidates for swapping out. Once the contents of the page are written to disk, the page can be reclaimed. When a VM accesses a swapped-out page, Difference Engine fetches it from disk and copies the contents into a newly allocated page that is mapped appropriately in the VM's memory.

Since disk I/O is involved, swapping in/out is an expensive operation. Further, a swapped page is unavailable for sharing or as a reference page for patching. Therefore, swapping should be an infrequent operation. Difference

Engine implements the core mechanisms for paging and leaves policy decisions, such as when and how much to swap, to user-level tools.

Implementation

We have implemented Difference Engine in the Xen 3.0.4 VMM in roughly 14,500 lines of code. An additional 20,000 lines come from ports of existing patching and compression algorithms (Xdelta, LZO, WKdm) to run inside Xen.

Xen and other platforms that support fully virtualized guests use a mechanism called "shadow page tables" to manage guest OS memory [2]. The guest OS has its own copy of the page table that it manages, believing that they are the hardware page tables, though in reality they are just a map from the guest's virtual memory to its notion of physical memory (V2P map). In addition, Xen maintains a map from the guest's notion of physical memory to the machine memory (P2M map). The shadow page table is a cache of the results of composing the V2P map with the P2M map, mapping guest virtual memory directly to machine memory.

Difference Engine relies on manipulating P2M maps and the shadow page tables to interpose on page accesses. For simplicity, we do not consider any pages mapped by Domain-0 (the privileged, control domain in Xen), which, among other things, avoids the potential for circular page faults.

Clock

Difference Engine implements a not-recently-used (NRU) policy [4] to select candidate pages for sharing, patching, compression, and swapping out. On each invocation, the clock scans a portion of machine memory, checking and clearing the referenced (R) and modified (M) bits on pages. Thus, pages with the R or the M bit set must have been referenced or modified since the last scan. We ensure that successive scans of memory are separated by at least four seconds in the current implementation, to give domains a chance to set the R/M bits on frequently accessed pages. In the presence of multiple VMs, the clock scans a small portion of each VM's memory in turn for fairness. The external API exported by the clock is simple: Return a list of pages (of some maximum size) that have not been accessed in some time.

In OSes running on bare metal, the R/M bits on page-table entries are typically updated by the processor. Xen structures the P2M map exactly like the page tables used by the hardware. However, since the processor does not actually use the P2M map as a page table, the R/M bits are not updated automatically. We modify Xen's shadow page table code to set these bits when creating readable or writable page mappings. Unlike conventional operating systems, where there may be multiple sets of page tables that refer to the same set of pages, in Xen there is only one P2M map per domain. Hence, each guest page corresponds unambiguously to one P2M entry and one set of R/M bits.

Real-World Applications

We now present the performance of Difference Engine on a variety of workloads. We seek to answer two questions. First, how effective are the memory-saving mechanisms at reducing memory usage for real-world applications? Second, what is the impact of those memory-sharing mechanisms on system performance? Since the degree of possible sharing depends on the software configuration, we consider several different cases of application mixes.

To put our numbers in perspective, we conduct head-to-head comparisons with VMware ESX Server for three different workload mixes. We run ESX Server 3.0.1 build 32039 on a Dell PowerEdge 1950 system. Note that even though this system has two 2.3-GHz Intel Xeon processors, our VMware license limits our usage to a single CPU. We therefore restrict Xen (hence, Difference Engine) to use a single CPU for fairness. We also ensure that the OS images used with ESX match those used with Xen, especially the file system and disk layout. Note that we are only concerned with the effectiveness of the memory sharing mechanism, not in comparing the application performance across the two hypervisors. Further, we configure ESX to use its most aggressive page sharing settings, in which it scans 10,000 pages/second (compared to its default of 200); we configure Difference Engine similarly.

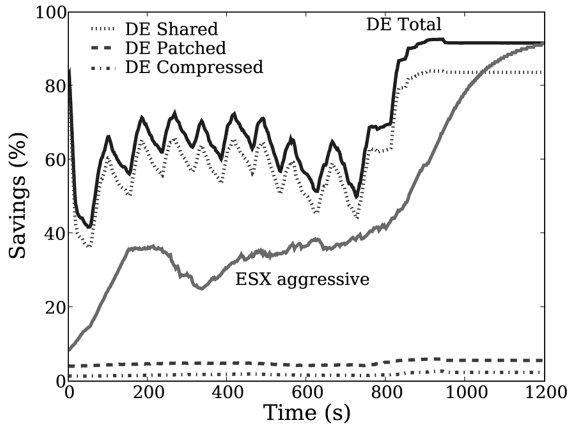


FIGURE 2: FOUR IDENTICAL VMS EXECUTE DBENCH. FOR SUCH HOMOGENEOUS WORKLOADS, BOTH DIFFERENCE ENGINE AND ESX EVENTUALLY YIELD SIMILAR SAVINGS, BUT DE EXTRACTS MORE SAVINGS WHILE THE BENCHMARK IS IN PROGRESS.

In our first set of benchmarks, we test the base scenario where all VMs on a machine run the same OS and applications. This scenario is common in cluster-based systems where several services are replicated to provide fault tolerance or load balancing. Our expectation is that significant memory savings are available and that most of the savings will come from page sharing. The graphs shown in Figures 2–4 break out the contributions in Difference Engine by page compression (the least), patching, and page sharing (the most) against page sharing in ESX Server.

We set up four 512-MB virtual machines running Debian 3.1. Each VM executes dbench for 10 minutes followed by a stabilization period of 20 minutes. Figure 2 shows the amount of memory saved as a function of time. First, note that eventually both ESX and Difference Engine reclaim roughly the same amount of memory (with the graph for ESX plateauing beyond 1,200 seconds). However, while dbench is executing, Difference Engine delivers approximately 1.5 times the memory savings achieved by ESX. As before, the bulk of Difference Engine savings comes from page sharing for the homogeneous workload case.

We used two different sets of guests VMs for testing heterogeneous performance.

- MIXED-1: Windows XP SP1 hosting RUBiS; Debian 3.1 compiling the Linux kernel; Slackware 10.2 compiling Vim 7.0 followed by a run of the lmbench benchmark.

- MIXED-2: Windows XP SP1 running Apache 2.2.8 hosting approximately 32,000 static Web pages crawled from Wikipedia, with httperf running on a separate machine requesting these pages; Debian 3.1 running the SysBench database benchmark using 10 threads to issue 100,000 requests; Slackware 10.2 running dbench with 10 clients for six minutes followed by a run of the IOZone benchmark.

Figures 3 and 4 show the memory savings as a function of time for the two heterogeneous workloads, MIXED-1 and MIXED-2. We make the following observations. First, in steady state, Difference Engine delivers a factor of 1.6 to 2.5 more memory savings than ESX. For instance, for the MIXED-2 workload, Difference Engine could host the three VMs allocated 512 MB of physical memory each in approximately 760 MB of machine memory; ESX would require roughly 1100 MB of machine memory. The remaining, significant, savings come from patching and compression. And these savings come at a small cost. The baseline configuration is regular Xen without Difference Engine. In all cases, performance overhead of Difference Engine is within 7% of the baseline. For the same workload, we find that performance under ESX with aggressive page sharing is also within 5% of the ESX baseline with no page sharing.

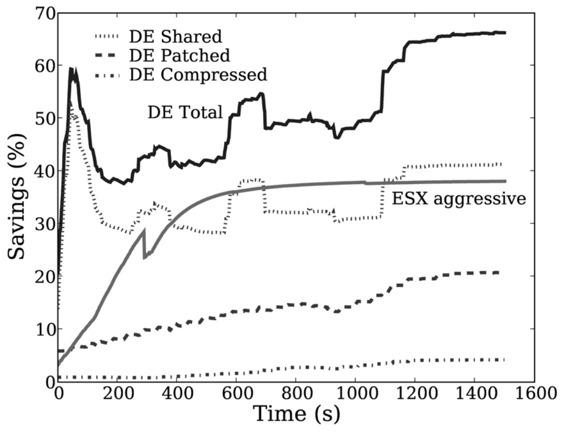


FIGURE 3: MEMORY SAVINGS FOR MIXED-1. DIFFERENCE ENGINE SAVES UP TO 45% MORE MEMORY THAN ESX.

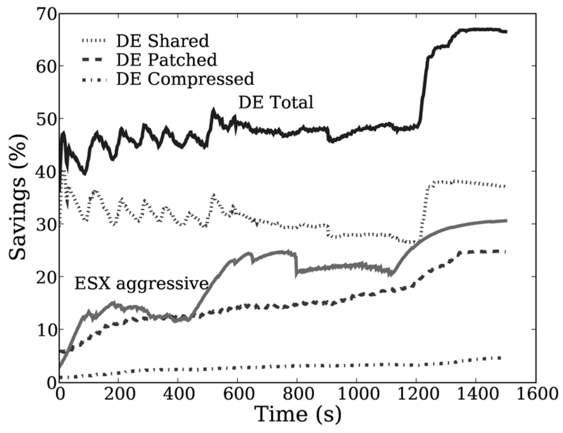


FIGURE 4: MEMORY SAVINGS FOR MIXED-2. DIFFERENCE ENGINE SAVES ALMOST TWICE AS MUCH MEMORY AS ESX.

Conclusion

One of the primary bottlenecks to higher degrees of virtual machine multiplexing is main memory. Earlier work shows that substantial memory savings are available from harvesting identical pages across virtual machines when running homogeneous workloads. The premise of this work is that there are significant additional memory savings available from locating and patching similar pages and in-memory page compression. We present the design and evaluation of Difference Engine to demonstrate the potential memory savings available from leveraging a combination of whole page sharing, page patching, and compression. We discuss our experience addressing a number of technical challenges, including algorithms to quickly identify candidate pages for patching, demand paging to support oversubscription of total assigned physical memory, and a clock mechanism to identify appropriate target machine pages for sharing, patching, compression, and paging. Our performance evaluation shows that Difference Engine delivers an additional factor of 1.6 to 2.5 more memory savings than VMware ESX Server for a variety of workloads, with minimal performance overhead. Difference Engine mechanisms might also be used to improve single OS memory management; we leave such exploration to future work.

ACKNOWLEDGMENTS

We would like to particularly thank Rik Farrow for crafting a condensed draft of this article from our conference paper. In the course of the project, we also received invaluable assistance from a number of people at VMware. We would like to thank Carl Waldspurger, Jennifer Anderson, and Hemant Gaidhani, and the Performance Benchmark group for feedback and discussions on the performance of ESX server. Also, special thanks are owed to Kiran Tati for assisting with ESX setup and monitoring and to Emil Sit for providing insightful feedback on the paper. Finally, we would like to thank Michael Mitzenmacher for his assistance with min-wise hashing, our shepherd Fred Douglis for his insightful feedback and support, and the anonymous OSDI ’08 reviewers for their valuable comments. This work was supported in part by NSF CSR-PDOS Grant No. CNS-0615392, the UCSD Center for Networked Systems (CNS), and UC Discovery Grant 07-10237. Vrable was supported in part by an NSF Graduate Research Fellowship.

REFERENCES

[1] D. Gupta, S. Lee, M. Vrable, S. Savage, A.C. Snoeren, G. Varghese, G.M. Voelker, and A. Vahdat, “Difference Engine: Harnessing Memory Redundancy in Virtual Machines,” *Proceedings of OSDI ’08*: http://www.usenix.org/events/osdi08/tech/full_papers/gupta/gupta_html/.

[2] C.A. Waldspurger, “Memory Resource Management in VMware ESX Server,” *Proceedings of OSDI ’02*: http://www.usenix.org/publications/library/proceedings/osdi02/tech/waldspurger.html.

[3] P.R. Wilson, S.F. Kaplan, and Y. Smaragdakis, “The Case for Compressed Caching in Virtual Memory Systems,” *Proceedings of the 1999 USENIX Annual Technical Conference*: http://www.usenix.org/publications/library/proceedings/usenix99/full_papers/wilson/wilson_html/.

[4] A.S. Tanenbaum, *Modern Operating Systems* (Englewood Cliffs, NJ: Prentice Hall, 2007).

JOHN DOUCEUR, JEREMY ELSON, JON HOWELL, AND JACOB R. LORCH, WITH RIK FARROW

leveraging legacy code for Web browsers

John Douceur manages the Distributed Systems Research Group in the Redmond lab of Microsoft Research. His interests are designing algorithms, data structures, and protocols for distributed systems.

johndo@microsoft.com

Jeremy Elson has worked in sensor networks, distributed systems, and occasional hare-brained schemes. He also enjoys flying and likes bicycling to work.

jelson@microsoft.com

Jon Howell works at the intersection of security and scalability in distributed systems. His recent projects focus on the convergence of utility computing and Web-delivered applications.

howell@microsoft.com

Jacob Lorch is a Researcher in the Systems and Networking group at Microsoft Research. His research interests include distributed systems, online games, Web security, and energy management.

lorch@microsoft.com

WEB BROWSERS HAVE BECOME A DE facto user interface for many online applications. But because browser applications are typically written in specialized Web languages, the vast quantity of existing tools, libraries, and applications are unavailable to Web developers. Xax provides a secure execution container that can run legacy code written in arbitrary languages. With a small porting effort, legacy applications can be turned into Xax applications, which execute natively but independently of the underlying OS.

Modern Web applications are driving toward the power of fully functional desktop applications such as email clients (e.g., Gmail, Hotmail, Outlook Web Access) and productivity apps (e.g., Google Docs). Web applications offer two significant advantages over desktop apps: security—in that the user's system is protected from buggy or malicious applications—and OS independence. Both of these properties are normally provided by a virtual execution environment that implements a type-safe language, such as JavaScript, Flash, or Silverlight. However, this mechanism inherently prohibits the use of non-type-safe legacy code. Since the vast majority of extant desktop applications and libraries are not written in a type-safe language, the enormous base of legacy code is currently unavailable to the developers of Web applications.

In a paper published at OSDI '08 [1], the authors demonstrated running the GhostScript PDF viewer, the eSpeak speech synthesizer, a Python interpreter, and an OpenGL demo that renders 3D animation. In total, it took roughly two person-weeks of effort to port 3.3 million lines of code to use the simple Xax interface. This existing code was written in several languages and produced with various tool chains, and it runs in multiple browsers on multiple operating systems.

Xax provides native-code-level performance in a secure and OS-independent manner. Xax relies on four mechanisms:

- The picoprocess, a native-code execution abstraction that is secured via hardware memory isolation and a very narrow system-call interface, akin to a streamlined hardware virtual machine
- The Platform Abstraction Layer (PAL), which provides an OS-independent Application Binary Interface (ABI) to Xax picoprocesses

- Hooks to existing browser mechanisms to provide applications with system services, such as network communication, user interface, and local storage, that respect browser security policies via the Xax Monitor
- Lightweight modifications to existing tool chains and code bases, for retargeting legacy code to the Xax picoprocess environment

Picoprocess

Most operating systems rely on hardware memory protection mechanisms to isolate processes from one another. Process isolation prevents one process from interfering with another process by reading or writing its program code or data. But process-level isolation provides insufficient protection for running downloaded code within a browser, as the browser itself is a process owned by the user.

Browsers do run downloaded code, such as JavaScript, Java, and Silverlight, but these are type-safe languages. These languages are interpreted within the browser that enforces a security policy, for example, the Same Origin Policy and limited access to filesystems. Legacy code expects to have complete access to a system via the system call API, and thus it cannot be limited by the browser.

Xax introduces the abstraction of the picoprocess. A picoprocess can be thought of as a stripped-down virtual machine without emulated physical devices, MMU, or CPU kernel mode. Alternatively, a picoprocess can be thought of as a highly restricted OS process that is prevented from making kernel calls. In either view, a picoprocess is a single hardware-memory-isolated address space with strictly user-mode CPU execution and a very narrow interface to the world outside the picoprocess, as illustrated in Figure 1.

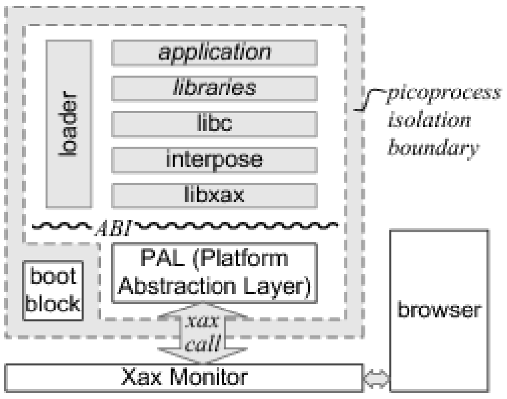


FIGURE 1: THE PICOPROCESS GETS ISOLATED WHEN THE BOOT BLOCK ARRANGES TO INTERCEPT FUTURE SYSTEM CALLS

The Xax Monitor is a user-mode process that creates, isolates, and manages each picoprocess and that provides the functionality of xaxcalls. The Xax Monitor launches the picoprocess, which runs as a user-level OS process, thus leveraging the hardware memory isolation that the OS already enforces on its processes. Before creating a new picoprocess, the Xax Monitor first allocates a region of shared memory, which will serve as a communication conduit between the picoprocess and the Monitor. Then the picoprocess is created as a child process of the Xax Monitor process.

This child process begins by executing an OS-specific boot block, which performs three steps. First, it maps the shared memory region into the child process's address space, thereby completing the communication conduit.

Second, it makes an OS-specific kernel call that permanently revokes the child process's ability to make subsequent kernel calls, thereby completing the isolation. Third, it passes execution to the OS-specific PAL, which in turn loads and passes execution to the Xax application.

The boot block is part of the TCB (Trusted Computing Base), even though it executes inside the child process. The boot block uses kernel mechanisms to control access to system calls. The Linux version uses the ptrace() system call, so that all subsequent system calls get trapped and passed to the Xax Monitor. If the Xax Monitor fails (exits), the picoprocess's system calls will no longer be trapped, a weakness in this present Linux implementation. Using ptrace() also hurts performance, as ptrace() was designed for debugging, with the kernel notifying the monitoring process when a system call is made *and* after the system call completes, but before results get passed back to the monitored process.

The Windows version makes a kernel call to establish an interposition on all subsequent syscalls via our XaxDrv driver. Because every Windows thread has its own pointer to a table of system call handlers, XaxDrv is able to isolate a picoprocess by replacing the handler table for that process's thread. The replacement table converts every user-mode syscall into an inter-process call (IPC) to the user-space Xax Monitor.

Platform Abstraction Layer

The Platform Abstraction Layer (PAL) translates the OS-independent ABI into the OS-specific xaxcalls of the Xax Monitor. The PAL is included with the OS-specific Xax implementation; everything above the ABI is native code delivered from an origin server. The PAL runs inside the Xax picoprocess, so its code is not trusted. Isolation is provided by the xaxcall interface (dashed border in Figure 1); the PAL merely provides ABI consistency across host operating systems (wiggly line in Figure 1).

For memory allocation and deallocation, the ABI provides two calls. The first:

```
void *xabi_alloc(void *start, long len);
```

maps len zero-filled bytes of picoprocess memory, starting at start if specified, and returns the address. Then:

```
int xabi_free(void *start);
```

frees the memory region beginning at start, which must be an address returned from xabi_alloc. It returns 0 for success or -1 for error.

As described in the next section, the picoprocess appears to the browser as a Web server, and communication is typically over HTTP. When the browser opens a connection to the picoprocess, this connection can be received by using this call:

```
int xabi_accept();
```

This returns a channel identifier, analogous to a UNIX file descriptor or a Windows handle, connected to an incoming connection from the browser. It returns -1 if no incoming connection is ready.

The picoprocess can also initiate connection to the origin server that provided the picoprocess application. To initiate a connection to the home server, the picoprocess uses the call:

```
int xabi_open_url(const char *method, const char *url);
```

This returns a channel identifier connected to the given URL, according to the specified method, which may be "get," "put," or "connect." It requests that the Xax Monitor fetch and cache the URL according to the Same Origin Policy (SOP) rules for the domain that provided the Xax picoprocess.

The operations that can be performed on an open channel are read, write, poll, and close. The read and write operations:

```
int xabi_read(int chnl, char *buf, int len);
int xabi_write(int chnl, char *buf, int len);
```

transfer data on an open channel and return the number of bytes transferred (0 if the channel is not ready, -1 if the channel is closed or failed). The poll operation:

```
int xabi_poll(xabi_poll_fd *pfds, int npfds, bool block);
```

indicates the ready status of a set of channels by updating events. If the value of block is true, it does not return until at least one requested event is ready, thereby allowing the picoprocess to yield the processor. It returns the number of events ready but does not return 0 if the value of block is true. Finally, the close operation:

```
int xabi_close(int chnl);
```

closes an open channel. It returns 0 for success or -1 for error.

During picoprocess boot, the loader needs to know the URL from which to fetch the application image. Xax uses a general loader that reads the application URL from the query parameters of the URL that launched the picoprocess. The following PAL call, which is normally used only by the loader, provides access to these parameters:

```
const char **xabi_args();
```

It returns a pointer to a NULL-terminated list of pointers to arguments specified at instantiation. (Note that there is no corresponding xaxcall; the parameters are written into the PAL during picoprocess initialization.)

Lastly, the ABI provides a call to exit the picoprocess when it is finished:

```
void xabi_exit();
```

Although the PAL runs inside the picoprocess, it is not part of the application. More pointedly, it is not delivered with the OS-independent application code. Instead, the appropriate OS-specific PAL remains resident on the client machine, along with the Xax Monitor and the Web browser, whose implementations are also OS-specific. When a Xax application is delivered to the client, the app and the PAL are loaded into the picoprocess and linked via a simple dynamic-linking mechanism: The ABI defines a table of function pointers and the calling convention for the functions.

A library called libxax exports a set of symbols (xabi_read, xabi_open_url, etc.) that obey the function linkage convention of the developer's tool chain. This shim converts each of these calls to the corresponding ABI call in the PAL. The shim thus provides a standard API to Xax applications.

The Xax Monitor

The Xax Monitor has the job of providing the services indicated by the xaxcall interface. Some of these services are straightforward for the Xax Monitor to perform directly, such as memory allocation/deallocation, access to URL query parameters, and picoprocess exit. The Xax Monitor also provides a communication path to the browser, via which the Xax picoprocess appears as a Web server. This communication path enables the Xax application to use read and write calls to serve HTTP to the browser. From the browser's perspective, these HTTP responses appear to come from the remote origin server that supplied the Xax app. It is clear that this approach is secure, since the Xax application is unable to do anything that the origin server could not have done by serving content directly over the Internet. The current Xax Monitor provides this browser interface by acting as a client-side proxy server.

Using the picoprocess-to-browser communication path, the Xax application can employ JavaScript code in the browser to perform functions on its behalf, such as user interface operations, DOM manipulation, and access to browser cookies. The evaluated applications employ a common design pattern: The Xax app sends an HTML page to the browser, and this page contains JavaScript stubs that translate messages from the picoprocess into JavaScript function invocations.

Lightweight Code Modification

Porting legacy applications took surprisingly little effort. This is surprising because the legacy code was written to run atop an operating system, so it was not obvious that the OS-specific code could be eliminated or replaced without crippling the applications. As an example, a quick test using graphviz and a Python interpreter found that this application made 2725 syscalls (39 unique). Porting this code to Xax would seem to require an enormous emulation of OS functionality. However, using lightweight modifications, it was possible to port this code, about a million lines, in just a few days.

Although the particular modifications required are application-dependent, they follow a design pattern that covers five common aspects: disabling irrelevant dependencies, restricting application interface usage, applying failure-oblivious computing techniques, internally emulating syscall functionality, and, when ultimately necessary, providing syscall functionality via xaxcalls.

The first step is to use compiler flags to disable dependencies on irrelevant components. Not all libraries and code components are necessary for use within the Web-application framework, and removing them reduces the download size of the Web app and also reduces the total amount of code that needs to be ported. For Python/graphviz, by disabling components such as pango and pthreads, 699 syscalls (16 unique) were eliminated.

The second step is to restrict the interfaces that the application uses. For instance, an app might handle I/O either via named files or via stdin/stdout, and the latter may require less support from the system. Restricting the interface is achieved in various ways, such as by setting command-line arguments or environment variables. For Python/graphviz, an entry-point parameter that changes the output method from "xlib" to "svg" was used, eliminating 367 syscalls (21 unique).

The third step is to identify which of the application's remaining system calls can be handled trivially. In some cases, it is adequate to return error codes indicating failure, in a manner similar to failure-oblivious computing [2]. For Python/graphviz, it was sufficient to simply reject 125 syscalls (11 unique: getuid32, rt_sigaction, fstat64, rt_sigprocmask, ioctl, uname, gettimeofday, connect, time, fcntl64, and socket).

The fourth step is to emulate syscall functionality within the syscall interposition layer (see Figure 1). For instance, Python/graphviz reads Python library files from a file system at runtime. The authors packaged these library files as a tarball and emulated a subset of filesystem calls using libtar to access the libraries. The tarball is read-only, which is all Python/graphviz requires. For some of the other ported applications, the authors also provided read/write access to temporary files by creating a RAM disk in the interposition layer. Code in the interposition layer looks at the file path to determine whether to direct calls to the tarball, to the RAM disk, or to somewhere else, such as a file downloaded from the origin server. For Python/graphviz, they used internal emulation to satisfy 1409 syscalls (14 unique), 943 of which fail obliviously.

The fifth and final step is to provide real backing functionality for the remaining system calls via the Xax ABI. For Python/graphviz, most of the remaining syscalls are for user input and display output, which get routed to the UI in the browser. The authors provided this functionality for the remaining 137 syscalls (11 unique: setsockopt, listen, accept, bind, read, write, brk, close, mmap2, old_mmap, and munmap).

The first three steps are application-specific, but for the final two steps, much of the syscall support developed for one app can be readily reused for other apps. The internally emulated tar-based file system was written to support eSpeak and later reused to support Python. Similarly, the backing functionality for the mmap functions and networking functions (listen, accept, bind . . .) are used by all of the example applications.

For any given application, once the needed modifications are understood, the changes become mechanical. Thus, it is fairly straightforward for a developer to maintain both a desktop version and a Xax version of an app, using a configure flag to specify the build target. This is already a common practice for a variety of applications that compile against Linux, BSD, and Win32 syscall interfaces.

Performance

To evaluate performance, microbenchmarks and macrobenchmarks were run to measure CPU- and I/O-bound performance. All measurements were done on a 2.8-GHz Intel Pentium 4.

Xax's use of native CPU execution, adopted to achieve legacy support, also leads to native CPU performance. The first microbenchmark [Table 1, column (a)] computed the SHA-1 hash of H.G. Wells's *The War of the Worlds*. Xax performs comparably to the Linux native host. The Windows native binary was compiled with a different compiler (Visual Studio versus gcc), likely producing the improved performance of the Windows native cases over Xax.

Environment	Tool	Computation	Syscall	Allocation
		SHA-1	close	16 MB
		(a)	(b)	(c)
Linux native	gcc	5,930,000	430	27,120
Linux Xax	gcc	5,970,000	69,400	202,600
XP native	VS	4,540,000	1,126	31,390
XP Xax	gcc	6,170,000	16,880	235,300
Vista native	VS	4,580,000	1,316	40,900
Vista Xax	gcc	6,490,000	59,900	612,000

TABLE 1: MICROBENCHMARKS IN UNITS OF MACHINE CYCLES, 1/(2.8X10^9) SEC; MAX [(SIGMA)/(MU)] = 6.6%

The benefits of native execution allowed the authors to accept overheads associated with hardware context switching. However, the simple noninvasive user-level implementations lead to quite high overheads. Table 1, column (b) reports the cost of a null xaxcall compared with a null native system call (close(-1)). Table 1, column (c) reports the cost of allocating an empty 16-MB memory region. The Xax overhead runs 7–161x.

Limitations and Future Work

For related work, we refer you to Section 7 of the OSDI paper [1]. In terms of security, the authors argue that Xax is secure by its small TCB. However, a production implementation deserves a rigorous inspection to ensure both that the kernel syscall dispatch path for a picoprocess is indeed closed and that no other kernel paths, such as exception handling or memory management, are exploitable by a malicious picoprocess. The authors suggest exploring alternative implementations that exclude more host OS code from the TCB, such as a Mac OS implementation that uses Mach processes or a VM-like implementation that completely replaces the processor trap dispatch table for the duration of execution of a picoprocess.

Rich Web applications, Xax or otherwise, will require browser support (such as remote differential compression) for efficiently handling large programs, and support for offline functionality. Because Xax applications access resources via the browser, any browser enhancements that deliver these features are automatically inherited by the Xax environment.

Integrating Xax with the browser using a proxy is expedient, but for several reasons it would be better to directly integrate with the browser. First, Xax currently rewrites the namespace of the origin server; this is an abuse of protocol. Instead, the browser should provide an explicit <embed> object with which a page can construct and name a picoprocess for further reference. Second, the proxy is unaware of when the browser has navigated away from a page, and when it is thus safe to terminate and reclaim a picoprocess. Third, the proxy cannot operate on https connections. For these reasons, the authors plan to integrate Xax directly into popular browsers.

Other issues include supporting conventional threading models, using a more mainstream C library such as glibc (here dietlibc was used), and using relocatable code (since Xax uses statically linked code).

Conclusions

Xax is a browser plug-in model that enables developers to adapt legacy code for use in rich Web applications, while maintaining security, performance, and OS independence.

- Xax's security comes from its use of the picoprocess minimalist isolation boundary and browser-based services; Xax's TCB is orders of magnitude smaller than alternative approaches.
- Xax's OS independence comes from its use of picoprocesses and its platform abstraction layer; Xax applications can be compiled on any toolchain and run on any OS host.
- Xax's performance derives from native code execution in picoprocesses.
- Xax's legacy support comes from lightweight code modification.

Over decades of software development in non-type-safe languages, vast amounts of design, implementation, and testing effort have gone into producing powerful legacy applications. By enabling developers to leverage this prior effort into a Web application deployment and execution model, we anticipate that Xax may change the landscape of Web applications.

REFERENCES

[1] John R. Douceur, Jeremy Elson, Jon Howell, and Jacob R. Lorch, "Leveraging Legacy Code to Deploy Desktop Applications on the Web": http://www.usenix.org/events/osdi08/tech/full_papers/douceur/douceur_html/index.html.

[2] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, T. Leu, and W.S. Beebee, Jr., "Enhancing Server Availability and Security through Failure-Oblivious Computing": http://www.usenix.org/events/osdi04/tech/rinard.html.

DAVID BEAZLEY

Python 3: the good, the bad, and the ugly

David Beazley is an open source software developer and the author of *Python Essential Reference* (4th edition, Addison-Wesley, 2009). He lives in Chicago, where he also teaches Python courses.

dave@dabeaz.com

IN LATE 2008, AND WITH MUCH FANfare, Python 3.0 was released into the wild. Although there have been ongoing releases of Python in the past, Python 3 is notable in that it intentionally breaks backwards compatibility with all previous versions. If you use Python, you have undoubtedly heard that Python 3 even breaks the lowly print statement—rendering the most simple "Hello World" program incompatible. And there are many more changes, with some key differences that no conversion program can deal with. In this article I give you a taste of what's changed, outline where those changes are important, and provide you with guidance on whether you want or need to move to Python 3 soon.

By the time you're reading this, numerous articles covering all of the new features of Python 3 will have appeared. It is not my intent to simply rehash all of that material here. In fact, if you're interested in an exhaustive coverage of changes, you should consult "What's New in Python 3?" [1]. Rather, I'm hoping to go a little deeper and to explain *why* Python has been changed in the way it has, the implications for end users, and why you should care about it. This article is not meant to be a Python tutorial; a basic knowledge of Python programming is assumed.

Python's C Programming Roots

The lack of type declarations and curly braces aside, C is one of the foremost influences on Python's basic design, including the fundamental operators, identifiers, and keywords. The interpreter is written in C and even the special names such as `__init__`, `__str__`, and `__dict__` are inspired by a similar convention in the C preprocessor (for example, preprocessor macros such as `__FILE__` and `__LINE__`). The influence of C is no accident—Python was originally envisioned as a high-level language for writing system administration tools. The goal was to have a high-level language that was easy to use and that sat somewhere between C programming and the shell.

Although Python has evolved greatly since its early days, a number of C-like features have remained in the language and libraries. For example, the integer math operators are taken straight from C—even truncating division just like C:

```
>>> 7/4
1
>>>
```

Python's string formatting is modeled after the C `printf()` class of functions. For example:

```
>>> print "%10s %10d %10.2f" % ('ACME',100,123.45)
  ACME 100    123.45
>>>
```

File I/O in Python is byte-oriented and really just a thin layer over the C stdio functionality:

```
>>> f = open("data.txt","r")
>>> data = f.read(100)
>>> f.tell()
100L
>>> f.seek(500)
>>>
```

In addition, many of Python's oldest library modules provide direct access to low-level system functions that you would commonly use in C systems programs. For example, the `os` module provides almost all of the POSIX functions and other libraries provide low-level access to sockets, signals, fcntl, terminal I/O, memory mapped files, and so forth.

These aspects of Python have made it an extremely useful tool for writing all sorts of system-oriented tools and utilities, the purpose for which it was originally created and the way in which it is still used by a lot of system programmers and sysadmins. However, Python's C-like behavior has not been without its fair share of problems. For example, the truncation of integer division is a widely acknowledged source of unintended mathematical errors and the use of byte-oriented file I/O is a frequent source of confusion when working with Unicode in Internet applications.

Python 3: Breaking Free of Its Past

One of the most noticeable changes in Python 3 is a major shift away from its original roots in C and UNIX programming. Although the interpreter is still written in C, Python 3 fixes a variety of subtle design problems associated with its original implementation. For example, in Python 3, integer division now yields a floating point number:

```
>>> 7/4
1.75
>>>
```

A number of fundamental statements in the language have been changed into library functions. For example, `print` and `exec` are now just ordinary function calls. Thus, the familiar "Hello World" program becomes this:

```
print("Hello World")
```

Python 3 fully embraces Unicode text, a change that affects almost every part of the language and library. For instance, all text strings are now Unicode, as is Python source code. When you open files in text mode, Unicode is always assumed—even if you don't specify anything in the way of a specific encoding (with UTF-8 being the usual default). I'll discuss the implications of this change a little later—it's not as seamless as one might imagine.

Borrowing from Java, Python 3 takes a completely different approach to file I/O. Although you still open files using the familiar open() function, the kind of "file" object that you get back is now part of a layered I/O stack. For example:

```
>>> f = open("foo")
>>> f
<io.TextIOWrapper object at 0x383950>
>>>
```

So, what is this TextIOWrapper? It's a class that wraps a file of type BufferedReader, which in turns wraps a file of type FileIO. Yes, Python 3 has a full assortment of various I/O classes for raw I/O, buffered I/O, and text decoding that get hooked together in various configurations. Although it's not a carbon copy of what you find in Java, it has a similar flavor.

Borrowing from the .NET framework, Python 3 adopts a completely different approach to string formatting based on composite format strings. For example, here is the preferred way to format a string in Python 3:

```
print("{0:10} {1:10d} {2:10.2f}".format(name,shares,price))
```

For now, the old printf-style string formatting is still supported, but its future is in some doubt.

Although there are a variety of other more minor changes, experienced Python programmers coming into Python 3 may find the transition to be rather jarring. Although the core language feels about the same, the programming environment is very different from what you have used in the past. Instead of being grounded in C, Python 3 adopts many of its ideas from more modern programming languages.

Python's Evolution Into a Framework Language

If you're a current user of Python, you might have read the last section and wondered what the Python developers must be thinking. Breaking all backwards compatibility just to turn print into a function and make all string and I/O handling into a big Unicode playground seems like a rather extreme step to take, especially given that Python already has a rather useful print statement and fully capable support for Unicode. As it turns out, these changes aren't the main story of what Python 3 is all about. Let's review a bit of history.

Since its early days, Python has increasingly been used as a language for creating complex application frameworks and programming environments. Early adopters noticed that almost every aspect of the interpreter was exposed and that the language could be molded into a programming environment that was custom-tailored for specific application domains. Example frameworks include Web programming, scientific computing, image processing, and animation. The only problem was that even though Python was "good enough" to be used in this way, many of the tricks employed by framework builders pushed the language in directions that were never anticipated in its original design. In fact, there were a lot of subtle quirks, limitations, and inconsistencies. Not only that, there were completely new features that framework builders wanted—often inspired by features of other programming languages. So, as Python has evolved, it has gradually acquired a new personality.

Almost all of this development has occurred in plain sight, with new features progressively added with each release of the interpreter. Although many users have chosen to ignore this work, it all takes center stage in Python 3. In fact, the most significant aspect of Python 3 is that it sheds a huge number of deprecated features and programming idioms in order to lay the groundwork for a whole new class of advanced programming techniques. Simply stated, there are things that can be done in Python 3 that are not at all possible in previous versions. In the next few sections, I go into more detail about this.

Python Metaprogramming

Python has always had two basic elements for organizing programs: functions and classes. A function is a sequence of statements that operate on some passed arguments and return a result. For example:

```
def factorial(n):
   result = 1
   while n > 1:
      result *= n
      n -= 1
   return result
```

A class is a collection of functions called methods that operate on objects known as instances. Here is a sample class definition:

```
class Rectangle(object):
   def __init__(self,width,height):
      self.width  = width
      self.height = height
   def area(self):
      return self.height*self.width
   def perimeter(self):
      return 2*self.height + 2*self.width
```

Most users of Python are familiar with the idea of *using* functions and classes to carry out various programming tasks. For example:

```
>>> print factorial(6)
720
>>> r = Rectangle(4,5)
>>> r.area()
20
>>> r.perimeter()
18
>>>
```

However, an often overlooked feature is that function and class definitions have first-class status. That is, when you define a function, you are creating a "function object" that can be passed around and manipulated just like a normal piece of data. Likewise, when you define a class, you are creating a "type object." The fact that functions and classes can be manipulated means that it is possible to write programs that carry out processing on their own internal structure. That is, you can write code that operates on function and class objects just as easily as you can write code that manipulates numbers or strings. Programming like this is known as *metaprogramming*.

METAPROGRAMMING WITH DECORATORS

A common metaprogramming example is the problem of creating various forms of function wrappers. This is typically done by writing a function that accepts another function as input and that dynamically creates a completely new function that wraps an extra layer of logic around it. For example, this function wraps another function with a debugging layer:

```
def debugged(func):
    def call(*args,**kwargs):
        print("Calling %s" % func.__name__)
        result = func(*args,**kwargs)
        print("%s returning %r" % (func.__name__, result))
        return result
    return call
```

To use this utility function, you typically apply it to an existing function and use the result as its replacement. An example will help illustrate:

```
>>> def add(x,y):
...     return x+y
...
>>> add(3,4)
7
>>> add = debugged(add)
>>> add(3,4)
Calling add
add returning 7
7
>>>
```

This wrapping process became so common in frameworks, that Python 2.4 introduced a new syntax for it known as a *decorator*. For example, if you want to define a function with an extra wrapper added to it, you can write this:

```
@debugged
def add(x,y):
    return x+y
```

The special syntax @*name* placed before a function or method definition specifies the name of a function that will process the function object created by the function definition that follows. The value returned by the decorator function takes the place of the original definition.

There are many possible uses of decorators, but one of their more interesting qualities is that they allow function definitions to be manipulated at the time they are defined. If you put extra logic into the wrapping process, you can have programs selectively turn features on or off, much in the way that a C programmer might use the preprocessor. For example, consider this slightly modified version of the `debugged()` function:

```
import os
def debugged(func):
    # If not in debugging mode, return func unmodified
    if os.environ.get('DEBUG','FALSE') != 'TRUE':
        return func

    # Put a debugging wrapper around func
    def call(*args,**kwargs):
        print("Calling %s" % func.__name__)
        result = func(*args,**kwargs)
        print("%s returning %r" % (func.__name__, result))
        return result
    return call
```

In this modified version, the `debugged()` function looks at the setting of an environment variable and uses that to determine whether or not to put a debugging wrapper around a function. If debugging is turned off, the function is simply left alone. In that case, the use of a decorator has no effect and the program runs at full speed with no extra overhead (except for the one-time call to debugged() when decorated functions are *defined*).

Python 3 takes the idea of function decoration to a whole new level of sophistication. Let's look at an example:

```
def positive(x):
   "must be positive"
   return x > 0

def negative(x):
   "must be negative"
   return x < 0

def foo(a:positive, b:negative) -> positive:
   return a - b
```

The first two functions, negative() and positive(), are just simple function definitions that check an input value x to see if it satisfies a condition and return a Boolean result. However, something very different must be going on in the definition of foo() that follows.

The syntax of foo() involves a new feature of Python 3 known as a function annotation. A function annotation is a mechanism for associating *arbitrary* values with the arguments and return of a function definition. If you look carefully at this code, you might get the impression that the positive and negative annotations to foo() are carrying out some kind of magic—maybe calling those functions to enforce some kind of contract or assertion. However, you are wrong. In fact, these annotations do absolutely nothing! foo() is just like any other Python function:

```
>>> foo(3,-2)
5
>>> foo(-5,2)
-7
>>>
```

Python 3 doesn't do anything with annotations other than store them in a dictionary. Here is how to view it:

```
>>> foo.__annotations__
{'a': <function positive at 0x384468>,
 'b': <function negative at 0x3844b0>,
 'return': <function positive at 0x384468> }
>>>
```

The interpretation and use of these annotations are left entirely unspecified. However, their real power comes into play when you mix them with decorators. For example, here is a decorator that looks at the annotations and creates a wrapper function where they turn into assertions:

```
def ensure(func):
   # Extract annotation data
   return_check = func.__annotations__.get('return',None)
   arg_checks   = [(name,func.__annotations__.get(name))
                   for name in func.__code__.co_varnames]

   # Create a wrapper that checks argument values and the return
   # result using the functions specified in annotations

   def assert_call(*args,**kwargs):
      for (name,check),value in zip(arg_checks,args):
            if check: assert check(value), "%s %s" % (name, check.__doc__)
        for name,check in arg_checks[len(args):]:
            if check: assert check(kwargs[name]), "%s %s" % (name, check.__
doc__)
        result = func(*args,**kwargs)
        assert return_check(result), "return %s" % return_check.__doc__
        return result

    return assert_call
```

This code will undoubtedly require some study, but here is how it is used in a program:

```
@ensure
def foo(a:positive, b:negative) -> positive:
    return a - b
```

Here is an example of what happens if you violate any of the conditions when calling the decorated function:

```
>>> foo(3,-2)
5
>>> foo(-5,2)
Traceback (most recent call last):
  File "", line 1, in
  File "meta.py", line 19, in call
    def assert_call(*args,**kwargs):
AssertionError: a must be positive
>>>
```

It's really important to stress that everything in this example is user-defined. Annotations can be used in any manner whatsoever—the behavior is left up to the application. In this example, we built our own support for a kind of "contract" programming where conditions can be optionally placed on function inputs. However, other possible applications might include type checking, performance optimization, data serialization, documentation, and more.

METACLASSES

The other major tool for metaprogramming is Python's support for metaclasses. When a class definition is encountered, the body of the class statement (all of the methods) populates a dictionary that later becomes part of the class object that is created. A metaclass allows programmers to insert their own custom processing into the last step of the low-level class creation process. The following example illustrates the mechanics of the "metaclass hook." You start by defining a so-called metaclass that inherits from `type` and implements a special method `__new__()`:

```
class mymeta(type):
    def __new__(cls,name,bases,dict):
        print("Creating class  :", name)
        print("Base classes   :", bases)
        print("Class body    :", dict)
        # Create the actual class object
        return type.__new__(cls,name,bases,dict)
```

Next, when defining new classes, you can add a special "metaclass" specifier like this:

```
class Rectangle(object,metaclass=mymeta):
    def __init__(self,width,height):
        self.width  = width
        self.height = height
    def area(self):
        return self.height*self.width
    def perimeter(self):
        return 2*self.height + 2*self.width
```

If you try this code, you will see the `_new__()` method of the metaclass execute once when the `Rectangle` class is *defined*. The arguments to this method contain all of the information about the class including the name, base classes, and dictionary of methods. I would strongly suggest trying this code to get a sense for what happens.

At this point, you might be asking yourself, "How would I use a feature like this?" The main power of a metaclass is that it can be used to manipulate the entire contents of class body in clever ways. For example, suppose that you collectively wanted to put a debugging wrapper around every single method of a class. Instead of manually decorating every single method, you could define a metaclass to do it like this:

```
class debugmeta(type):
    def __new__(cls,name,bases,dict):
        if os.environ.get('DEBUG','FALSE') == 'TRUE':
            # Find all callable class members and put a
            # debugging wrapper around them.
            for key,member in dict.items():
                if hasattr(member,'__call__'):
                    dict[key] = debugged(member)
        return type.__new__(cls,name,bases,dict)

class Rectangle(object,metaclass=debugmeta):
    ...
```

In this case, the metaclass iterates through the entire class dictionary and rewrites its contents (wrapping all of the function calls with an extra layer).

Metaclasses have actually been part of Python since version 2.2. However, Python 3 expands their capabilities in an entirely new direction. In past versions, a metaclass could only be used to process a class definition after the entire class body had been executed. In other words, the entire body of the class would first execute and then the metaclass processing code would run to look at the resulting dictionary. Python 3 adds the ability to carry out processing *before* any part of the class body is processed and to incrementally perform work as each method is defined. Here is an example that shows some new metaclass features of Python 3 by detecting duplicate method names in a class definition:

```
# A special dictionary that detects duplicates
class dupdict(dict):
    def __setitem__(self,name,value):
        if name in self:
            raise TypeError("%s already defined" % name)
        return dict.__setitem__(self,name,value)

# A metaclass that detects duplicates
class dupmeta(type):
    @classmethod
    def __prepare__(cls,name,bases):
        return dupdict()
```

In this example, the `__prepare__()` method of the metaclass is a special method that runs at the very beginning of the class definition. As input it receives the name of the class being defined and a tuple of base classes. It returns the dictionary object that will be used to store members of the class body. If you return a custom dictionary, you can capture each member of a class as it is defined. For example, the `dupdict` class redefines item assignment so that, if any duplicate is defined, an exception is immediately raised. To see this metaclass in action, try it with this code:

```
class Rectangle(metaclass=dupmeta):
   def __init__(self,width,height):
      self.width = self.width
      self.height = self.height
   def area(self):
      return self.width*self.height
   # This repeated method name will be rejected (a bug)
   def area(self):
      return 2*(self.width+self.height)
```

Finally, just to push all of these ideas a little bit further, Python 3 allows class definitions to be decorated. For example:

```
@foo
class Bar:
   statements
```

This syntax is shorthand for the following code:

```
class Bar:
   statements

Bar = foo(Bar)
```

So, just like functions, it is possible to use decorators to put wrappers around classes. Some possible use cases include applications related to distributed computing and components. For example, decorators could be used to create proxies, set up RPC servers, register classes with name mappers, and so forth.

HEAD EXPLOSION

If you read the last few sections and feel as though your brain is going to explode, then your understanding is probably correct. (Just for the record, metaclasses are also known as Python's "killer joke"—in reference to a Monty Python sketch that obviously can't be repeated here.) However, these new metaprogramming features are what really sets Python 3 apart from its predecessors, because these are the new parts of the language that can't be emulated in previous versions. They are also the parts of Python 3 that pave the way to entirely new types of framework development.

Python 3: The Good

On the whole, the best feature of Python 3 is that the entire language is more logically consistent, entirely customizable, and filled with advanced features that provide an almost mind-boggling amount of power to framework builders. There are fewer corner cases in the language design and a lot of warty features from past versions have been removed. For example, there are no "old-style" classes, string exceptions, or features that have plagued Python developers since its very beginning, but which could not be removed because of backwards compatibility concerns.

There are also a variety of new language features that are simply nice to use. For example, if you have used features such as list comprehensions, you know that they are a very powerful way to process data. Python 3 builds upon this and adds support for set and dictionary comprehensions. For example, here is an example of converting all keys of a dictionary to lowercase:

```
>>> data = { 'NAME' : 'Dave', 'EMAIL':'dave@dabeaz.com' }
>>> data = {k.lower():v for k,v in data.items}
>>> data
{'name': 'Dave', 'email': 'dave@dabeaz.com'}
>>>
```

Major parts of the standard library—especially those related to network programming—have been reorganized and cleaned up. For instance, instead of a half-dozen scattered modules related to the HTTP protocol, all of that functionality has been collected into an HTTP package.

The bottom line is that, as a programming language, Python 3 is very clean, very consistent, and very powerful.

Python 3: The Bad

The obvious downside to Python 3 is that it is not backwards compatible with prior versions. Even if you are aware of basic incompatibilities such as the `print` statement, this is only the tip of the iceberg. As a general rule, it is not possible to write any kind of significant program that simultaneously works in Python 2 and Python 3 without limiting your use of various language features and using a rather contorted programming style.

In addition to this, there are no magic directives, special library imports, environment variables, or command-line switches that will make Python 3 run older code or allow Python 2 to run Python 3 code. Thus, it's really important to stress that you must treat Python 3 as a completely different language, not as the next step in a gradual line of upgrades from previous versions.

The backwards incompatibility of Python 3 presents a major dilemma for users of third-party packages. Unless a third-party package has been explicitly ported to Python 3, it won't work. Moreover, many of the more significant packages have dependencies on other Python packages themselves. Ironically, it is the large frameworks (the kind of code for which Python 3 is best suited) that face the most daunting task of upgrading. As of this writing, some of the more popular frameworks aren't even compatible with Python 2.5—a release that has been out for several years. Needless to say, they're not going to work with Python 3.

Python 3 includes a tool called 2to3 that aims to assist in Python 2 to Python 3 code migration. However, it is not a silver bullet nor is it a tool that one should use lightly. In a nutshell, 2to3 will identify places in your program that might need to be fixed. For example, if you run it on a "hello world" program, you'll get this output:

```
bash-3.2$ 2to3 hello.py
RefactoringTool: Skipping implicit fixer: buffer
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: ws_comma
--- hello.py (original)
+++ hello.py (refactored)
@@ -1,1 +1,1 @@
-print "hello world"
```

```
+print("hello world")
RefactoringTool: Files that need to be modified:
RefactoringTool: hello.py
bash-3.2$
```

By default, 2to3 only identifies code that needs to be fixed. As an option, it can also rewrite your source code. However, that must be approached with some caution. 2to3 might try to fix things that don't actually need to be fixed, it might break code that used to work, and it might fix things in a way that you don't like. Before using 2to3, it is highly advisable to first create a thorough set of unit tests so that you can verify that your program still works after it has been patched.

Python 3: The Ugly

By far the ugliest part of Python 3 is its revised handling of Unicode and the new I/O stack. Let's talk about the I/O stack first.

In adopting a layered approach to I/O, the entire I/O system has been reimplemented from the ground up. Unfortunately, the resulting performance is so bad as to render Python 3 unusable for I/O-intensive applications. For example, consider this simple I/O loop that reads a text file line by line:

```
for line in open("somefile.txt"):
    pass
```

This is a common programming pattern that Python 3 executes more than 40 times slower than Python 2.6! You might think that this overhead is due to Unicode decoding, but you would be wrong—if you open the file in binary mode, the performance is even worse! Numerous other problems plague the I/O stack, including excessive buffer copying and other resource utilization problems. To say that the new I/O stack is "not ready for prime time" is an understatement.

To be fair, the major issue with the I/O stack is that it is still a prototype. Large parts of it are written in Python itself, so it's no surprise that it's slow. As of this writing, there is an effort to rewrite major parts of it in C , which can only help its performance. However, it is unlikely that this effort will ever match the performance of buffered I/O from the C standard library (the basis of I/O in previous Python versions). Of course, I would love to be proven wrong.

The other problematic feature of Python 3 is its revised handling of Unicode. I say this at some risk of committing blasphemy—the fact that Python 3 treats all text strings as Unicode is billed as one of its most important features. However, this change now means that Unicode pervades every part of the interpreter and its standard libraries. This includes such mundane things as command-line options, environment variables, filenames, and low-level system calls. It also means that the intrinsic complexity of Unicode handling is now forced on *all* users regardless of whether or not they actually need to use it.

To be sure, Unicode is a critically important aspect of modern applications. However, by implicitly treating all text as Unicode, Python 3 introduces a whole new class of unusual programming issues not seen in previous Python versions. Most of these problems stem from what remains a fairly loose notion of any sort of standardized Unicode encoding in many systems. Thus, there is now a potential mismatch between low-level system interfaces and the Python interpreter.

To give an example of the minefield awaiting Python 3 users, let's look at a simple example involving the file system on a Linux system. Take a look at this directory listing:

```
% ls -b
image.png jalape\361o.txt readme.txt
%
```

In this directory, there is a filename Jalepe\361o.txt with an extended Latin character, "ñ," embedded in it. Admittedly, that's not the most common kind of filename one encounters on a day-to-day basis, but Linux allowed such a filename to be created, so it must be assumed to be technically valid.

Past versions of Python have no trouble dealing with such files, but let's take a look at what happens in Python 3. First, you will notice that there is no apparent way to open the file:

```
>>> f = open("jalape\361o.txt")
Traceback (most recent call last):
IOError: [Errno 2] No such file or directory: 'jalapeño.txt'
>>>
```

Not only that, the file doesn't show up in directory listings or with file globbing operations—so now we have a file that's invisible! Let's hope that this program isn't doing anything critical such as making a backup.

```
>>> os.listdir(".")
['image.png', 'readme.txt']
>>> glob.glob("*.txt")
['readme.txt']
>>>
```

Let's try passing the filename into a Python 3.0 program as a command-line argument:

```
% python3.0 *.txt
Could not convert argument 2 to string
%
```

Here the interpreter won't run at all—end of story.

The source of these problems is the fact that the filename is not properly encoded as UTF-8 (the usual default assumed by Python). Since the name can't be decoded, the interpreter either silently rejects it or refuses to run at all. There are some settings that can be made to change the encoding rules for certain parts of the interpreter. For example, you can use `sys.setfilesystemencoding()` to change the default encoding used for names on the file system. However, this can only be used after a program starts and does not solve the problem of passing command-line options.

Python 3 doesn't abandon byte-strings entirely. Reading data with binary file modes [e.g., `open(filename,"rb")`] produces data as byte-strings. There is also a special syntax for writing out byte-string literals. For example:

```
s = b'Hello World'          # String of 8-bit characters
```

It's subtle, but supplying byte-strings is one workaround for dealing with funny file names in our example. For example:

```
>>> f = open(b'jalape\361o.txt')
>>> os.listdir(b'.')
[b'jalape\xf1o', b'image.png', b'readme.txt']
>>>
```

Unlike past versions of Python, byte-strings and Unicode strings are strictly separated from each other. Attempts to mix them in any way now produce an exception:

```
>>> s = b'Hello'
>>> t = "World"
>>> s+t Traceback (most recent call last):
  File "", line 1, in
TypeError: can't concat bytes to str
>>>
```

This behavior addresses a problematic aspect of Python 2 where Unicode strings and byte-strings could just be mixed together by implicitly promoting the byte-string to Unicode (something that sometimes led to all sorts of bizarre programming errors and I/O issues). It should be noted that the fact that string types can't be mixed is one of the most likely things to break programs migrated from Python 2. Sadly, it's also one feature that the 2to3 tool can't detect or correct. (Now would be a good time to start writing unit tests.)

System programmers might be inclined to use byte-strings in order to avoid any perceived overhead associated with Unicode text. However, if you try to do this, you will find that these new byte-strings do not work at all like byte-strings in prior Python versions. For example, the indexing operator now returns byte values as integers:

```
>>> s[2]
108
>>>
```

If you print a byte-string, the output always includes quotes and the leading b prefix—rendering the `print()` statement utterly useless for output except for debugging. For example:

```
>>> print(s)
b'Hello'
>>>
```

If you write a byte-string to any of the standard I/O streams, it fails:

```
>>> sys.stdout.write(s)
Traceback (most recent call last):
  File "", line 1, in
  File "/tmp/lib/python3.0/io.py", line 1484, in write
    s.__class__.__name__)
TypeError: can't write bytes to text stream
>>>
```

You also can't perform any kind of meaningful formatting with byte-strings:

```
>>> b'Your age is %d' % 42
Traceback (most recent call last):
  File "", line 1, in
TypeError: unsupported operand type(s) for %: 'bytes' and 'int'
>>>
```

Common sorts of string operations on bytes often produce very cryptic error messages:

```
>>> 'Hell' in s
Traceback (most recent call last):
  File "", line 1, in
TypeError: Type str doesn't support the buffer API
>>>
```

This does work if you remember that you're working with byte-strings:

```
>>> b'Hell' in s
True
>>>
```

The bottom line is that Unicode is something that you will be forced to embrace in Python 3 migration. Although Python 3 corrects a variety of problematic aspects of Unicode from past versions, it introduces an entirely new set of problems to worry about, especially if you are writing programs that need to work with byte-oriented data. It may just be the case that there is no good way to entirely handle the complexity of Unicode—you'll just have to choose a Python version based on the nature of the problems you're willing to live with.

Conclusions

At this point, Python 3 can really only be considered to be an initial prototype. It would be a mistake to start using it as a production-ready replacement for Python 2 or to install it as an "upgrade" to a Python 2 installation (especially since no Python 2 code is likely to work with it).

The embrace of Python 3 is also by no means a foregone conclusion in the Python community. To be sure, there are some very interesting features of Python 3, but Python 2 is already quite capable, providing every feature of Python 3 except for some of its advanced features related to metaprogramming. It is highly debatable whether programmers are going to upgrade based solely on features such as Unicode handling—something that Python already supports quite well despite some regrettable design warts.

The lack of third-party modules for Python 3 also presents a major challenge. Most users will probably view Python 3 as nothing more than a curiosity until the most popular modules and frameworks make the migration. Of course this raises the question of whether or not the developers of these frameworks see a Python 3 migration as a worthwhile effort.

It will be interesting to see what programmers do with some of the more advanced aspects of Python 3. Many of the metaprogramming features such as annotations present interesting new opportunities. However, I have to admit that I sometimes wonder whether these features have made Python 3 too clever for its own good. Only time will tell.

General Advice

For now, the best advice is to simply sit back and watch what happens with Python 3—at the very least it will be an interesting case study in software engineering. New versions of Python 2 continue to be released and there are no immediate plans to abandon support for that branch. Should you decide to install Python 3, it can be done side-by-side with an existing Python 2 installation. Unless you instruct the installation process explicitly, Python 3 will not be installed as the default.

REFERENCES

[1] What's New in Python 3?: http://docs.python.org/dev/3.0/whatsnew/3.0.html.

RUDI VAN DRUNEN

the basics of power

Rudi van Drunen is a senior UNIX systems consultant with Competa IT B.V. in the Netherlands. He also has his own consulting company, Xlexit Technology, doing low-level hardware-oriented jobs.

rudi-usenix@xlexit.com

WE OFTEN TALK ABOUT SYSTEMS FROM a "in front of the (working) screen" or a "software" perspective. Behind all this there is a complex hardware architecture that makes things work. This is your machine: the machine room, the network, and all. Everything has to do with electronics and electrical signals. In this article I will discuss the background of some of the electronics, introducing the basics of power and how to work with it, so that you will be able to understand the issues and calculations that are the basis of delivering the electrical power that makes your system work.

There are some basic things that drive the electrons through your machine. I will be explaining Ohm's law, the power law, and some aspects that will show you how to lay out your power grid.

Power Law

Any piece of equipment connected to a power source will cause a current to flow. The current will then have the device perform its actions (and produce heat). To calculate the current that will be flowing through the machine (or light bulb) we divide the power rating (in watts) by the voltage (in volts) to which the system is connected. An example here is if you take a 100-watt light bulb and connect this light bulb to the wall power voltage of 115 volts, the resulting current will be 100/115 = 0.87 amperes.

This equation can be written as follows:

$$I = P/U$$

or, after performing some algebra,

$$U = P/I \text{ or } P = UI$$

where

P is the power (in units of watts [W])

U is the voltage (in units of volts [V])

I is the current (in units of amperes [A])

Note the use of U for the voltage here; this is commonly used to distinguish between the voltage at a certain point and the unit of voltage (volts, V). In the literature the symbol V is also used for both the voltage and the unit volts, which can be confusing.

Ohm’s Law

To have current flow, the device you are connecting to a voltage source has to pose some resistance to the electrons that want to flow from one terminal to the other terminal. The more resistance the device has, the less current will be flowing through the device.

We calculate resistance using Ohm’s law, which can be written as

$$R = U/I$$

or

$$U = IR$$

or

$$I = U/R$$

where

R is the resistance (in units of ohms [Greek capital omega, Ω])

U is the voltage (in units of volts [V])

I is the current (in units of amperes [A])

We can calculate the resistance of the light bulb just discussed by dividing the voltage (115 V) by the current flowing (0.87 A). This results in a resistance of 132 Ω.

Note that for a light bulb the calculated resistance is the resistance in the “on” or hot state. The “off” resistance can be very different.

Combining Ohm’s law and the power law, we can calculate the power that a resistor as a load to a voltage source will convert into heat (or a motor will convert into both heat and mechanical power):

$$P = U(U/R) = U^2/R$$

which can be rewritten using Ohm’s law to

$$P = I(RI) = I^2R$$

This might also be applicable to a power cable. We all know that if you use a too thin (too low a rating) power cable for heavy equipment, the cable will get hot and eventually catch fire on the points where the resistance is the highest (most likely at the points where the plug connects to the wire). If we take a standard power cable (12 gauge), it will have a resistance of 2.0 Ω per 1000 ft. If you connect a 800-W piece of equipment to it (a strand of 500 ft of extension wire, totaling 1000 ft of conductor), this cable will dissipate more than 300 W of heat. (The resistance of wire can be found at http://www.powerstream.com/Wire_Size.htm and seen in Table 1.)

Rating	Type	ohms/1000 ft	ft/ohm
12 AWG	stranded	1.65	606
14 AWG	stranded	2.62	381
16 AWG	stranded	4.17	239
12 AWG	Solid	1.59	629
14 AWG	Solid	2.52	396
16 AWG	Solid	4.02	249

TABLE 1: RESISTANCE RATINGS FOR COMMON GAUGES OF WIRE

We first calculate the resistance of the device: 800 W/115 V = 6.9 Ω. We then add the power cable resistance to it, for a total of 8.9 Ω. Now we calculate the current flowing through the complete system: 115 V/8.9 Ω = 12.9 A. So both through the device and the cord, we see a current of 12.9 A. With these values we can both calculate the power dissipated in the power cord and the voltage that drops over the power cord. A schematic drawing is shown in Figure 1. The power equals $(12.9)^2 \times 2.0 = 333$ W and the voltage drop is $12.9 \times 2.0 = 25.8$ V. Now as we connect to the mains of 115 V and we have a total voltage drop of 25.8 V (12.9 V per conductor) over the power cable, so only $115 - 25.8 = 89.2$ V remains for the device.

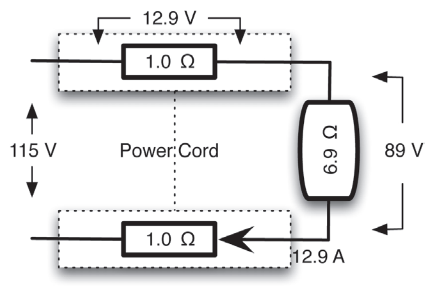


FIGURE 1: THE VOLTAGE DROP WHEN USING 500 FT OF 12 GAUGE EXTENSION CORD WITH 115 V WILL BE 25.8 V, LEAVING ONLY 89 V FOR THE DEVICE.

This calculation shows that it is important to have the correct rating of the power cable, to ensure that as much power as possible flows to the device where it does the work and to minimize the losses in the power cabling (which is just wasted heat).

Signals

Previously we assumed that the voltage applied to the resistor (or light bulb) was constant. However, the mains voltage in the United States varies between 162.15 V and –162.15 V. Figure 2 shows the sine wave for this alternating current (AC) voltage. A number of parameters can be derived. We have the top (maximum) value of the voltage (162.15 V), the average (over a integer number of periods) voltage (0 V), and the root mean square (RMS) voltage (which is the rated voltage of the mains, i.e., the effective voltage). For a sine wave the maximum voltage is √2 times the RMS voltage. If the curve is not a sine wave the factor is different. The ratio between the peak (max) value of a signal and the RMS value of this signal is called the crest factor. A direct current (DC) voltage has a crest factor of 1, and a pure sine wave 1.41 (√2).

To do power calculations as described previously, we use the RMS value.

We can determine the period and frequency. The frequency is defined as 1/period. With a 16-ms period, the frequency equals 60 hertz [Hz].

A signal can have both AC and DC components. The mains voltage has no DC component, but, as we see later, we can have an AC voltage biased by a DC voltage, as shown in Figure 3. The average of this signal (over an integer number of periods) is not 0, of course, and the RMS value equals the original RMS value (without bias) plus the bias voltage.

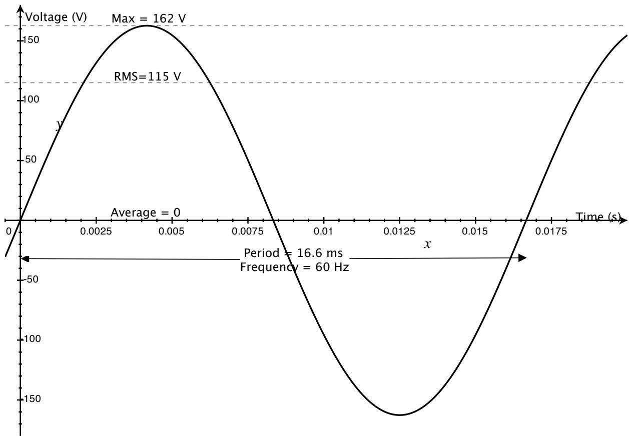


FIGURE 2: AC VOLTAGE SHOWING MAXIMUM AND MINIMUM VALUES ALONG WITH THE RMS VALUE, PERIOD, AND FREQUENCY

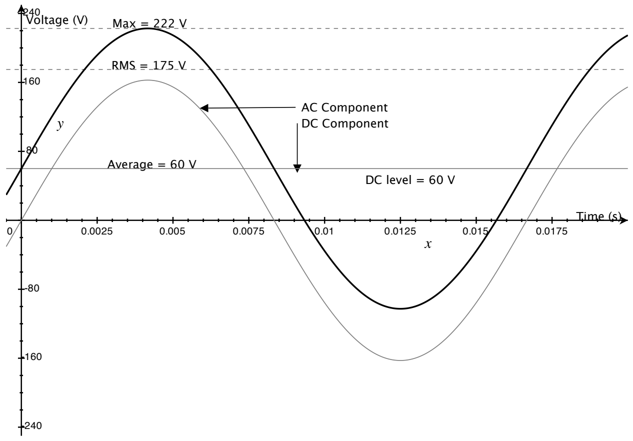


FIGURE 3: AN AC VOLTAGE WITH A DC COMPONENT

Components

In electronics we distinguish passive components (resistors, capacitors, and inductors) from active components (diodes, transistors, or more complex semiconductors). The active components can take actions, such as preventing current from flowing in one direction, or switching current on or off, whereas passive components cannot. In this article we introduce the three basic passive components. Figure 4 shows the corresponding schematic symbols.

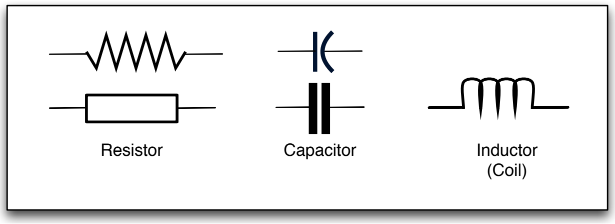


FIGURE 4: THE THREE BASIC PASSIVE COMPONENTS

RESISTOR

Earlier, I silently introduced the resistor as a component. A resistor is often a small component found in almost all electronic circuits, which reduces the flow of electrons. You can compare it to a obstruction in a garden hose. The water (electrical current) cannot flow freely, and a pressure difference (voltage) builds up over this obstruction. Also the power cord acts as an obstruction. The unit of resistance is the ohm (denoted by the Greek letter omega, Ω).

CAPACITOR

A capacitor can hold electrical charge. Capacitors are generally very small when used in a circuit or take on (very) large forms in power systems as described later. If a current flows to a capacitor, it fills it with electrical charge over time and a voltage builds up over it. Compare this with a bucket that is being filled by the garden hose. The unit of capacitance is the farad (denoted by F).

INDUCTOR

The inductor is a basic coil. It acts about the same as a capacitor, but instead of an electrical field it builds a magnetic field inside the core, as a result of an applied voltage. An applied voltage causes a current to flow and this is used to build up the magnetic field; after the field is in place, current will flow through the inductor. Inductors are often used together with capacitors to filter signals or generate oscillating signals. We use the unit henry (H) for the inductance.

Complex Power

All of what I have discussed so far in power calculations is true if we have a perfect resistor as a load (e.g., a heating element or a lamp), but the calculations get more complicated when we have more complex loads. A piece of electronics has not only resistors but also capacitors and inductors. We can describe the behavior of a capacitor and an inductor by the following expressions: For an inductor,

$$U = -L(dI/dt)$$

and for a capacitor,

$$U = C \int I dt$$

where

L is the value of the inductor (in henries)

C is the value of the capacitor (in farads)

So, if we connect a circuit containing reactive components (capacitors and inductors) to an AC voltage source, some time will be needed to build up energy in capacitors, for example, so the result will be a time difference (phase difference) between the applied voltage waveform and the resulting current waveform. The stored energy [in electric fields (capacitor) or magnetic fields (inductor)] cannot result in “work” and flows back into the applied source. The factor between the current able to do actual work and the current that is used to build up charge (blind current) and not to perform work is called the power factor (*Pf*).

Real (pure) resistive loads (heaters, lamps, etc.) have a power factor of 1, and the voltage and current are in phase. But if we observe motors, for example, [containing coils (inductors)] we see that the power factor is below 1. In general:

Pf = *P*/*S*

where

P is the real power (in watts)
S is the apparent power (in volt-amperes)

We define the power factor as the cosine of the angle (phi) between the voltage and current [*Pf* = cos(phi)]. I need to explain the angle here. I define a full wave as 360 degrees, or 2π radians; hence we can relate a time period to an angle. Figure 5 shows the waveforms of the voltage, current, and power and their relationship over time: if phi = [1.8 ms (phi)]/[16.6 ms (full wave)] × 360 degrees = 39 degrees, the power factor *Pf* = cos(39 degrees) = 0.77.

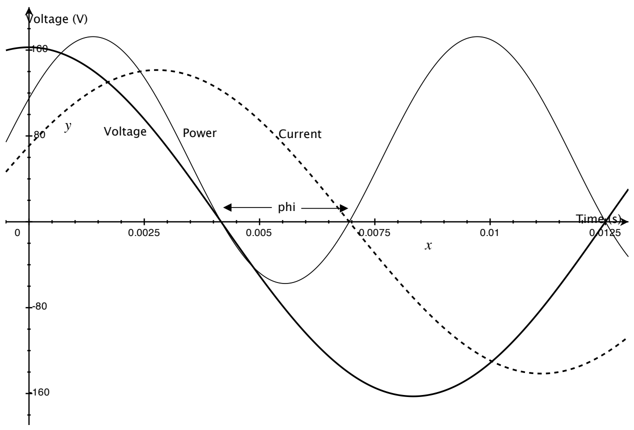


FIGURE 5: RELATIONSHIP AMONG VOLTAGE, CURRENT, AND POWER OVER TIME

If you have a load of 10 kW and the power factor is 1.0, your power supply has to supply 10 kW of power to the system. However, if the power factor is 0.6, the supply system in place needs to be set up for transport of 10/0.6 = 16.6 kW. The power company wants you to have your power factor as close to 1.0 as possible, so (as they bill you only for the power that actually does do work) they can design their systems and bill you for the power properly. Mostly in your contract with a power company it states that you (your datacenter) should have a power factor of 0.8 or better. The UPS that you probably have installed in your datacenter is specified in volt-amperes (VA), which is the total (apparent) power you need.

To get your power factor in shape (which generally means *Pf* > 0.8) you might have devices that contain capacitors and/or inductors installed to cancel out (part of) the effects of the loads you have. Mostly they consist of some measuring circuitry to measure the phase angle between the voltage and resulting current and a system to add capacity or induction to the system.

The power factor of most modern computer equipment (with switched mode power supplies) is close to 1, but other equipment used in the datacenter (for, say, cooling or ventilation) involving large motors can change the power factor dramatically.

Multiphase Systems

Generally, the power company delivers your power in three phases. The wire that enters your building (not residential) contains three or four conductors, each (of the three) carrying a single phase of 120 V power. A common return line can be provided. The 120 V is measured against this common return line. The sinusoidal voltage on each of the three phases is shifted 120 degrees.

The voltage between the phases equals √3 × 120 = 208 volts.

Using three phases is a more efficient way of transporting and using electrical power. A motor can more easily be driven by a magnetic field that rotates by itself. This can be accomplished by using three or more coils in the static part of the motor that are connected to the different phases. As the different coils are driven 120 degrees apart electrically, the magnetic field will rotate accordingly and drive the motor. Large motors (larger than 5 kW) are only available in three-phase versions. Using three-phase systems for large loads can be up to 75% more efficient than single-phase systems.

You can use each individual phase to act as a single-phase supply and connect part of your racks to that by using a step-down transformer (converting 208 to 120 V, as often the "common return" line is not available). The power company will request that you distribute your load evenly over the different phases. Some three-phase UPS devices will do that for you.

In a residential environment a number of buildings are connected to one phase, using a converter transformer often found on the pole, yielding an evenly distributed load.

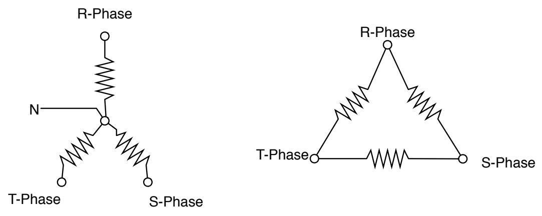


FIGURE 6: DIFFERENT WAYS OF USING THREE-PHASE POWER IN MOTORS

There are two different wiring schemes for three-phase systems, called star and delta (triangle) configurations (see Fig. 6). These configurations are the layout of how the coils in a motor are connected to the phases of a three-phase system. The star configuration is used to start the motor and deliver high torque; after that some smart electronics switch over to delta mode when running. As the different phases (coils in the static part of the motor) are driven with a time lag (see Fig. 7) the magnetic field inside the motor rotates, causing the rotor to rotate. You will see this in large UPS systems, flywheels, and air-conditioning systems comprising large motors. Most "normal" computing systems will connect to one phase. However, large machines and disk cabinets will eventually use three-phase input and often use the three phases to feed three power supplies.

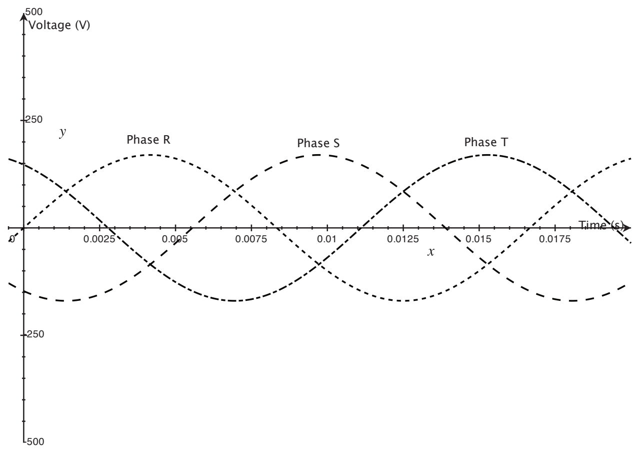


FIGURE 7: THREE-PHASE POWER VOLTAGE GRAPH

Grounding

Each system (computer, rack, etc.) should have an adequate connection to the ground wire. Exceptions to this rule are some double-insulated consumer devices, which we will discuss later. The ground wire (often a green/yellow striped wire) is connected to an electrode that is physically driven into the earth. The ground connection prevents the physically exposed conducting parts of a device from getting exposed to dangerous potentials in case of a failure of the insulation. A special kind of circuit breaker will signal such a fault (by comparing the current in the power line with that in the return line and noting any difference) and switch off the power. This is commonly called a Ground Fault Interrupter (GFI).

Also, the connection to ground prevents the buildup of static electricity (e.g., from air friction from the fans inside a cabinet). I will discuss this in a later article in which we evaluate static electricity and the issues concerned with it.

A common issue is the use of so-called transzorbs (overvoltage protecting devices) in power supplies. These devices make a short circuit from the mains line to the grounding pin when a power surge occurs, thereby protecting the device from getting fried. These devices can go bad (which often happens when they have already done their job once) and leak just a little of the mains voltage to the ground and the metal chassis. They then will act as a capacitor and have a noninfinite resistance to AC voltage.

If this is the case (and you might have an improper connection to earth) your metal case will carry (part of) the mains voltage. This is particularly dangerous if you touch both the case and, for example, a central heating pipe (which is supposed to be grounded), or if you connect this system to another system that is properly grounded.

So, in short, make sure all equipment is securely grounded, including the racks. Be very careful to have the ground connection itself done properly.

Ground also refers to a reference level against which all voltages are measured. In a computer system we have the ground level, which represents the 0-V rail on the power supply, and the mains ground, which is connected to the chassis. They may or may not be tied together. (If not, we refer to this as a system with a “floating ground.”) Sometimes there is an AC coupling (capacitor) between the circuit ground and mains ground to get rid of high-frequency noise on the power line.

Adequately insulated equipment, such as some consumer equipment, is allowed by the regulatory body not to have ground connections. Often this equipment does not have a metal case or metal knobs.

GROUND LOOPS

Ground loops occur when the ground (0-V rail) potential of one device is not the same as the ground potential of the connecting device. Because of the resistance in one of the ground leads and the current flowing through it, the 0-V rail in device A will not have the same potential as the 0-V rail of device B. Figure 8 (next page) shows the voltages in time at two points in the circuit. Device A “sees” a different voltage compared to device B.

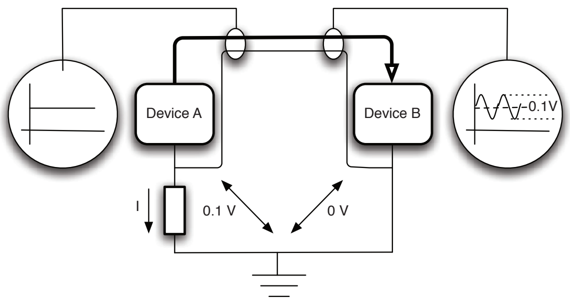


FIGURE 8: PRINCIPLE OF A GROUND LOOP

Figure 9 shows the circuit diagram, where the output voltage is not the same as U2 (as you would expect) but is the addition of U2 and the voltage generated through the ground current Uground:

Uout = U2 – Uground = U2 – U1[Rground/(Rground + R1)]

= U2 – (I1 × Rground)

Often, as the bias voltage is a result of the mains (ground) connection current, the bias voltage will be a sinusoidal waveform with the mains frequency. This is the "hum" you can encounter when connecting two mains-powered devices to each other (such as a computer and an audio amplifier). In a video system we can see this as bars scrolling vertically over the screen.

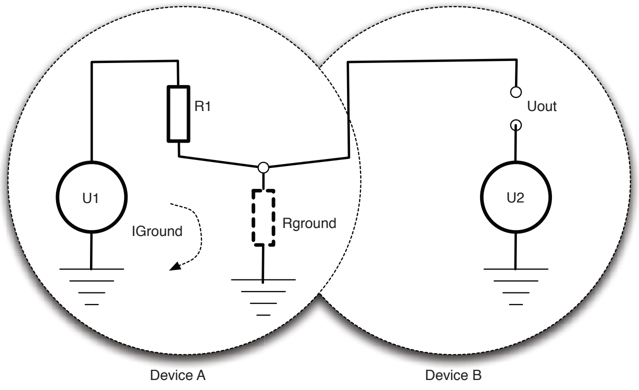


FIGURE 9: SCHEMATIC DIAGRAM OF A GROUND LOOP

To prevent ground loops, there should be one point at which a complete system is connected to ground, so we end up with a star topology. In practice that means that you should plug all parts of a system into the same outlet (keeping in mind that you do not want to overload the outlet or power cord).

Another source of hum, to be explained in a later article, is that from the bias signal as a result of the magnetic and electric field around a conductor that carries a current. When you run a power cable along an audio cable, the electromagnetic radiation coming from the current flowing through the power cable can induce a voltage in the low-signal (audio) cable. Therefore audio and other small signal-carrying cables are often shielded [i.e., have a metal shield that is connected to (signal) ground surrounding the conductors].

Conclusion

In this article I have described the way to calculate power requirements, how to lay out power cabling, and the various other issues that become important when scaling up from a couple of boxes to a datacenter. Using this knowledge, you can understand why it is important not only to build a nice software design or a systems design but also to take into account the way you connect these together and have the electrons flow through them. This is the first of a series of articles in which I will give some background on systems hardware, with the purpose of bringing you some more insight into what is happening behind the faceplate of your system, with the ultimate goal of helping you troubleshoot some of the hardware problems that you may encounter.

DAVID N. BLANK-EDELMAN

practical Perl tools: polymorphously versioned

David N. Blank-Edelman is the Director of Technology at the Northeastern University College of Computer and Information Science and the author of the O'Reilly book *Perl for System Administration* (with a second edition due out very soon). He has spent the past 24+ years as a system/network administrator in large multi-platform environments, including Brandeis University, Cambridge Technology Group, and the MIT Media Laboratory. He was the program chair of the LISA '05 conference and one of the LISA '06 Invited Talks co-chairs.

dnb@ccs.neu.edu

TODAY WE'RE GOING TO TALK ABOUT how Perl can improve life in the land of the version control system (VCS). I think you would be hard pressed to find anyone doing a serious amount of programming these days who doesn't use a VCS of some sort. But for those of you who are new to the biz, let me spend a paragraph or two bringing you up to speed.

The basic idea behind any VCS is that there is value in tracking all copies of the data, even and especially the intermediate copies, that make up some project. By tracking I mean "recording who worked on what piece of data, when, and what precisely they did." If this can be done well, it does wonders toward coordinating work being done by a number of people working on the same project so that the end result is congruent.

On top of this, two immensely helpful side effects emerge:

1. You can determine who did what to something and when (especially crucial for debugging).
2. You can revert to a previous working version when a new version causes something to break or recover when something was deleted in error.

The obvious thing for you to do is to keep all of the Perl code you write under some sort of version control system, but that's not what we're going to talk about in this column. If that was the only message of this column, I could simply say "do it" and then let you get back to flossing the cat or whatever you had planned to do today. Instead, we're going to look at how to use Perl to automate and augment several VCS packages. The particular VCS packages we're going to use in this column are those I'm most familiar with: RCS, Subversion, and Git. I'm going to assume you already have a little familiarity with them when I write about them in this column.

Note: This selection is not meant as a slight to any of the other interesting VCS packages, of which there are many (see http://en.wikipedia.org/wiki/Comparison_of_revision_control_software for a comparison). Rabid users of Mercurial, Bazaar, DARCS, Monotone, etc., are more than welcome to write me to let me know how much I'm suffering on a daily basis by not using their favorite system. (I'm always looking for new cool tools.) I'd also be remiss if I didn't mention the one Perl-based VCS, SVK (http://svk.bestpractical.com/view/HomePage). I have not used it, but it looks as though it has some very impressive distributed VCS features.

Automating the Revision Control System (RCS)

If you have only used the newer VCS packages, the notion of using RCS might seem a bit anachronistic. This may be like using a cotton gin for system administration but I assure you that RCS still has its place. Unlike the other systems we're going to look at, it has a few properties that come in handy in the right situations:

1. RCS is lightweight and fairly portable (there are RCS ports for most operating systems).
2. RCS is *file*-based (unlike SVN, which is directory-based, and git, which is content-based). This works in your favor if you just need to keep a file or two under version control.
3. RCS largely assumes strict locking will be used. Sometimes it makes no sense to have two people be able to work on the same file at once and have their changes merged in the end. System configuration files are one such example where there's good reason to serialize access.
4. Files under RCS almost always live right in the same place they are archived versus some nebulous and nonspecific "working directory." (/etc/services has to be in /etc; it does you no good if it lives just in a working directory somewhere else on the system.)

RCS is a good place to start our Perl discussion because it offers a simple example of the basic modus operandi we're going to see for each of these systems. To work with RCS from Perl, you use a Perl module called (surprise) Rcs.

First we load the module and let it know where it should expect to find the RCS binaries installed on your system. Most of the VCS modules are actually wrappers around your existing VCS binaries. Although that may be a little wasteful in terms of resources (e.g., Perl has to spin up some other program), this is more than made up for by the portability it provides. The module author does not have to maintain glue code to some C library that could change each time a new version is released or distribute libraries with the module that will also stale quickly.

Here's the start of all of our RCS code:

```
use Rcs;

Rcs->bindir('/usr/bin');
```

Once we've got that set up, we ask for a new Rcs object through which we'll perform all of our Rcs operations. To do anything we first have to tell the object just what file we're going to manipulate:

```
my $rcs = Rcs->new;
$rcs->file('/etc/services');
```

At this point we can start issuing Rcs commands using object methods named after the commands. Let's say we wanted to check a file out for use, modify it, and then check it back in again. That code would look like this:

```
$rcs->co('-l'); # check it out locked

# do something to the file
...

$rcs->ci('-u',  # check it back in, but leave unlocked version in situ
  '-m'
     . 'Modified by '
     . ( getpwuid($<) )[6] .   ' ('
     . ( getpwuid($<) )[0] .   ') on '
     . scalar localtime );
```

The last line of this code is in some ways the most interesting. For version control to be really useful, it is important to provide some sort of information each time a change is written back to its archive. At the bare minimum, I recommend making sure you log who made the change and when. Here's what the log messages might look like if you used this code:

```
revision 1.5
date: 2009/05/19 23:34:16;  author: dnb;  state: Exp;  lines: +1 -1
Modified by David N. Blank-Edelman (dnb) on Tue May 19 19:34:16 2009
----------------------------
revision 1.4
date: 2009/05/19 23:34:05;  author: eviltwin;  state: Exp;  lines: +1 -1
Modified by Divad Knalb-Namlede (eviltwin) on Tue May 19 19:34:05 2009
----------------------------
revision 1.3
date: 2009/05/19 23:33:35;  author: dnb;  state: Exp;  lines: +20 -0
Modified by David N. Blank-Edelman (dnb) on Tue May 19 19:33:16 2009
```

Eagle-eyed readers might note that the VCS itself should be recording the user and date automatically. That's true, except (a) sometimes your code runs as another user (e.g., root) and you want the uid and not the effective uid logged, and (b) the VCS records the time it hit the archive, but more interesting is probably the time the change was made. If your code takes a while before it gets to the part where it performs the VCS operation, the time information you care about might not get recorded.

This is the very basics for RCS use. The Rcs module also has methods such as revisions()/dates() to provide the list of revisions of a file and rcsdiff() to return the difference between two revisions. With methods like this you could imagine writing code that would analyze an RCS archive and provide information about how a file has changed over time. If you ever wanted to be able to search for a string found any time in a file's history, it would be fairly easy to write code to do that, thanks to this module.

Automating Subversion (SVN)

There are a few modules that allow you to operate on Subversion repositories in a similar fashion to the one we just saw (i.e., using an external program to automate operations versus calling the SVN libraries directly). The two I'd recommend you consider using are SVN::Agent and SVN::Class. I'm going to show you one example from each because they both have their strengths. SVN::Agent is the simpler of the two:

```
use SVN::Agent;

# SVN::Agent looks for the svn binaries in your path
$ENV{PATH} = '/path/to/svnbins' . ':' . $ENV{PATH};

# this assumes we've already got a working dir with files in it,
# if not, we could use the checkout() method
my $svn = SVN::Agent->load({ path => '/path/to/working_dir' });

$svn->update;  # update working dir with latest info in repos

print "These are the files that are modified:\n";
print join("\n",@{$svn->modified});

$svn->add('services')';  # add the file services to the changes list

$svn->prepare_changes;

$svn->commit('Files checked in by'
        . ( getpwuid($<) )[6] .   ' ('
        . ( getpwuid($<) )[0] .   ') on '
        . scalar localtime );
```

Most of that code should be fairly straightforward. The one line that is less than obvious is the call to prepare_changes. SVN::Agent keeps separate lists of the modified, added, deleted, etc., files in the working directory. When we said `$svn->add('services')` we added the services file to the added list. To give you the flexibility to choose which items should be committed to the repository, SVN::Agent keeps a separate changes list of files and dirs to be committed. This list starts out empty. The prepare_changes method copies the other lists (added, modified, etc.) in the change list so that the commit() method can do its stuff.

The second SVN module, SVN::Class, is interesting because it is essentially an extension of the excellent Path::Class module. Path::Class is a worthy replacement for the venerable File::Spec module. I'm sure we'll see it again in later columns, but, briefly, it provides an OS-independent, object-oriented way to work with file/directory names and filesystem paths. SVN::Class extends it by adding on the same sort of methods you'd expect in an SVN module [e.g., add(), commit(), delete()]. If you are using Path::Class in your programming, this will lend a consistent feel to your programs. Here's a very simple example:

```
use SVN::Class;

my $svnfile = svn_file('services');

$svnfile->svn('/usr/bin/svn'); # explicitly set location of svn command

$svnfile->add;

# ...  perform some operation on that file, perhaps using the Path::Class
#       open() method

my $revision = $svnfile->commit('File checked in by'
        . ( getpwuid($<) )[6] .   ' ('
        . ( getpwuid($<) )[0] .   ') on '
        . scalar localtime );

die "Unable to commit file 'services':" . $svnfile->errstr
   unless $revision;
```

SVN::Path does not have the same sort of interface to collectively commit() items as SVN::Agent, which may or may not be a plus in your eyes. It could be argued that an interface that forces you to actively call a commit() object for every file or directory makes for clearer code (versus using some backend data structure). However, SVN::Class does have some methods for querying the objects it uses (e.g., repository information, author of a file). I'd recommend picking the module that suits your style and the particular task.

Automating Git

For the last peek at automating a VCS from Perl we're going to look at Git, the wunderkind that has been storming the open source world. In fact, Perl development itself is now conducted using Git (and for a fun geek story, read about the transition at http://use.perl.org/articles/08/12/22/0830205.shtml).

Driving Git from Perl via Git::Wrapper is as simple as using it from the command line. You start in a similar fashion as the other wrappers we've seen:

```
use Git::Wrapper;

my $git = Git::Wrapper->new('/path/to/your/repos');
```

From this point on the $git object offers methods with the exact same name of each of the standard Git commands. If you look at the code of the module itself, you'll see that it has virtually no internal knowledge of how Git works. This means you have to understand Git's commands and semantics really well, because you'll get virtually no help from the module. That's a plus if you think Git does things perfectly and the module should get out of the way, but a minus if you were hoping for some (syntactic) sugar-coated methods to make your life easier.

Probably the best way to learn this module is to first get a good handle on Git itself from either the official doc [1] or a good book such as *Pragmatic Version Control Using Git* [2].

Automate All (Many) of Them

If you spotted a certain commonality among these modules (or perhaps repetition in my description of them), you are not alone. Max Kanat-Alexander decided to see if he could take a lesson from DBI and create VCI, the generic Version Control Interface. Just like DBI, where Tim Bunce created one interface for performing the operations generic to any number of databases, VCI tries to provide a similar framework for the various VCI packages. The distribution ships with submodules to provide support for Bazaar, Cvs, Git, Mercurial, and Subversion. The documentation is full of warnings about the alpha nature of the effort, but it is worth your consideration, especially if you have to switch between VCS packages on a regular basis.

VCS Augmentation

To end this column I want to briefly mention that Perl can be useful not only for automating VCS packages but also for augmenting them. Leaving aside SVK, the most extreme example (it builds upon parts of Subversion to make it into a whole new beast), there are a number of excellent modules and scripts that make working with these packages easier. Here are just a few to get you started:

- SVN::Access makes maintaining the Subversion repository access control file easy. This is handy if you programmatically provision SVN repositories.
- SVN::Mirror can help keep a local repository in sync with a remote one.
- App::SVN::Bisect provides a command-line tool to make bisecting a repository (searching for a particular change by splitting the commits in half, and then in half again, and then in half again) easy.
- App::Rgit executes a command on every Git repository found in a directory tree.
- Github::Import allows for easy importing of a project into the Git community repository hub github.com.

If you haven't thought of using Perl with your favorite VCS package before, hopefully this column has given you some ideas on how to head in that direction. Take care, and I'll see you next time.

REFERENCES

[1] http://git-scm.com/.

[2] T. Swicegood, *Pragmatic Version Control Using Git* (Pragmatic Bookshelf, 2008).

PETER BAER GALVIN

Pete's all things Sun: the Sun virtualization guide

Peter Baer Galvin is the Chief Technologist for Corporate Technologies, a premier systems integrator and VAR (www.cptech.com). Before that, Peter was the systems manager for Brown University's Computer Science Department. He has written articles and columns for many publications and is coauthor of the *Operating Systems Concepts* and *Applied Operating Systems Concepts* textbooks. As a consultant and trainer, Peter teaches tutorials and gives talks on security and system administration worldwide. Peter blogs at http://www.galvin.info and twitters as "PeterGalvin."

pbg@cptech.com

BACK IN THE DAY (SAY, FOUR YEARS ago) there was only one choice available for a system administrator wanting to run multiple environments within a single Sun server—domains. Domains provide a valuable solution to some problems, but they leave many other needs unaddressed. Fast-forward to today, and there are literally five main virtualization options (and dozens more if you count all of the x86 virtual machine technologies). These technologies vary in features, functionality, maturity, performance overhead, and supported hardware. This month in PATS, I'll provide a brief overview of the main choices and a chart that can be used in deciding which virtualization solution to bring to bear given a set of criteria.

Virtualization Options

The reason for the great increase in Sun virtualization options—which for the purposes of this column includes technologies both from Sun and from other providers—are multifold. Sun now has SPARC and x86 systems, for example. Each has its own virtualization choices. And as technology advances and servers increase in capacity, there is a natural need for implementation of features that help administrators best utilize those systems. Sometimes maximum utilization can be had by running one big application, but more and more frequently many, many applications can run comfortably within a system's resources. This situation will increasingly occur over time as more and more cores and threads fit into smaller and smaller systems.

Sometimes resource management can allow those multiple applications to play well together within a system, but in other situations more segregation between applications is needed. With that segregation, be it virtual machines or similar functionality such as zones, comes a new set of abilities for system administrators. Consider that VMware ESX software allows the movement of processes between systems without interrupting the processes or their users. The use of these technologies can totally revolutionize how computing facilities are designed, implemented, and managed. Disaster recovery (DR), resource use, load balancing, availability, and flexibility can all benefit from virtualization technologies. They are clearly worth using, but which should be used, and when?

Overview

A previous article in *;login:* provided details on many of the technologies [1]. To complete the picture, here is a brief overview of all of the options.

Dynamic System Domains [2] (or just "Domains") have existed for years in Sun Enterprise servers. They started as a method by which a larger server (such as an E10K) could be sliced into multiple, electronically isolated virtual systems. Each domain has its own operating system installation, and even a hardware fault in one domain typically does not affect any other domains. More recent Enterprise servers increase the ease with which domains can be dynamically reconfigured, in that CPU, memory, and I/O resources can be moved between them as needed. Typically this is done infrequently, to handle a temporary load or to change the system from interactive-oriented to batch-oriented (for example, between a daytime shift and an evening set of jobs).

Logical Domains [3] (LDoms) are the baby brother of Domains. They segregate a "coolthreads" (a.k.a. CMT or Niagara) system into virtual systems. Because these systems are motherboard-based, there is less fault isolation than on Enterprise servers. Some resources may be moved between LDoms, and "warm" migration is supported between systems. In this scenario, the two systems coordinate to move an LDom from the source to the destination, and the LDom is automatically suspended on the source system and resumed on the destination.

Zones/Containers [4] are a secure application environment, rather than a virtual machine implementation. There is one Solaris kernel running, and within that many, many zones can contain applications that run independently of other zones and can be resource-managed to a very fine degree. Fair-share scheduling can assure that each zone receives a "fair" amount of CPU, for example, while memory and CPU caps can limit the exact amount of each of those resources a zone can use.

xVM Server [5] is based on the Xen virtualization software project. It is a separate, open source product from Sun, in which Solaris is used as the software that creates and manages virtual machines. This is similar to the way ESX is the manager of virtual machines in a VMware environment.

VMware [6] and the like are virtual machine managers in which multiple operating systems, running multiple applications, can coexist within the same hardware.

The Guide

Choosing the right virtualization technology can be complicated and depends on many factors. Additionally, many of the available Sun virtualization technologies can coexist, adding functionality but also complexity. How is a system administrator to choose? Table 1 was designed to help guide such a decision. In this chart are summaries of all the major attributes of the technologies, including functionality and limitations. Use this as your initial guide, and then read the Explanation section to sort out the details.

Feature	Dynamic System Domains	LDoms	Zones / Containers	xVM Server	VMware ESX
Available on	USPARC	Niagara I and II	SPARC and x86	X86	X86
OS	≥Solaris 8	S10	S10	xVM Server host, Windows, Linux, Solaris, and other guests	VMware ESX host, Windows, Linux, Solaris, and other guests
Separate kernels/ packages/patches?	Yes	Yes	No	Yes	Yes
Fault isolation	Most	Some	Less	Some	Some
ISV support	None needed	None needed	Needed	Some needed	Some needed
Moveable between servers	Only via SAN boot, downtime	Yes, via warm migration	Yes, via attach/detach, downtime	Yes, live	Yes, via Vmotion, live
Resource management	Limited, at single CPU granularity	Yes, at single-thread granularity	Yes, at subthread granularity, very flexible	Yes, very flexible	Yes, very flexible
P to V effort	Using standard install tools	Using standard install tools	Solaris 8 Containers, Solaris 9 Containers, otherwise standard system tools	Physical to virtual tools included	Physical to virtual tools included
Efficiency	Higher	Higher	Higher	Lower	Lower
Speed to produce another instance	Slow (standard tools)	Slow (standard tools)	Fastest (cloning)	Fast (copying)	Fast (copying)
Other Features					Snapshots for version control
Maturity	Mature	Young	Middle-aged	Youngest	Middle-aged
Cost	Free	Free	Free	Free/$	$$

TABLE 1: COMPARISON OF THE SERVER VIRTUALIZATION TECHNOLOGIES AVAILABLE WITH SOLARIS 10

Explanation

This column includes enterprise virtualization features, not desktop options. For desktops there are also many choices, including the new kid on the block, VirtualBox. It is now owned by Sun, open source, and feature-rich. It's available at http://www.virtualbox.org/.

Standard install tools include install from DVD, Solaris Jumpstart, or Flash Archive and other network installation tools. Each of these has functions and limits that need to be planned for. For example, Flash Archive can only capture and build like-architecture systems (e.g., you can't capture a SPARC system and build an x86 system from that image).

There are several important limits with respect to Domains. For example, the system must be preconfigured to allow domains to change dynamically.

LDoms are limited to only the "coolthreads" CMT servers from Sun. For warm migration, the LDom's boot disk must be stored on an NFS server or SAN (and SAN functionality was limited at the time of this writing).

Some applications are not supported with Zones, so it is important to check the vendor support matrix for every application.

"Solaris 8 Containers" and "Solaris 9 Containers" are a commercial offering from Sun allowing physical to virtual (P to V) transfer of existing S8 or S9 (SPARC) systems into containers within S10 (SPARC).

DTrace may be used within Zones, but it may not probe kernel resources. Only the "global zone" (Solaris, not within a Zone) may do that. Further, DTrace can only go so far when used in conjunction with other operating systems. With xVM server, DTrace will be usable within the host but likely not provide any value looking within the guests (unless they are Solaris too).

xVM Server has not yet shipped (at the time of this writing) so the expected features may or may not be included or work as expected.

Each of these technologies has features to manage the resources used by the guests. Those technologies vary wildly in their abilities, ease of use, and ease of monitoring. Generally, domains are the least granular and zones are the most granular. If you have specific resource management needs, then a thorough read of the documents and an on-line search for best practices are your next steps.

The creation of a new "clone" virtual machine from an existing virtual machine is a very useful feature. Although some tricks can be played to speed up the process, I only address the feature set included with the virtualization technology when describing how quickly a clone can be made. For example, if SAN or NAS boot is used for Domains or LDoms, the SAN or NAS capabilities can be used to quickly clone new instances.

Certainly, the efficiency of all of these solutions will vary depending on many aspects of the deployment. However, I assert that the closer to the raw hardware the applications are, the faster they will be. For example, there is essentially no difference in the path that the application code takes when run within a Domain or on a Sun server without virtualization. Similarly, Zones are very efficient because only one kernel is running. Hundreds of zones can run within one system. Of course, with VMware and ESX Server, if there is only one kind of guest operating system running then the system likewise can be highly efficient. As with all performance analysis, consider the details of your specific deployment to determine how efficient one virtualization technology would be compared to the others.

More so with virtualization than many other technologies, the details, features, and functionality are changing all the time. I plan to make this article available on-line and keep the chart up-to-date, so if you are interested in updates keep an eye on my blog at http://www.galvin.info.

Conclusion

The many virtualization choices on Sun platforms vary in all degrees. Fundamentally, each has pros and cons, as well as uses and abuses. This guide should help sort through all of those aspects to help you determine the best choice(s) given your environment and needs. Aside from the features, be sure not to lose track of the complexity added with each of these technologies. Management, monitoring, debugging, and performance analysis and tuning are all major aspects of a virtualized environment. That is where tools such as VMware Virtual Center and the new Sun xVM Ops Center can ease the burden and control virtual machine sprawl. Consider the tools to manage the environment as well as the environment itself.

Random Tidbits

Sun LDoms 1.1 shipped in January and added new features (as described in this column).

Also, if you have any interest in using OpenSolaris, check out the recently published book *The OpenSolaris Bible* [7].

There is a new Sun blueprint that goes into great detail on configuring Sun 7000 storage for running Oracle databases [8].

REFERENCES

[1] http://www.usenix.org/publications/login/2008-10/openpdfs/hu.pdf.

[2] http://docs.sun.com/source/819-3601-14/Domains.html.

[3] http://www.sun.com/servers/coolthreads/ldoms/index.jsp.

[4] http://en.wikipedia.org/wiki/Solaris_Containers.

[5] http://xvmserver.org/.

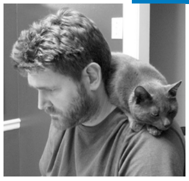
[6] http://www.vmware.com/.

[7] N.A. Solter, J. Jelinek, and D. Miner, *The OpenSolaris Bible* (New York: Wiley, 2009).

[8] http://wikis.sun.com/display/BluePrints/Configuring+Sun+Storage+7000+Systems+for+Oracle+Databases.

DAVE JOSEPHSEN

iVoyeur: top 5 2008



David Josephsen is the author of *Building a Monitoring Infrastructure with Nagios* (Prentice Hall PTR, 2007) and is Senior Systems Engineer at DBG, Inc., where he maintains a gaggle of geographically dispersed server farms. He won LISA '04's Best Paper award for his co-authored work on spam mitigation, and he donates his spare time to the SourceMage GNU Linux Project.

dave-usenix@skeptech.org

HAVE YOU EVER BEEN SO BUSY THAT the phrase "I've been busy" rings hollow and cliche in your ears? In the past year, desperate to position ourselves to scale at the pace our tiny company is growing, my cohort and I have ripped out, redesigned, and replaced our entire production infrastructure. Our routers went from IOS to JunOS to BSD, and our balancers from Alteons to Mod_proxy_balancer to balancer clusters using Carp to those using ClusterIP. Static routes over IPSEC VPNs have been replaced with OSPF routes over SSL VPNs. We fought through PCI certifications, designed IP and DNS standards, and implemented BGP-based datacenter fail-over systems. Anyway . . . I've been busy.

So busy, in fact, that for the first time since I discovered USENIX, I didn't make it to a single conference this year, USENIX or otherwise. January finds me shell-shocked, wondering where the time went, and curious as to what great work I missed out on in 2008. Fed up with being in the dark, I immediately declared a paper-reading weekend and set about going through conference proceedings to see what went on outside the cave I've been trapped in. I wasn't disappointed; as usual, the papers track of the various USENIX Cons in 2008 put me to shame. I might as well be a janitor. Been busy? Well, if you've been missing out on some great work, let me help you out with this list of my top 5 favorite monitoring-related papers of 2008.

We'll start with number 5, a paper called "Error Log Processing for Accurate Failure Prediction" [1]. This paper is from the new USENIX workshop on the analysis of system logs. With apologies to the authors of this paper whose intent was to explore failure prediction, this paper caught my attention because of the several clever log preprocessing techniques they used on free-form system logs to make them more machine-readable. These included removing data such as numbers from the logs, using Levenshtein distance to categorize and group similar individual events, and making use of a technique called tupling to identify multiple event entries that correspond to the same actual error. The log preparation sections in the preamble of this paper are of immediate practical use to folks who have a lot of logs and are looking for some quick ways to get a handle on them.

Speaking of logs, number 4, a tool paper called “Picviz: Finding a Needle in a Haystack” [2], applies a visualization technique called parallel coordinate plots to system logs. If you’ve read Greg Conti’s excellent book *Security Data Visualization*, then you’ve seen parallel coordinate plots used to great effect to plot port scans. These graphs take any number of variables and assign them columns in the vertical plane. A single element of data made up of those variables is then represented as a line in the horizontal plane. Figure 1 is an example parallel coordinate plot of SSH logins shamelessly stolen from the Picviz Web site.

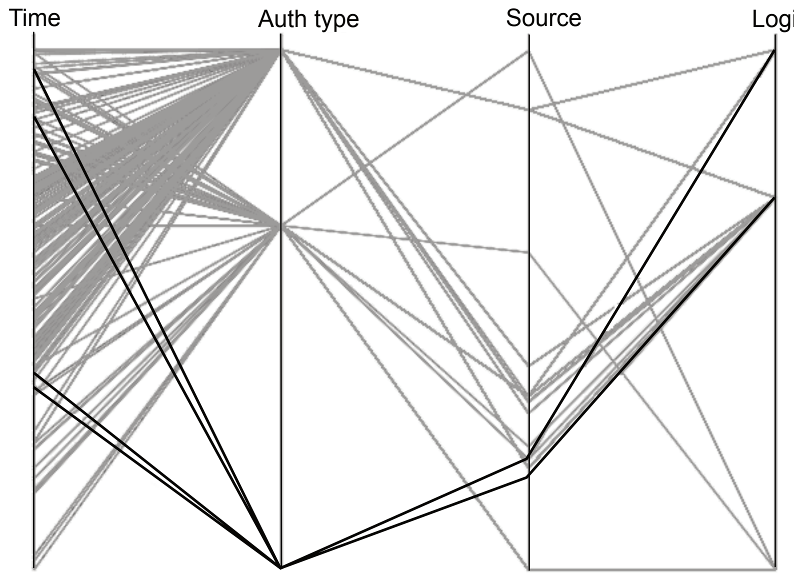


FIGURE 1: THIS PARALLEL COORDINATE PLOT MAKES IT EASY TO SPOT “SUSPICIOUS” SSH CONNECTIONS THAT ORIGINATE FROM MULTIPLE SOURCES BUT USE THE SAME USER ID.

The Picviz paper has a lot of elements that I like in a good paper. They’ve identified and solved a problem for me; catching odd stuff going on in a large amount of log data is difficult, and Picviz makes it easier. Behind the paper is a GPL’d tool that I can get my hands on and play with right now. Too often, promising work never goes very far for lack of an available implementation. And, speaking of implementation, I really like Picviz’s design, which mimics that of GraphViz. The authors have created a mark-up language for drawing parallel coordinate plots, which means I can write glue code to connect it to existing parsers and monitoring apps.

They didn’t stop at providing a framework; they used their own mark-up language to write a GUI for lighter-weight, interactive, or real-time use. I’d use Picviz over something like Conti’s Rumint [3] when I want something less purpose-specific and more flexible. I can graph whatever I want with Picviz whether my data originated from log files, PCAP dumps, or NetFlow logs. I can use it to bolt parallel coordinate plots onto existing dashboards, and it works equally well as a forensics or real-time IDS tool. Good work.

The writers of my third favorite monitoring-related paper in 2008 had a fascinating problem, that of being ignored. It seems that botnet attacks against Web applications have increasingly abandoned the traditional brute-force scanning mechanisms in favor of clever Google searches. This is bad news for folks trying to write honeypots, since unless your honeypot gets indexed in a way that piques the interest of the attacker, no traffic will arrive at your honeypot. So the problem becomes getting a single machine to respond to search robots in a way that makes it attractive to whatever the botnets happen to be into that week.

Their rather ingenious solution, described in “To Catch a Predator: A Natural Language Approach for Eliciting Malicious Payloads” [4], was to use natural language processing techniques to generate dynamic responses to the indexing services based on real host interactions. The methodology they used to glean their training data was clever, as is their solution to a rather sticky problem. And although this work is probably not immediately useful to those of us outside botnet research, I think their methodology could really blow the roof off the honeynet, tar pit, and protocol misdirection scene. It seems like the kind of technique that would find unexpected practical application all over the place. Good work.

Number 2 on the list is a paper called “CloudAV: N-Version Antivirus in the Network Cloud” [5]. I’ve long been flummoxed by the concept of multi-version programming. I’ve read several papers arguing about the reliability it adds or doesn’t and whether its core assumption of statistical independence is borne out or not. I have a pretty good mental picture of how it’s supposed to work, but I’ve never been able to come up with a problem domain into which it seems to fit. The concept is, given some problem, you write *N* different software solutions and compare the results. Folks argue whether it’s actually possible to come up with *N* independent ways to solve a single problem (in other words, there’s an assumption the *N* solutions can be statistically independently derived, and this doesn’t hold water). Folks also argue whether solving a problem *N* times gives you a better solution or not, or they wonder why you would want to do that work.

Well this paper solidified my thinking: malware detection is a great problem domain for multi-version programming. It’s a problem, in fact, for which most of us have used *N*-version programming without ever realizing it. Every moderate-sized to large company I’ve ever worked for has used at least two different anti-virus programs from competing vendors, usually at least one at the mail gateway and another on the desktop. That viruses sometimes hit the desktop that aren’t caught by the MX might suggest that competing anti-virus software is independent enough in practice to work, or maybe not. Perhaps the viruses hit the desktops from a different vector, or maybe there was a temporal problem with updates. In practice, it doesn’t really matter. Having some heterogeneity there always seemed like a good idea.

I’ll be honest: I *hate* the entire realm of malware detection. I’ve always put a large space between myself and those unfortunate enough to be tasked with managing the corporate/campus/whatever AV system. That software has been a buggy, intrusive, ill-thought-out nightmare for years. CloudAV, however, if the implementation does what the paper says it does, has single-handedly made AV not suck anymore (or at least reduced the suck factor by several orders of magnitude). They’ve taken the multi-version programming concept to the extreme, implementing ten anti-virus engines and two behavioral detection engines on a central Xen host running a VM for each engine. They then install lightweight clients on each host that trap file-creation system calls, blocking them while sending the new file to a broker in front of the scan engines. The broker, before sending the file back for scanning, records metadata about the file in a database and checks it against hashsums of already scanned files, preserving network bandwidth and creating a gold mine of forensic data.

The client can run in three modes: transparently allowing all user actions while sending files to the scan server; blocking user actions and warning the user if a suspicious file is encountered; or blocking user actions and ultimately preventing them regardless of what the end user thinks. I'm not clear on how policy is enforced in the system (e.g., what prevents end users from killing the client locally, a question I expect was asked in the session), but IMO whatever problems it has are certainly more than made up for in what it delivers.

Right off the bat it provides a model the malware hasn't accounted for, so it goes a long way toward limiting the malware's ability to detect and attack the AV system itself. It eliminates the need to maintain definitions on end-user systems, as well as eliminating pretty much every other virus-software integration problem that so plagues the desktops of the heathen solitaire proletariat. It completely insulates you from management decisions regarding AV vendor licensing; you'll never have to rip out McAfee on 1000 workstations to install Sophos instead, for example. And as a bonus it gives you a powerful forensics tool. Have a data sensitivity standard? CloudAV can tell you every host that touched a given file and in what context. Crazy good work. The only two problems it has are that I can't download it now, and I can't buy it now.

Finally, my number 1 pick in 2008 is "Sysman: A Virtual File System for Managing Clusters" [6], although a more accurate title might be "Sysman, a Virtual File System for Managing Whatever You Have Plugged into the Wall." First off, I should warn you that I'm a bit of a sucker for filesystem interfaces. You might have guessed this if you've read my recent NagFS articles. I know XML is all the rage, and configuration management doesn't equate to remote control, but nothing beats /proc when it comes to management simplicity and leveraging existing skills. OpsWare understood this when they created its Global Shell (GS) before the company was purchased by HP. Long have I wished for an OpsWare GS that I could afford. We've even discussed rolling it ourselves several times in FUSE. The one year I don't get to go to LISA, along comes Sysman to scratch that itch. I really wish I had been at this session—I have a lot of questions.

Curiously, the Sysman authors don't seem to have heard of OpsWare Global Shell, but no matter, they appear to have gotten it right the first time. Sysman creates a filesystem containing directories that represent devices on the network. These devices can be detected and set up automatically and their features automatically become subdirectories inside Sysman. Upon detecting Linux servers, for example, Sysman will create a subdirectory representing the server and subdirectories inside the server directory corresponding to the server's proc and sys directories. Reading from /sysman/10.10.1.111/proc/cpuinfo will return the contents of the cpuinfo file in the proc directory on server 10.10.1.111. A "commands" file provides access to remote command execution: write to a server's commands file to send it a command and then read from its commands file to glean the last command's output.

Filesystem interfaces are great because you can bring all of your existing scripting skills to bear, and existing tools gain enormous amounts of power. Imagine how easy Nagios Event Handlers are to write given a Sysman filesystem, for example. Have a down Apache daemon? Just do:

```
echo '/etc/init.d/apache restart' > /sysman/apachehost/commands
```

There are considerations, of course, including security and stability, but you get my drift. If CloudAV has the potential to actually get me interested in reining in the malware problem, then Sysman has the potential to completely turn my world on its ear. A filesystem interface to my entire datacenter? Are you kidding me? It would be the most enabling thing to happen to me since I learned regular expressions. Simple, powerful, elegant: great work. Now, where can I get it? There's an eight-year-old version on SourceForge that I doubt is very functional. What gives, guys? Been busy?

REFERENCES

[1] F. Salfner and S. Tschirpke, "Error Log Processing for Accurate Failure Prediction," WASL '08: http://www.usenix.org/events/wasl08/tech/full_papers/salfner/salfner_html.

[2] S. Tricaud, "Picviz: Finding a Needle in a Haystack," WASL '08: http://www.usenix.org/events/wasl08/tech/full_papers/tricaud/tricaud_html.

[3] Rumint, open source network and security visualization tool: http://www.rumint.org/.

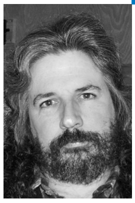
[4] S. Small, J. Mason, F. Monrose, N. Provos, and A. Stubblefield, "To Catch a Predator: A Natural Language Approach for Eliciting Malicious Payloads," Security '08, http://www.usenix.org/events/sec08/tech/full_papers/small/small_html/index.html. [Editor's note: The authors of this paper also wrote a related article that appears in the December 2008 issue of *;login:*.]

[5] J. Oberheide, E. Cooke, and F. Janhanian, "CloudAV: N-Version Antivirus in the Network Cloud," Security '08: http://www.usenix.org/events/sec08/tech/full_papers/oberheide/oberheide_html/index.html.

[6] M. Banikazemi, D. Daly, and B. Abali, "Sysman: A Virtual File System for Managing Clusters," LISA '08: http://www.usenix.org/events/lisa08/tech/full_papers/banikazemi/banikazemi_html/index.html.

ROBERT G. FERRELL

/dev/random



Robert G. Ferrell is an information security geek biding his time until that genius grant finally comes through.

rgferrell@gmail.com

I SUPPOSE I'LL ALWAYS REGARD LARRY Wall as the enlightened soul who finally coaxed my primal programming urge to the fore. Oh, I'd played around with Cobol, Snobol, C, Pascal, Natural, and a few other languages in an idle fashion, the way one might toy with an Elvis-shaped bowtie pasta noodle during a boring testimonial dinner, but I made no commitment to any of them. We both knew it was merely a case of ships passing in the night and nothing lasting would come of it.

I can no longer recall exactly when or where I first encountered Perl. I was using it extensively for CGI scripting by early 1996, so it must have been at least a couple of years prior to that. I was a corporate Webmaster back in the Paleozoic Era of the Web (1994), just after the release of Mosaic. In those days content was still mostly text, Gopher/WAIS was the only real way to search the Web, and the CGI standard hadn't yet been developed. Men were real men, women were real women, and both of us had to trudge for miles uphill both ways through blinding snowstorms to get to the coffee machine every morning. Oh, and our TV remotes only had five or six buttons.

As far as I can remember, the siren song that captivated me where Perl was concerned was that being interpreted rather than compiled meant instant feedback, especially when using the debug command-line option. Prior to that I'd been accustomed to the routine of header files, libraries, and hoping against hope that I'd made the proper ablutions to the compiler gods in order to experience the fruits of my coding labors. Now when I made a careless mistake (notice I said "when," not "if") I found out about it immediately and with a reasonable hope of tracing the problem to its source before my next birthday.

Add to that the fact that I actually understood the syntax of Perl on a level not experienced with any previous language and you can probably see why Larry's brainchild was almost on the level of spiritual revelation for me. My mental syntax and Perl's are similar to begin with (snickering allowed only in designated snickering areas, owing to the potential negative health effects of secondhand mirth), so I didn't have to do a lot of translation of thought patterns into formal semantics. This certainly sped things up for me and made troubleshooting programming errors less painful. One curious side effect was that I began to dream in Perl. This is symbolism that would have driven Sigmund Freud quite sane.

My Perl code is not what you would term "elegant," unless you employ that same adjective to describe frozen microwave cuisine. It works—most of the time—and that's about all I feel comfortable asking of myself. I have bought a significantly large fraction of the books ever written about Perl, such as *The Perl Cookbook*, *Object Oriented Perl*, *Perl for Toasters*, and so on, but I never seem to get much out of them. Oh, I might get on a kick and turn out more professional-looking code with fancy-schmancy stuff like comments for a while, but eventually I slide back into my established habits. You can't teach an old primate new hacks.

Like this CPAN thing. I never really got the hang of using modules. In the time it took me to figure out exactly what each one did and the syntax for using it in my own scripts, I could usually thrash out an equivalent, if less sophisticated, method myself. I guess part of my reluctance to jump on the module bandwagon was due to the fact that the keyboard I had on the AViiON box where I did most of my coding in the early days of CPAN had a very sticky shift key. That meant that, while the semicolon worked fine, I had a heck of a time getting a colon out of it. To call most module methods you had to type two of those in a row and that was just more trouble than it was usually worth to me. By the time I got around to obtaining a new keyboard (which wasn't until we went from DG/UX to Solaris 2.5.1), the colon aversion was deeply ingrained.

I realize, of course, that my computer (via its keyboard) was in effect programming *me*. I, for one, welcome our new dopant-laden silicon overlords and wish them better luck in maintaining control over their servants than I've ever had. Turn about is, after all, fair play.

Now that I'm past fifty the colon has taken on another, more insidious aspect. Somewhere in middle age a man's organs just seem to revolt *en masse*, leaving him with a paunch, a mortgage, all the vigor of a store-window mannequin, and kids whose idea of a summer job is taking out the household trash once a week at fifty bucks a pop.

Perl in many ways was the centerpiece of my golden years as a system administrator. I was never a hotshot module jockey, as I've hopefully made clear, but I could whip out a one- or two-hundred-line Perl script to solve some pressing sysadmin issue in nothing flat. Last time I looked (which was admittedly about ten years ago), a dozen or so of my scripts were still floating around at Sun's BigAdmin site. I sent them those scripts, and they shipped me a Sun ballpoint pen. Seemed like a reasonable exchange. At least the pen is still useful.

book reviews

ELIZABETH ZWICKY, WITH MORTEN WERNER FORSBRING, TROND HASLE AMUNDSEN, STÅLE ASKEROD JOHANSEN, BRANDON CHING, JASON DUSEK, AND SAM F. STOVER

DESIGN PATTERNS EXPLAINED: A NEW PERSPECTIVE ON OBJECT-ORIENTED DESIGN

Alan Shalloway and James R. Trott

Addison-Wesley, 2005. 418 pages.
ISBN 0-321-24714-0

Here's a telling fact: to write the review of this book, I had to make my husband give it back to me, because I suggested it when he needed to solve a problem, and he had squirreled it away so he could read more of it. He did lend it back to me for the duration of the review, but not without regret.

I learned my object-oriented programming on the street, and my patterns mostly by tripping over them, so I've been looking for a good book explaining what patterns really, properly are, and how real programmers use them. This is a lucid, practical explanation of patterns and how you can use them to design programs.

It works through a bunch of basic patterns, motivating the use of the pattern by presenting a problem that is well solved by the pattern. It discusses issues in integrating pattern-based designing with other systems and provides an outline of how you can figure out what patterns to use and how to string them together, starting from a customer specification, and including suggestions on identifying what the customer left out of the specification.

This is a good starter book on patterns, offering a refreshingly concrete take on a very abstract subject. It will make the most sense to somebody with programming and program maintenance experience, and it helps to know something about Agile/XP practices as well (at least, you should be aware that they exist). If you don't have a basic familiarity with patterns and the pattern literature, start at the beginning. There's no reason one might believe otherwise, but my poor husband got thrown in at page 300-odd, and while he did figure out that it was in fact the answer he was looking for, he had some trouble with the references to "the Gang of Four," which he associated with politics (not Chinese politics, either, so my attempts to disambiguate merely dragged us in irrelevant directions). Starting at the beginning would have clarified this as a shorthand for the authors of the seminal book on patterns in programming.

BEYOND BARBIE & MORTAL KOMBAT: NEW PERSPECTIVES ON GENDER AND GAMING

Yasmin B. Kafai, Carrie Heeter, Jill Denner, and Jennifer Y. Sun

MIT Press, 2008. 350 pages.
ISBN 978-0-262-11319-9

If you're interested in gender issues in computing, this is a nicely nuanced collection. The issues for games are slightly different from the issues for other kinds of computing, but not massively so. Mostly, they're just more visible. The book also points out that gaming is explicitly linked to entry into other computing fields, particularly IT. (To play a networked game, you've got to get the computer on the network.)

This book explores several themes dear to my heart. For instance, appealing to women doesn't mean adding pink bows, and doing it well may give you a product that's more appealing to everybody. Women are, after all, not a different species. The social pressures that restrict women's access to computers are not all overt; they include subtle and unsubtle differences in where people can go when. Often it's not obvious to men that a situation may be unsafe or socially unacceptable for women and that "lack of interest" reflects an effective lack of access.

PROCESSING: A PROGRAMMING HANDBOOK FOR VISUAL DESIGNERS AND ARTISTS

Casey Reas and Ben Fry

MIT Press, 2008. 698 pages.
ISBN 978-0-262-18262-1

I love processing; I've said it before, and no doubt I'll say it again. I already reviewed (and loved) Ben Fry's *Visualizing Data*, which includes a basic introduction to processing suitable for people who're comfortable programming in something or other. This book is aimed, as it says, at people who know more about pictures than about programs. Part of what it does is proselytize for the joy of making art with programs, and it assumes the reader knows nothing about programming. (Seriously, nothing.

Like, variables and loops are explained from the ground up.)

From there, it gets to complicated programs, and on the way it throws in handy tips I learned the hard way (such as the fact that sine and cosine functions are endlessly useful to give smooth variation). It covers the basics of programming, but the artistic bent is visible in any number of ways. The clearest are the sections about computer artworks, which are luscious, but the topics covered are also different from the ones you might expect. Anybody could figure out there'd be more about color handling, but if you haven't thought about computer-based art a lot, you might be surprised to note the early and extensive handling of sources of randomness and noise.

This is a good introduction to a good language for visual artists. It may also provide a good window into the world of art programming for programmers, although anybody with a basic knowledge of any object-oriented programming language is going to find themselves skipping a lot. For people who already do art programming, it may be worth it for the inspiration and the coverage of extensions, but if you're at all confident with programming, you'd do just fine with the Web site.

DAVID POGUE'S DIGITAL PHOTOGRAPHY: THE MISSING MANUAL

David Pogue

O'Reilly, 2009. 284 pages.
ISBN 978-0-596-15403-5

I recently bought a new camera, and my husband asked if he should read the manual. I said that was a terrible idea, as I had barely survived the experience, and I have a lot more background than he does. But he wanted something that would help him understand more camera geekery. Normally, for somebody who doesn't know anything about cameras and wants to take better pictures, I recommend Nick Kelsh's books—*How to Photograph Your Life* is a good all-purpose choice. And he's read it, but he wants more: something with more technical details, but still reasonable for somebody who never can remember what an f-stop is and whether you want to increase or decrease exposure when you're trying to take a picture of something white.

As I had hoped, *David Pogue's Digital Photography* is the right choice here. It's a beginner's book, for sure, but it talks about both technique and what buttons to push. It puts everything in the context of photos you might want and how to get them, so you don't have to try to understand f-stops without first deciding that you care about how much of your picture is blurry and what's in focus. It gives a swift picture of the whole camera-buying and -using process, starting from selecting a camera, continuing through using it, dealing with the pictures, editing them, and showing them off.

That's a lot to cover in 284 pages, but he does a good job of hitting the high points, for the most part. When there's no right answer to a dilemma, he's willing to say so. Should you delete photos on the camera or after downloading? It depends; do you have space left? Is looking at them in glorious detail going to help you decide between them or merely depress you?

I disagree with him on some points, of course; I used to love lens-cap tethers, but they disappointed me too often, either by failing suddenly or by letting a lens cap or tether dangle into my photo. The front cap I just leave the way it comes. But I have to modify the back cap with a dot of contrasting paint, because one of my lenses is terribly finicky about alignment.

This is a good introduction to digital cameras to people who are not intimidated by technology but don't know the nuts and bolts of photography. It's not going to satisfy a serious film photographer looking to switch to digital, and it's not going to give you much geek cred. It is going to help the average person take better photographs.

LEARNING NAGIOS 3.0

Wojciech Kocjan

Packt Publishing, 2008. 301 pages.
ISBN 978-1-847195-18-0

REVIEWED BY MORTEN WERNER FORSBRING, TROND HASLE AMUNDSEN, AND STÅLE ASKEROD JOHANSEN

The University of Oslo has been using Nagios as the primary monitoring tool for UNIX servers and their services for two years. We switched to version 3.0 only some months after its release. We monitor about 750 hosts and, in Nagios terms, approximately 11k services, in a diverse and heterogeneous academic installation with ~60k users. We have spent much time tuning and adapting Nagios to our needs, and we know some of its strengths and weaknesses, although we do not use or know in detail all of its features.

This book aims to be a general guide to Nagios as well as to the new features in version 3.0. Our impression is quite good. The focus is said to be on giving a good introduction to system administrators who are new to Nagios and want to learn more about it in general. In our opinion, the author does indeed provide a good overview of Nagios as an administration tool and offers a good start for people who are curious about trying it out.

The book does not venture into all of the challenges we have faced with Nagios, so people who are looking for more hardcore information might need to look deeper in Nagios's own documentation or on the various mailing lists.

That being said, in its 11 chapters the book covers basically all the steps necessary to get a more-than-basic Nagios rig up and running. Everything from installation to using more complex but essential features such as NRPE and NSCA is covered. In between you'll find explanations on what monitoring really is all about and the role of Nagios, as well as configuration and explanations on how to monitor specific typical things (e.g., email, processes, how ports answer).

There is also a focus on scalability and tuning, with good examples of "best practice." We also like the fact that the book mentions using Nagios in larger environments—for example, discussing the challenges of handling SSH checks. We agree that, for many, NRPE will be a better choice, but you have to pay attention to its security issues.

What do we miss? Considering the title, the book does its job. We would have liked more info on tuning the GUI, but many use it less than we do, and there are projects to improve it.

There are also several other things the book does not include, but it feels unfair to expect too much detail from a book targeted at beginning Nagios users.

What did we learn? We had several "wakeups," where we got reminded about things we want to look into—for instance, features we haven't started using yet. In this sense we think this book could be a useful read for many people who use this great software already. The book also covers many of the new features in version 3.0, which are very useful for those who are already into Nagios to be aware of.

Regarding the features of Nagios we don't use, we naturally cannot evaluate the content. This includes: monitoring Windows; extensive use of dependencies; adaptive monitoring; and running multiple Nagios servers.

This book is a good read for all system administrators who want to learn more about Nagios or want to start using it. Although some large installations may require more configuration and adaptation than this book goes into, it does cover all the important topics to get you up and running, and then some.

WEB 2.0: A STRATEGY GUIDE

Amy Shuen

O'Reilly Media, 2008. 266 pages.
ISBN 978-0596529963

REVIEWED BY BRANDON CHING

Some say that imitation is the sincerest form of flattery. If that is indeed the case, then Amy Shuen has written an informative guideline for all of us to flatter the likes of Google, Flickr, Amazon, and Facebook. In *Web 2.0: A Strategy Guide*, Shuen analyzes the Web 2.0 movement by deconstructing the strategies used by successful Web companies large and small; then she explains the how and why of their success through Web 2.0 principles.

Shuen details, in six chapters of highly informative analysis, how companies can adopt Web 2.0 ideas to enhance profitability and expand their businesses. Shuen does this through a detailed analysis of many successful Web 2.0 companies and many of the strategies they used to add value and profitability to the services they offer. According to Shuen, the strategies of primary importance to Web 2.0 businesses are to build on collective user value, activate network effects, work through social networks, dynamically syndicate competence, and recombine innovations.

Shuen begins by analyzing the collective user value concept by scrutinizing the strategies of Flickr and Netflix. The user value concept is the idea that users, not the companies themselves, are the main contributors of value. By allowing users to add, share, organize, and promote their own content, businesses need only provide a context through which users can interact. Although this may sound easy, balancing user demands while ensuring profits and adding value to services is, as Shuen points out, difficult to attain.

Building upon the collective user value concept is the network effect. "Positive network effects increase the value of a good or service as more people use or adopt it" (p. 41). In this chapter, Shuen breaks down how Google's pay-per-click keyword advertising has used a variety of positive network effects to generate sustainable profitability ($8 billion per quarter, p. 44). With coverage of five different types of positive network effects, Shuen provides a solid overview of ways your business can capitalize on this core Web 2.0 principle.

Perhaps one of the most visible Web 2.0 strategies is that of social networking. Chapter three delves into the understanding of this effect and covers topics such as Malcolm Gladwell's tipping point theory, the Rogers adoption curve, and the Bass diffusion curve. These more theoretical explanations for social networking effects Shuen follows by an analysis of the rise in membership, participation, and success of Facebook and LinkedIn.

Every business has a set of core competencies. Chapter four analyzes IBM, SalesForce.com, and Amazon, showing how each company used their core competencies in dynamic ways to expand service offerings and increase profits. In the case of Amazon, the decision to open up their online book-selling service to third-party sellers in 2001 resulted in additional billions of dollars in sales. Later, capitalizing on their experience of running a large, multi-million-user system, Amazon began offering back-office competencies through Web services such as the Simple Storage Service (S3) and the Elastic Compute Cloud (EC2).

Finally, Shuen covers the recombining innovation aspect of the Web 2.0 movement by addressing how businesses can expand upon existing services utilizing different modes of innovation including democratized, crowd-sourcing, ecosystem, platform, and recombinant innovation. The most prominent example Shuen addresses here is that of Apple's iPod and iTunes application capitalizing on platform innovation through iPod assembly, creative software, accessories, and user-provided music data.

The final chapter is a short how-to consisting of five steps on how to incorporate Web 2.0 strategies into your business. Essentially a summary of the previous five chapters, it also provides general business ideas and strategies on getting started with your newly acquired Web 2.0 knowledge. Weighing in at only 15 pages, this chapter won't give you all your answers, but it does give you a good push in the right direction.

Shuen's book is an exceptional analysis of many processes and strategies used by successful Web 2.0 companies. Each chapter ends with a valuable series of strategic and tactical questions that can be asked by business owners and executives which directly relate the principles in the chapter to Web and IT-based organizations.

Web 2.0: A Strategy Guide should be considered required reading for business executives and students, entrepreneurs, and information technology investors. Although approachable, the book does contain a good amount of business and economic principles and terminology that may leave some just skimming a few pages here and there. Yet, overall this was a very good, and relatively concise, source of valuable information on the business strategies and principles of successful Web 2.0 businesses.

REAL WORLD HASKELL

Bryan O'Sullivan, John Goerzen, and Don Stewart

O'Reilly, 2008. 710 pages.
ISBN 978-0-596-51498-3

REVIEWED BY JASON DUSEK

Haskell has long lacked a book with both up-to-date worked examples and an overview of language fundamentals. *Real World Haskell* introduces pure functional programming with the Haskell language and covers a wide array of libraries that are in active use.

Most programmers coming to Haskell will find their greatest difficulty lies in learning purely functional programming and type inference. These two features allow a rich combination of optimization, code verification, and script-like brevity. *Real World Haskell* is veined with explanatory material, highlighting the utility of Haskell's novel features; for example, Chapter 25, on profiling and optimization, illustrates how functional purity and static types allow for a remarkable speedup without damage to clarity or code size.

At first, many common idioms appear impossible or nonsensical in the functional setting. Among them are error handling with exceptions, pointer arithmetic, and multithreaded programming with shared values. Nothing could be further from the truth, and *Real World Haskell* does an excellent job of clarifying both how these idioms fit in with purely functional programming and how they are executed in practice in Haskell.

Material in the text runs the gamut from introductory to practical to sophisticated. The first four chapters are devoted to introducing the language and pure functional programming; remaining chapters are divided between the case studies—syslog, sockets, databases, parsing /etc/passwd—and discussion of functional programming idioms. The famed science of monadology is made approachable without dumbing down.

Type inference, pattern matching, and the Haskell type checker are powerful aids to static verification. These unfamiliar tools are demonstrated throughout the text with appropriate comparison to idioms in other languages.

Some figures and footnotes are notably in error; this is generally obvious (e.g., a footnote labeled "a" but referenced by an asterisk). Most of the errors I encountered have found their way onto the errata page (at http://oreilly.com/catalog/9780596514983/errata/). There are some places where the text repeats itself (e.g., on page 574). These mistakes in presentation are a small blemish on fine material.

In referencing online documentation and API search tools, the book upholds the standard of all O'Reilly books. *Real World Haskell* is itself available online, at http://book.realworldhaskell.org/. The site supports per-paragraph comments and has been collecting notes since January of 2008.

ALGORITHMS IN A NUTSHELL

George T. Heineman, Gary Pollice, and Stanley Selkow

O'Reilly, 2008. 362 pages.
ISBN 978-0-596-51624-6

REVIEWED BY JASON DUSEK

Algorithms in a Nutshell offers an informal, compact reference to basic algorithms for sorting, searching, and graph traversal as well as more specialized material for computational geometry, pathfinding, and network flow.

The text touches upon a number of programming languages but the vast majority of examples are in Java, C, or C++. A chapter tends to use all one or another: the reader is assumed (not unreasonably) to have a reading knowledge of all three.

Deliberately light on math, the text's presentation is inspired by Christopher Alexander's *A Pattern Language,* which is introduced in the chapter "Patterns & Domains." After a short visit with the philosophers, this chapter introduces the finest organizational aspect of this text: the algorithm trading cards, short fact sheets with well-thought-out iconography to allow you, at a glance, to assess an algorithm's performance, operating principles, and relation to fundamental data structures. In addition to the trading cards, the book offers plentiful source code, pseudo-code, and diagrammatic explanations of the algorithms.

In keeping with the practical emphasis of the text, the authors generally benchmark the algorithm under study on two or three computer systems in addition to presenting its "big O" characteristics.

A few small printing errors detract from the text. On page 137, there is either an error or there is an airline with service from one's desktop to one's router.

Algorithms in a Nutshell is readable and informative; it is both an excellent refresher and a fine introduction to a number of important algorithms.

OS X EXPLOITS AND DEFENSE

Kevin Finisterre, Larry H., David Harley, and Gary Porteous

Syngress, 2008. 400 pages.
ISBN 978-1-59749-254-6

REVIEWED BY SAM F. STOVER

I walked away from this book with a burning question in my mind: Just who exactly is the target audience? In my opinion, if you are a reverse engineer who can walk though the stack but need screenshots to help you install KisMAC on a Mac and you are interested in a lot of OS 9 historical facts, this is the book for you. Instead of a book for newbies and veterans alike, this book feels unbalanced in its content.

The first chapter starts out with an incredibly brief description of the boot processes for OS 9 and OS X. The rest of the chapter discusses forensic toolkits designed for the Mac, which is really just a bunch of GUIs wrapped around *NIX utilities we've all been using for years. Chapter 2 then goes into an incredible amount of detail (e.g., discussing memory register contents) of several bugs in OS 9, early versions of OS X, and even Windows running via Cross Over/WINE. Very deep stuff.

Chapter 3 then takes a look at Mac-focused malware since the beginning of time. There is a lot of background and historical information; in fact the chapter contains more OS 9 information than I ever knew existed. Chapter 4 outlines the different tools, both past and present, to detect malware on the Mac platform. Again, there's lots of historical info, but maybe more than you would find interesting—definitely more than I found interesting.

Chapter 5, "Mac OS X for Pen Testers," was the chapter I was honestly looking forward to but was ultimately disappointed in. If you need help installing the Developer Tools, X11, CPAN, Wireshark, and nmap, then read this chapter. If you can figure that stuff out on your own, don't bother. The same goes for the next chapter, which addresses wardriving and wireless pen testing by walking you through installing and using KisMAC.

Chapter 7, "Security and OS X," is very reminiscent of Chapter 2 in that you get a very in-depth description of buffer overflows, library injection, and address space layout randomization. I thought this chapter was the highlight of the book: The information is current, valid, and interesting, and the explanations were solid.

The final chapter, on encryption implementations, starts with an overview of encryption in OS 9, then follows changes and additions in OS X such as File-

Vault, SSHD, WEP, and WPA. Not a bad chapter, if you are interested in learning the evolution of encryption in the Mac world.

Overall, I'd have to say that this book was a bit of a disappointment. I really liked Chapter 7, and even Chapter 2 held my interest, but beyond that, I felt as though I was being inundated with either OS 9 information I don't care about or installation instructions for open source tools. If you really want to learn about pen testing, on a Mac or otherwise, I'd recommend the *Penetration Tester's Open Source Toolkit*. If you are interested in learning more about your Mac, I'd recommend *OS X for Hackers*. But that's just me.

USER FRIENDLY by J.D. "Illiad" Frazer
OH, HA HA! SIR, IT'S JUST A SECURITY QUESTION TO ENSURE YOU'RE THE AUTHORIZED PARTY. SO WHO DO YOU BANK WITH?
COPYRIGHT©2008 J.D. "Illiad" Frazer HTTP://WWW.USERFRIENDLY.ORG/
UM. I DON'T HAVE A BANK.
BUT...YOU SAID YOU BANKED WITH US?
THAT'S BECAUSE YOU SAID YOU WERE CALLING FROM MY BANK. SO HOW MUCH DO I HAVE?

USENIX notes

USENIX MEMBER BENEFITS

Members of the USENIX Association receive the following benefits:

FREE SUBSCRIPTION to *;login:*, the Association's magazine, published six times a year, featuring technical articles, system administration articles, tips and techniques, practical columns on such topics as security, Perl, networks, and operating systems, book reviews, and summaries of sessions at USENIX conferences.

ACCESS TO ;LOGIN: online from October 1997 to this month: www.usenix.org/publications/login/.

DISCOUNTS on registration fees for all USENIX conferences.

DISCOUNTS on the purchase of proceedings from USENIX conferences.

SPECIAL DISCOUNTS on a variety of products, books, software, and periodicals: www.usenix.org/membership/specialdisc.html.

THE RIGHT TO VOTE on matters affecting the Association, its bylaws, and election of its directors and officers.

FOR MORE INFORMATION on membership or benefits, please see www.usenix.org/membership/ or contact office@usenix.org. Phone: 510-528-8649

USENIX BOARD OF DIRECTORS

TRIBUTE TO JAY LEPREAU, 1952–2008

Ellie Young, USENIX Executive Director

As many of you may know, Jay Lepreau, Research Professor and Director of the Flux Research Group in the School of Computing at the University of Utah, died last September, following complications of his battle with cancer. Jay was a valued supporter of USENIX for over 25 years, presenting numerous papers and introducing generations of students to USENIX.

As program chair of the USENIX 1984 Technical Conference, Jay instituted the USENIX tradition of providing published proceedings for the attendees.

One of Jay's most enduring contributions was to conceive and found the USENIX Symposium on Operating Systems Design and Implementation (OSDI).

To commemorate Jay's contribution to USENIX and to computer science, USENIX inaugurated the Jay Lepreau Best Paper Award at the eighth OSDI. A tribute for Jay and the presentation of the first awards were held on Monday, December 8, 2008, at OSDI '08. Watch the video or listen to the MP3 of the tribute:

http://www.usenix.org/media/events/osdi08/tech/videos/lepreau_320x240.mp4

http://www.usenix.org/media/events/osdi08/tech/mp3s/Inauguration.mp3

writing for *;login:*

Writing is not easy for most of us. Having your writing rejected, for any reason, is no fun at all. The way to get your articles published in *;login:*, with the least effort on your part and on the part of the staff of *;login:*, is to submit a proposal to login@usenix.org.

PROPOSALS

In the world of publishing, writing a proposal is nothing new. If you plan on writing a book, you need to write one chapter, a proposed table of contents, and the proposal itself and send the package to a book publisher. Writing the entire book first is asking for rejection, unless you are a well-known, popular writer.

;login: proposals are not like paper submission abstracts. We are not asking you to write a draft of the article as the proposal, but instead to describe the article you wish to write. There are some elements that you will want to include in any proposal:

- What's the topic of the article?
- What type of article is it (case study, tutorial, editorial, mini-paper, etc.)?
- Who is the intended audience (syadmins, programmers, security wonks, network admins, etc.)?
- Why does this article need to be read?
- What, if any, non-text elements (illustrations, code, diagrams, etc.) will be included?
- What is the approximate length of the article?

Start out by answering each of those six questions. In answering the question about length, bear in mind that a page in *;login:* is about 600 words. It is unusual for us to publish a one-page article or one over eight pages in length, but it can happen, and it will, if your article deserves it. We suggest, however, that you try to keep your article between two and five pages, as this matches the attention span of many people.

The answer to the question about why the article needs to be read is the place to wax enthusiastic. We do not want marketing, but your most eloquent explanation of why this article is important to the readership of *;login:*, which is also the membership of USENIX.

UNACCEPTABLE ARTICLES

;login: will not publish certain articles. These include but are not limited to:

- Previously published articles. A piece that has appeared on your own Web server but not been posted to USENET or slashdot is not considered to have been published.
- Marketing pieces of any type. We don't accept articles about products. "Marketing" does not include being enthusiastic about a new tool or software that you can download for free, and you are encouraged to write case studies of hardware or software that you helped install and configure, as long as you are not affiliated with or paid by the company you are writing about.
- Personal attacks

FORMAT

Please send us text/plain formatted documents for the proposal. Send proposals to login@usenix.org.

DEADLINES

For our publishing deadlines, including the time you can expect to be asked to read proofs of your article, see the online schedule at http://www.usenix.org/publications/login/sched.html.

COPYRIGHT

You own the copyright to your work and grant USENIX permission to publish it in *;login:* and on the Web. USENIX owns the copyright on the collection that is each issue of *;login:*. You have control over who may reprint your text; financial negotiations are a private matter between you and any reprinter.

FOCUS ISSUES

In the past, there has been only one focus issue per year, the December Security edition. In the future, each issue may have one or more suggested focuses, tied either to events that will happen soon after *;login:* has been delivered or events that are summarized in that edition.

conference reports

THANKS TO OUR SUMMARIZERS

OSDI '08: 8th USENIX Symposium on Operating Systems Design and Implementation

San Diego, CA
December 8–10, 2008

The Jay Lepreau Award for Best Paper was instituted at OSDI '08 (see p. 87). Three awards were presented.

CLOUD COMPUTING

Summarized by Dutch Meyer (dmeyer@cs.ubc.ca)

- ***DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language***
  Yuan Yu, Michael Isard, Dennis Fetterly, and Mihai Budiu, Microsoft Research Silicon Valley; Úlfar Erlingsson, Reykjavík University, Iceland, and Microsoft Research Silicon Valley; Pradeep Kumar Gunda and Jon Currey, Microsoft Research Silicon Valley

Winner of Jay Lepreau Award for one Best Paper

Yuan Yu explained that DryadLINQ comprises Microsoft's cluster computing framework (Dryad) and the .NET Language INtegrated Query system (LINQ). Together, these tools allow programmers to write in their familiar .NET languages, with integrated SQL queries. In the compilation process, DryadLINQ can produce an executable appropriate for a single core, a multi-core, or a cluster of machines. The tool chain is incorporated into Visual Studio as well, allowing programmers to work in a familiar IDE. The result is an environment for concurrent programming that transparently works in the much more difficult field of distributed programming. In summary, Yuan described the work as a "modest step" toward using the cluster as one would use a single computer.

As an example, Yuan showed how a simple task (counting word frequency in a set of documents) could be expressed with the system. The LINQ expression "select many" is used to transform the set of documents into a set of words, then "group" is used to collate the words into groups. Finally, the count can be computed for each word. At each step, the use of LINQ's relational algebra ensures that the task will parallelize well, while Dryad provides benefits such as data location transparency.

A recurring element of both the presentation and the question-and-answer period was comparing DryadLINQ to MapReduce, as is used by Google. Yuan offered Dryad as a more generalized framework for concurrent programming, showing how MapReduce could be implemented with three operations. He also posited that DryadLINQ provided a clear separation between the execution engine and the programming model, whereas in his view MapReduce conflates the properties of the two. This places unnecessary restrictions on both the execution engine and the programming model.

Yuan explained that the project was released internally and is being used in several projects. He described several lessons drawn from the effort, praising deep language integration, easy parallelism, and an integrated cross-platform environment. In describing future work, he proposed exploring the types of programs that could be created with this approach, asked how he could better generalize the programming model, and called for languages providing strong static typing in the datacenter.

Audience interest was strong for the paper, and session chair Marvin Theimer was forced to cut off discussion to keep the session on schedule. Eric Eide of the University of Utah began by asking what opportunities programmers have to control the compilation of their programs, for example optimizing for failure tolerance versus performance. In reply Yuan explained that some of these optimizations would need to be handed to the execution engine as a policy but that the problem has not yet been tackled directly by the compiler. Brad Chen from Google wanted to know whether the programming model facilitated good intuitive assumptions about performance from programmers. Similarly, Armando Fox wondered whether data mining and/or machine learning could give performance estimates. Yuan explained that they were still learning about the system and that people are actively working on understanding the performance characteristics. Modeling performance with machine learning is ongoing work. Yuan was also asked about support for continuous querying, and he thought that this would be an interesting area for future work.

- ***Everest: Scaling Down Peak Loads Through I/O Off-Loading***
  Dushyanth Narayanan, Austin Donnelly, Eno Thereska, Sameh Elnikety, and Antony Rowstron, Microsoft Research Cambridge, United Kingdom

Dushyanth Narayanan presented Everest, which uses I/O offloading to mitigate peak load conditions on storage servers. This work follows his paper at FAST earlier in the year, in which Dushyanth applied a similar mechanism to the problem of power consumption. He began by showing a trace of a production Exchange server; despite overprovisioning, an observed I/O load was shown to increases the server's response time twentyfold. At the same time, this load increase was not correlated with the workload on other disks. This key observation provides hope that spare resources on these other hosts can be utilized to lessen the burden on storage systems.

Everest operates within a client/server model. Any machine that needs protection against peak loads runs an Everest client, while a set of other machines operate Everest stores. In practice Everest clients may also be stores. Under normal operation, requests to an Everest client merely pass through to the local storage without modification. If the client detects that a peak load condition may be occurring (according to some threshold), it begins write offloading. This means preserving disk I/O bandwidth by issuing writes to Everest stores across the network interface. Once the peak subsides, the client stops offloading and begins reclaiming previously offloaded writes. Naturally, this solution targets temporary peaks in load; it will not provide long-term relief to a cluster that is fundamentally under-provisioned.

To minimize the burden of additional writes to the client's disk, the destination of offloaded writes is not written to local storage. Everest store nodes instead track the source of writes along with the data, written to a circular log. In recovering from a failure, clients query relevant stores to find all outstanding writes. Everest also includes features beyond this basic operation, such as offloading to multiple stores and load balancing, although these features were not discussed in depth.

Questions addressed a variety of motivational and technical details. Jonathon Duerig of the University of Utah gently challenged the assumption that there is no correlation between peak loads and asked whether there was a metric for how much correlation the system could endure. Dushyanth had not performed this analysis yet and was not aware of a synthetic workload that would allow experimenters to vary peak load correlation. Preston Crow of EMC asked whether disk caches could be grown to provide the same effect, but Dushyanth replied that the size of the observed peaks was in the gigabyte range—simply too large for a reasonably sized cache. Finally, Armando Fox of UC Berkeley wondered whether new failure modes were being introduced by spreading data across many volumes. Dushyanth thought that N-way replication, already present in Everest, was likely necessary to ensure that the system could tolerate failures in $N - 1$ Everest stores.

- ***Improving MapReduce Performance in Heterogeneous Environments***
  Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica, University of California, Berkeley

Matei Zaharia discussed the MapReduce programming model and the improvements his team at Berkeley has brought to Hadoop's implementation and algorithm. Their principal findings were based on the observation that performance heterogeneity breaks task-scheduling assumptions. Such heterogeneity is not hard to find in practice; for example, utility computing services don't generally provide strong performance isolation guarantees.

In the process of analyzing Hadoop, Matei used Amazon's EC2 service as an academic testbed, a direction he vigorously encouraged others to follow. In their experiments, they were able to operate at a convincingly large scale, yet with relatively low cost.

Backup selection in Hadoop occurs as nodes used for primary tasks become free. In that case, primary tasks that have not made sufficient progress will be replicated onto free nodes. The study found four principal problems with backups in Hadoop. First, too many backups will thrash network bandwidth and other shared resources. Second, the wrong tasks can be selected for backup. Third, backups can be directed to slow nodes, which is suboptimal. Fourth, if tasks are started at nonuniform times, scheduling decisions can be made incorrectly. In an example, Matei showed how 80% of the tasks for an operation could be selected for backup, with the majority of these tasks losing to the originals.

To fix these problems, a LATE scheduler was contributed to the Hadoop project (available at http://hadoop.apache.org/core/). The new scheduler estimates the completion time for each task and selects only slow-running tasks that are believed to complete late. It also caps the number of backup tasks and places backups on fast nodes. In the evaluation it was shown that this new scheduler offered a 58% performance improvement, on average.

Brad Chen of Google asked how tasks of variable length would be scheduled. In their work, Matei and his team had assumed that tasks were of roughly constant length. This is justified in that Hadoop itself attempts to maintain this invariant. To tackle this problem, Matei believed that one would probably need to prioritize according to task size; however, this problem has yet to be addressed. Marvin Theimer of Amazon also asked whether the authors had any advice for utility computing providers. The presenter encouraged more visibility into machine status, network topology, and rack locations.

OS ARCHITECTURE

Summarized by Dutch Meyer (dmeyer@cs.ubc.ca)

- ***Corey: An Operating System for Many Cores***
  Silas Boyd-Wickizer, Massachusetts Institute of Technology; Haibo Chen, Rong Chen, and Yandong Mao, Fudan University; Frans Kaashoek, Robert Morris, and Aleksey Pesterev, Massachusetts Institute of Technology; Lex Stein and Ming Wu, Microsoft Research Asia; Yuehua Dai, Xi'an Jiaotong University; Yang Zhang, Massachusetts Institute of Technology; Zheng Zhang, Microsoft Research Asia

Silas Boyd-Wickizer presented a new operating system interface as part of ongoing work on the Corey operating system. The work is premised on the observation that for application performance to scale with an increasing number of processing units, the time spent accessing kernel data structures must be made parallelizable. In current practice, synchronizing access to such data must be done conservatively. Although kernel subsystems can sometimes be redesigned to avoid or eliminate this locking, the process is time consuming for programmers. Such an approach, Silas argued, forces developers into a continuously incremental evolution of the kernel. As an alternative, Corey seeks to remove unnecessary sharing by letting applications explicitly declare when they intend kernel state to be shared.

Corey provides three new interfaces for creating kernel data structures: shares, address ranges, and kernel cores. However, only the first two were discussed in the presentation, each with an example. To demonstrate the use of shares, Silas showed how different file descriptors opened by different processes could result in poor performance scaling. Some access to the global file descriptor table was shown to be clearly contentious, with an order of magnitude decrease in performance. However, much of the synchronization around this table is clearly unnecessary, since processes may be using unrelated file descriptors. Corey's support for shares allows an application to explicitly define a private file descriptor to avoid the unnecessary locking. To demonstrate address ranges, Silas showed how shared memory can result in contention, even when processes are not actively writing into the shared memory space. The solution was to build a private memory space but to map in shared memory when explicitly requested by the application. This allows updates to private memory to occur without contention. Both examples were demonstrated with microbenchmarks. During the question-and-answer session Silas conceded that the workloads used in the system's evaluation targeted only the observed contention behavior, not the overall performance. He believed that overall performance would increase relatively with more cores and would likely be visible on a 32-core system.

The question-and-answer session was very active, and Silas's jovial nature drew appreciation from the audience on several occasions. Marvin Theimer of Amazon wondered whether any degree of public/private locking would prove to be scalable in a many-core architecture. He pointed out that successful grid and cluster architectures eschewed locking entirely, in favor of message passing. Silas pointed to the potential for nonglobal sharing in Corey as a possible means of scaling into a large number of cores. Alex Snoeren of UCSD pointed out that the architecture extracted the basic elements of NUMA and asked whether partitioning into sharing groups might be done automatically. Although this had been considered and may be a part of future work at MIT, Silas stressed that sharing explicitly could be a desirable quality to programmers.

- ***CuriOS: Improving Reliability through Operating System Structure***
  Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle, and Roy H. Campbell, University of Illinois at Urbana-Champaign

Francis David addressed issues of system failure in his presentation on CuriOS. The system attempts to make use of restart-based recovery to deal with errors and uses a novel separation of client state to ensure that service restart does not result in additional failures.

Francis began by showing how faults in Minix 3, L4, Chorus, and Eros could each result in lingering failures even after a successful service restart. The problem is that a service stores state on behalf of its clients. When the service fails, its state is restarted along with the service, which puts clients in an inconsistent state. More broadly, the authors identified four requirements for transparent restart: address transparency, suspension of the client for recovery, and persistence and isolation of client state.

To provide these properties, Server State Regions (SSRs) were introduced. Each SSR represents the current state of a client and exists in an isolated address space. When a request is received by a server, the SSR is temporarily mapped into the server's address space and is correspondingly unmapped for the response. In the event of a fault, the server will only have access to the process it was directly servicing. On recovery, the client data can be checked for consistency. To test the techniques, a working timer and network service were created. These were validated with fault injection on an embedded platform. Faults, in this case, are memory aborts or bit flips that result in a system crash. With SSRs, CuriOS was able to recover from more than 87% of these failures and was usually able to restart the failed service. Francis evaluated the system with microbenchmarks, showing that the overhead consisted mainly of flushing the TLB and was therefore similar to a context switch.

Francis then fielded questions from the audience. Emin Gün Sirer from Cornell took a moment to disagree with an earlier claim that type safety would fix many faults in system operation, effectively strengthening the motivation for the work. He then asked whether the fault detection relied on a component detecting its own fault. When Francis replied that the fault detection had been ported from the existing code base, Gün asked whether fault detection needed to be protected in general. To this, Francis agreed and pointed to techniques for checking in-line, but he made it clear that their focus was exclusively on recovery, not detection. Francis was also asked about developing user-mode recovery services and recovering from faults in the SRR management service itself. Both were identified as areas for future work. He also clarified a few additional points from his presentation before session chair Remzi Arpaci-Dusseau was forced to cut off the discussion in the interests of time.

- ***Redline: First Class Support for Interactivity in Commodity Operating Systems***
  Ting Yang, Tongping Liu, and Emery D. Berger, University of Massachusetts Amherst; Scott F. Kaplan, Amherst College; J. Eliot B. Moss, University of Massachusetts Amherst

Ting Yang presented Redline, an operating system that provides first-class support for interactive applications. The commonly used schedulers in popular operating systems strive for a state of fairness. This ideal is one in which each process shares resources equally at fine granularity. The best-known alternative to this is real-time scheduling, in which processes receive dedicated resources to ensure timely responses. Interactive applications sit between these two ideals. They may often be idle for long periods of time followed by bursts of activity, as driven by an external event. Arguably, this can be a more important metric than fairness, since it is capable of capturing the user's perception of responsiveness.

To illustrate the challenge of scheduling for interactive applications Ting showed how simultaneous video playback and kernel compilation results in jumpy and unresponsive video playback. The problem is one of resource management. For example, both applications rely on getting data from disk, but the queues and caches in the storage stack are not necessarily fair.

The goal of Redline is to maintain responsiveness in applications that need it. It operates by coordinating resource management to devices as well as CPU and memory. Users identify interactive applications and reserve the desired CPU, memory, and IO priority. In the case of memory management, interactive applications are vulnerable to LRU eviction because they scan memory less frequently. Worse, as their working set shrinks, they fault more frequently, making the problem a potentially degenerative case. To correct this, Redline preserves working set size by protecting the pages of interactive applications for 30 minutes. If the system has insufficient memory for this assurance, processes will be degraded to best effort. At the same time, memory access by best-effort applications are "speed-bumped" to keep them from touching memory too frequently.

To evaluate Redline, Ting showed the result of video playback under the pressure of a fork bomb, a malloc bomb, and an IO intensive workload. In each, Redline was shown to outperform Linux (measured in frames per second) by a significant margin.

Ashvin Goel, from the University of Toronto, noted that the resource specification is described differently for each type of resource and wondered why a more unified approach wasn't used. Ting acknowledged that uniformity is desirable but explained that this approach was difficult in practice. They ultimately found that different descriptions were appropriate for each resource. To simplify the specifications, default rules and inference take some of the burden off the user. Ting also addressed concerns of starvation, stating that it might be necessary given that the goal was to provide isolation for interactive applications.

MONITORING

Summarized by Olga Irzak (oirzak@cs.utoronto.ca)

- ***Network Imprecision: A New Consistency Metric for Scalable Monitoring***
  Navendu Jain, Microsoft Research; Prince Mahajan and Dmitry Kit, University of Texas at Austin; Praveen Yalagandula, HP Labs; Mike Dahlin and Yin Zhang, University of Texas at Austin

Navendu Jain began by pointing out the importance of monitoring large-scale distributed systems. A motivating example showed that best effort is not always sufficient. For an application monitoring PlanetLab, half of the nodes' reports deviated by more than 30% from the true value, and 20% of the nodes by more than 65%. These inaccuracies are due to slow nodes, slow paths, and system reconfiguration. Hence, a central challenge in monitoring large-scale systems is safeguarding accuracy despite node and network failures. Network imprecision exposes the state of the network so that applications can decide whether or not to trust the accuracy of the result. Current techniques used for scalability in monitoring systems are aggregation, arithmetic filtering, and temporal batching. However, during network or node failure, these techniques can cause failure amplification, silent failure, and blockage of many updates even for a short disruption. As a result, the authors suggest accepting that the system is unreliable and quantify system stability using the "Network Imprecision" metric.

Instability manifests itself in either missed/delayed updates or double-counted updates. To quantify the former, a lower bound on the number of nodes whose recent inputs are guaranteed to be in the result (Nreachable) versus the number of total nodes in the system (Nall) is reported. The Ndupmetric is an upper bound on the number of nodes that are double-counted. Together, these metrics are useful to expose the impact of disruptions on monitoring accuracy. Network imprecision is application-independent, inexpensive, and flexible. Applications can set a policy to improve accuracy by applying the right techniques, such as filtering inconsistent results or performing redundant or on-demand reaggregation. Filtering with network imprecision results in 80% of the reports having less than 15% error, in contrast to best effort, in which there is 65% error. Network imprecision is simple to implement but difficult to implement efficiently. Using DHT trees, which form a butterfly network, intermediate results can be reused across different trees. This reduces the load from $O(Nd)$ to $O(d\log N)$ messages per node.

Discussion began with an observation and a challenge. Observation: Presented was a very nice subset of what DB people call observation quality—add metrics that are available where things such as freshness come into account. This gives more metrics to make decisions with. Challenge: Wouldn't it be nice if I could ask a monitoring system for a particular level of accuracy or network imprecision or whatever, and could you construct me a system that gives me that level of accuracy? The approach has been to separate the measurement mechanism from the policy that sets the accuracy bounds. Applications can accept results with a given accuracy bound and throw away others. But you can't guarantee an accuracy bound because of the CAP problem. Since the problem is closely related to aggregation techniques from sensor networks, the next questioner wondered whether looking at order- and duplicate-insensitive aggregation functions would be in order. These are complementary because in principle these try to minimize the impact of disruptions on accuracy, whereas network imprecision exposes disruptions. The SM techniques provide a policy that can be applied in this framework.

- ***Lightweight, High-Resolution Monitoring for Troubleshooting Production Systems***
  Sapan Bhatia, Princeton University; Abhishek Kumar, Google Inc.; Marc E. Fiuczynski and Larry Peterson, Princeton University

Sapan Bhatia made the observation that there will always be bugs in production systems despite analysis and testing frameworks. The problem is aggravated as system complexity increases. There are easy and hard bugs in systems. Easy bugs come with an exact description and are easy to reproduce. Hard bugs are hard to characterize, hard to reproduce, hard to trace to a root cause, spatially ambiguous, temporally removed from root cause, and can be intermittent and unpredictable. The authors experienced many hard bugs in their dealings with PlanetLab, such as intermittent kernel crashes with an out-of-memory bug, unusual ping latencies, and a kernel crash every 1–7 days (or not). A medical analogy is that easy problems are single, localized injuries such as a broken foot, whereas a hard one might be a vague sense of malaise. To facilitate handling such bugs, the authors present Chopstix. Chopstix monitors low-level events (vital signs) such as scheduling, I/O, system calls, memory allocation, and cache misses. It then captures abnormal deviations in the system's behavior; these are referred to as symptoms. As logging all events is too expensive, sampling is used. Uniform sampling is biased toward high-frequency events. Instead, Chopstix uses frequency-dependent sampling from the measurement community, which uses a sketch—an approximation of the frequency distribution of a set of events. On an event trigger, one extracts a signature of the event, hashes it, and updates the sketch. If a sampling decision is made, one collects the sample and performs logging. Collecting the sample is heavyweight but is more efficient because of the principle of locality. At the end of every data-collecting epoch (60 s in Chopstix), the collected data is transferred to user space and stored as a series of summaries. This results in a rich, fine-grained data set for an overhead of 1% CPU utilization and around 50 MB per day of disk consumption on each node.

The Chopstix GUI was also presented. The GUI polls events from different nodes for the requested vital signs. These are displayed on a graph that can be drilled into to see more detail. The GUI allows the use of intuition for certain bugs (e.g., high CPU usage means a busy loop). Alternatively, use of a library of diagnosis rules to detect less obvious conditions such as low process CPU utilization with high scheduling delay and high CPU utilization per system likely means a kernel bottleneck. An example of using Chopstix to solve an elusive PlanetLab bug was then presented. The observed behavior was that some nodes would freeze every few days (or not), there was no info on console, the SSH session would stall prior to hangs, and vmstat reported high IO utilization. Chopstix showed abnormal blocking and I/O vital signs from the journaling daemon, which turned out not to be responsible for the freezes. It also showed spikes in the scheduling delay coincident with spikes in kernel CPU utilization, which pointed to the critical bug in a loop in the scheduler. They evaluated performance overhead using the lmbench microbenchmark. The slowdown is between 0% and 2.6% for getpid. Kernel compile and Apache macrobenchmarks showed almost no overhead in the benchmarks.

The probability of an error is a function of the distribution of counters in a sketch. By varying the size of the sketch, the probability of error is between 0.0001% and 0.00001%. Future work includes automating bug detection by data-mining Chopstix data and combining NetFlow and Chopstix data to diagnose network-wide behavior.

The first questioner asked whether Bhatia could compare work with DCPI from SOSP 1997. They were also monitoring with 1.2% overhead. Bhatia replied that they had a simpler notion of sampling. Their work is similar to oprofile, which this beats. Another person wondered about diagnosis rules. Is it possible (looking into the future) to capture the thinking of a systems guru looking at this output by data-mining? There are various knowledge-base systems that analyze code for bugs, but they are imperfect, so you really need human expertise to analyze rich data. Someone else pointed out that this is useful for the developer who is a kernel hacker and knows the system inside out, but how could it be extended to abstract the vital signs to a higher level and give signals of more abstract health, so that a system could say, "I'm sick, I need to see a doctor"? A lot of intelligence could be coded into rules, and the rule database could grow over time (comparable with WebMD), allowing diagnosis by less experienced people. How scalable can this be made if you want hundreds or thousands of metrics? Bhatia said that they haven't carried out these experiments. The <1% utilization is between 0.1% and 1%, so there should be more room to squeeze in other metrics. Finally, someone pointed out an issue: In the 1970s and 1980s medical diagnosis with AI was prevalent, but there was an explosion after 20 or so rules, so there was a move to other systems such as Bayesian filtering. One of the reviewers was excited about rule-based systems and wanted it cited, but it was hard to find. Maybe that should tell us something.

- ***Automating Network Application Dependency Discovery: Experiences, Limitations, and New Solutions***
  Xu Chen, University of Michigan; Ming Zhang, Microsoft Research; Z. Morley Mao, University of Michigan; Paramvir Bahl, Microsoft Research

Xu Chen highlighted the fact that enterprise networks support various business-critical applications such as VoIP and email. In a large network, there are usually thousands of applications running simultaneously, thousands of people are doing IT support, and lots of money is spent thereon. Network management is complicated since applications are very complicated and distributed across multiple components (e.g., MS Office Communicator, a VoIP/messenger app, uses DNS, Kerberos, VoIP, Director, and many back-end servers such as file and SQL). Extracting dependency information is hard. Applications are heterogeneous, in terms of functionality and deployment. The knowledge of these dependencies is distributed across layers and locations. Also, applications continuously evolve, adding new services periodically and reconfiguring/consolidating others. Currently, when there is a service outage, human knowledge and understanding of the system and its dependencies is used to troubleshoot the problem. This is expensive, error-prone, and difficult to keep up-to-date. This provides the motivation for automatically discovering dependencies for network applications.

This work introduces a new technique to discover service dependencies based on delay distributions, identifies the limitations of dependency discovery based on temporal analysis, and evaluates the technique on five dominant applications in the Microsoft enterprise. The goal is to design a generic solution for various applications using nonintrusive packet sniffing and TCP/IP header parsing. The proposed system, called Orion, identifies the time delay between dependent services, which reflects the typical processing and network delay. Orion identifies service based on IP address, port, and protocol. To make this scalable, ephemeral ports are ignored. Another problem is that dependencies exist between application messages. To address this Orion analyzes only TCP/IP headers and aggregates packets into flows. This reduces bias introduced by long flows and reduces the number of pairs. Orion needs a fair number of samples to infer dependencies. That can be overcome by aggregating across clients, servers and ports.

Orion was deployed in an MS enterprise network. It focused on extracting dependencies from MS Exchange, Office Communicator, Source Depot (such as CVS), Distributed File System, and intranet sites. Orion has no false negatives and a smaller footprint than previous work. As we see more flows, the false positives increase, but false negatives decrease. True positives eventually converge to the correct value. Orion's drawback is that it requires training, isn't applicable to P2P applications, and may miss certain kinds of interactions (e.g., periodic ones). It may include false positives. Lessons learned are that temporal analysis is limited, as it has no app-specific knowledge. Regardless, false positive can be reduced to a manageable level.

Why did filtering induce a small peak in the delay distribution in the rightmost bins? Chen said that this was an artifact of the filtering, but it doesn't affect the result. It may need more sophisticated FFT. Someone else pointed out that in a datacenter, you may have services dependent on services and so on, so a client action may go through lots of servers indirectly, so is the resolution enough to identify this without deep packet inspection? Chen said you could definitely do deep packet inspection, but temporal analysis can be made usable.

Finally, someone asked, given the prevalence of virtualization technology, services could be migrated among machines, so does the technique take that into account? Service migration is a potential application of this information, allowing dependent servers to migrate together.

WORK-IN-PROGRESS REPORTS (WIPS)

Summarized by Daniel Peek (dpeek@eecs.umich.edu)

- ***Multikernel: An Architecture for Scalable Multi-core Operating Systems***
  Simon Peter, Adrian Schupbach, Akhilesh Singhania, Andrew Baumann, and Timothy Roscoe, ETH Zurich; Paul Barham and Rebecca Isaacs, Microsoft Research, Cambridge

Locking and shared data limit scalability, so we should treat multiple cores in a computer as if they are on a network and use distributed systems ideas to get them to work together. These include partitioning, replication, and agreement protocols. For more information see www.barrelfish.org.

- ***Transcendent Memory: Re-inventing Physical Memory Management in a Virtualized Environment***
  Dan Magenheimer, Chris Mason, David McCracken, and Kurt Hackel, Oracle Corporation

Efficient memory utilization in the presence of virtual machines (VMs) was addressed. As an improvement on ballooning techniques, idle memory held by VMs can be reclaimed. VMs can access the resulting pool of memory through paravirtualized APIs.

- ***Towards Less Downtime of Commodity Operating Systems: Reboots with Virtualization Technology***
  Hiroshi Yamada and Kenji Kono, Keio University

This project explores the use of virtual machines to improve the speed of reboots caused by patching. To patch a system, first a VM clone of the currently running system is created. The clone is patched and rebooted. Then, the clone replaces the currently running system. This avoids disruption of the currently running system until the clone system is patched and rebooted.

- ***TeXen: Virtualization for HTM-aware Guest OSes***
  Christopher Rossbach, UT Austin

TeXen is the first virtual machine monitor to use hardware transactional memory to virtualize HTM-aware OSes. This combination has difficulties, such as preserving the guarantees provided by HTM hardware, and opportunities, such as moving the complexity of input and output out of the kernel.

- ***SnowFlock: Cloning VMs in the Cloud***
  H. Andres Lagar-Cavilla, University of Toronto

VMs in cloud computing need to be improved. They can take minutes to start up and require application-specific state to be transferred to them. Instead, we should be able to use a fork-like interface to quickly create many stateful VMs.

- ***CPU Scheduling for Flexible Differentiated Services in Cloud Computing***
  Gunho Leo, UC Berkeley; Byung-Gon Chun, Intel Research Berkeley; Randy H. Katz, UC Berkeley

No summary available.

- ***Toward Differentiated Services for Data Centers***
  Tung Nguyen, Anthony Cutway, and Weisong Shi, Wayne State University

Because application demands differ, developers should be able to choose VMs with varying properties, such as the number of replicas and network topology. Instead of offering generic VMs, cloud computing providers can offer several kinds of VMs, each with varying properties.

- ***TCP Incast Throughput Collapse in Internet Datacenters***
  Yanpei Chen, Junda Liu, Bin Dai, Rean Griffith, Randy H. Katz, and Scott Shenker, University of California, Berkeley

In a situation with *N* clients connected to a server through a switch, application throughput drops dramatically when all of the clients attempt to communicate simultaneously. This work explores the interaction of this communication pattern and TCP and proposes changes to TCP to improve this situation. Although simply adding more buffer space on the switch would mitigate this problem, the underlying TCP issues should really be solved by a fix to TCP.

- ***Gridmix: A Tool for Hadoop Performance Benchmarking***
  Runping Qi, Owen O'Malley, Chris Douglas, Eric Baldeschwieler, Mac Yang, and Arun C. Murthy, Yahoo! Inc.

Gridmix is a set of Hadoop benchmarks that addresses the needs of several audiences, including Hadoop developers, application developers, and cluster builders.

- ***CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems***
  Masyam Yabandeh, Niola Knezevic, Dejan Kostic, and Victor Kuncak, EPFL

Many errors in distributed systems are a result of violations of safety properties. To avoid these situations, members of a distributed system can gather snapshots of neighboring participants and do state space exploration to understand the results of actions and steer execution away from inconsistent states.

- ***Coscheduling of I/O and Computation in Active Storage Clusters***
  Saba Sehrish, Grant Mackey, and Jun Wang, University of Central Florida

This work deals with Hadoop map tasks that work with several data objects that may not be co-located. The authors are making a scheduler for map tasks that takes into account the performance difference between local and remote I/O operations.

- ***Honor: A Serializing On-Disk Writeback Buffer***
  Rick Spillane, Chaitanya Yalamanchili, Sachin Gaikwad, Manjunath Chinni, and Erez Zadok, Stony Brook University

Random writes are becoming a larger component of many I/O workloads, but for performance reasons sequential writes are preferable. This project redirects writes to a separate disk, called the sorting disk, to sequentially log writes. These logged writes can later be applied to a general file system.

- ***Zeno: Eventually Consistent Byzantine Fault Tolerance***
  Atul Singh, MPI-SWS/Rice University; Pedro Fonseca, MPI-SWS; Petr Kuznetsov, TU-Berlin/T-Labs; Rodrigo Rodrigues, MPI-SWS; Petros Maniatis, Intel Research Berkeley

Storage backends generally favor availability over consistency. This project proposes an eventually consistent byzantine-fault-tolerance algorithm to improve consistency in such storage systems.

- ***Scalable Fault Tolerance through Byzantine Locking***
  James Hendricks and Gregory R. Ganger, Carnegie Mellon University; Michael K. Reiter, University of North Carolina at Chapel Hill

To improve throughput and latency of byzantine-fault-tolerant systems, this project allows clients to use a byzantine-fault-tolerance algorithm to acquire a byzantine lock on a part of the system state and specify the order of operations.

- ***Fault Tolerance for Free***
  Taylor L. Riche and Allen Clement, The University of Texas at Austin

Multicore machines are now widely available; the difficulty is in programming fault tolerance for these systems. This project takes programs that are already constructed in a language that maps applications to multicore machines and reuses those interfaces to provide redundant execution for fault tolerance.

- ***Writing Device Drivers Considered Harmful***
  Leonid Ryzhyk and Ihor Kuz, University of New South Wales

The information needed to make device drivers is in the specification of the OS interface and the device interface. This work automatically creates drivers by composing state machines representing both specifications.

- ***CitySense: An Urban-Scale Open Wireless Sensor Testbed***
  Ian Rose, Matthew Tierney, Geoffrey Mainland, Rohan Murty, and Matt Welsh, Harvard University

This project aims to build a 100-node city-wide sensor network testbed aimed at public health studies, security, and novel distributed applications. Researchers can get an account to run experiments. See www.citysense.net for further details.

- ***WiFi-Reports: Improving Wireless Network Selection with Collaboration***
  Jeffrey Pang and Srinivasan Seshan, Carnegie Mellon University; Ben Greenstein, Intel Research Seattle; Michael Kaminsky, Intel Research Pittsburgh; Damon McCoy, University of Colorado

WiFi-Reports aggregates user-contributed information about the quality of pay-to-access wireless access points. Challenges include privacy, fraud, and estimation of packet loss regimes with distributed measurements.

- ***S3: Securing Sensitive Stuff***
  Sachin Katti and Andrey Ermolinskiy, University of California, Berkeley; Martin Casado, Stanford University; Scott Shenker, University of California, Berkeley; Hari Balakrishnan, Massachusetts Institute of Technology

This project aims to prevent high-bandwidth data theft by external attackers by enforcing policies at data egress points such as the network and USB keys. A network of hypervisors is used to track information flow at the word level. This uses hardware support for virtualization and speculation for performance.

- ***Communities as a First-class Abstraction for Information Sharing***
  Alan Mislove, MPI-SWS/Rice University; James Stewart, Krishna Gummadi, and Peter Druschel, MPI-SWS

A community is a densely connected subgraph of users in an online service (e.g., Facebook, MySpace). These communities can be leveraged to infer trust, control access to communities, and discover more relevant search results.

FILE SYSTEMS

Summarized by Vivek Lakshmanan (vivekl@cs.toronto.edu)

- ***SQCK: A Declarative File System Checker***
  Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, University of Wisconsin, Madison

Since filesystem corruption can lead to data loss, it is important to keep filesystems consistent at all times. Nonetheless, filesystems continue to be corrupted. Although techniques such as journaling have proved to be effective against the most common cause for filesystem inconsistency, system failure during metadata updates—it is not sufficient to repair filesystem corruption. Offline filesystem consistency checkers such as fsck are therefore seen as the last line of defense against data loss. However, commodity filesystem consistency checkers (e.g., e2fsck for ext2) can themselves corrupt the file system. The authors suggest that the implementation complexity of consistency checks—normally written in C—is a major contributor to the limited reliability of such tools. SQCK improves filesystem reliability by providing a SQL-based declarative language to filesystem developers to succinctly define consistency checks and repairs. The result is the ability to encode the functionality of 16 KLOC from e2fsck in 150 SQL queries.

Type-aware fault injection was used to expose a number of flaws in e2fsck. For instance, corrupting an inode's indirect pointer to point to the superblock results in an unmountable filesystem when processed by e2fsck. In addition, e2fsck does not take advantage of all the information available to it during repairs, making consistent but incorrect repairs. SQCK decouples specification of consistency checks and repair policies from the interpretation of filesystem structure, making it easy to specify intent. SQCK takes an FS image and loads all of its metadata into database tables. The specified SQL queries are run on these tables and any modifications are then serialized back to the filesystem image. Several examples of existing fsck checks and their equivalent in SQCK to illustrate the improvement were presented. When specifying filesystem repairs declaratively in SQL is not possible, a library of SQL queries can be composed together through bits of C code to produce the desired effect.

The current prototype uses a MySQL database that holds the tables in a RAM disk. The 150 queries representing e2fsck were implemented in 1100 lines of SQL. It is also easy to add new checks or repairs. Through the use of some performance optimizations, the runtime overhead on their current prototype was brought within 50% of the existing e2fsck.

A questioner asked about the size of the RAM disk for e2fsck. Gunawi explained that the database only concerns itself with metadata. For a half-full 1-TB filesystem, roughly 500 MB of storage is required for metadata. Another attendee wondered whether the comparison of the number of lines of code between e2fsck and SQCK took the complexity of the scanner-loader that interprets the filesystem image. Gunawi said that e2fsck includes 14 KLOC of scanning code, which was ignored in the comparison. When asked whether reordering SQL statements can affect correctness, the presenter said that an automated reordering of the SQL statements has not been implemented, but he found that ensuring the ordering manually was easier in SQCK. An attendee wondered whether flushing back to disk could introduce errors. Gunawi said that e2fsck itself does not atomically repair the filesystem; thus a crash during e2fsck could cause another inconsistent state. However, implementing transactional updates in SQCK would be simpler, although the current implementation does not have such support. The final issue raised was whether it wouldn't be easier to specify what the filesystem should look like than to specify checks. Gunawi explained that this was indeed the original intent and led them to explore developing their own declarative language. However, they found that SQL was a close fit and already had traction in the technical community.

- ***Transactional Flash***
  Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou, Microsoft Research, Silicon Valley

Solid state disks (SSDs) are a significant departure from traditional magnetic disks. Currently, these disks have adopted the existing thin interface with which existing magnetic disks comply. Although this allows backward compatibility, Vijayan states that this choice represents a lost opportunity in proliferating new abstractions to truly take advantage of this shift in storage technology. TxFlash is a new abstraction that allows SSDs to provide transactional semantics. TxFlash proposes transactional support in SSDs, reducing file system complexity while providing performance improvements. The existing logic in SSDs to handle write-leveling and cleaning makes them particularly well suited for transactional support.

A commit protocol is essential to provide transactional semantics. The traditional log-based commit protocol is heavily used for its relative simplicity: A separate log is maintained where data is written, following which an explicit commit record is written. Once committed, write-back can occur asynchronously. This adds a space overhead for the explicit commit record, as well as a performance penalty owing to the strict ordering required between the data and commit records. These can have a significant impact on workloads that require short and frequent transactions. TxFlash's new WriteAtomic and Abort abstractions could eliminate these problems. The WriteAtomic abstraction can inform the TxFlash device what pages need to be updated atomically. The TxFlash device links pages committed within the same transaction to form a closed loop, which can then be written in parallel. A commit is said to be successful if the updates cause a cycle. However, broken cycles are not a sufficient condition for detecting aborted transactions. To solve this ambiguity, the Simple Cyclic Commit (SCC) protocol is proposed by the authors. It ensures that if a version of a page exists, any previous version of the same page on disk must be committed, and if a transaction aborts, the affected pages and their references are erased before a newer version is retried.

TxFlash was evaluated through a simulator as well as an emulator implemented as a pseudo block device. The authors also implemented a modified version of the ext3 file system, TxExt3, which exercises the WriteAtomic interface. This approach allowed the authors to strip out roughly 50% of the ext3's journaling implementation. Benchmarks suggest that TxFlash improves performance by roughly 65% for IO-bound workloads, particularly for those with small transactions, while having a negligible impact on performance for large transactions.

An audience member asked what deployment scenarios the authors envisioned that would proliferate the use of abstractions like those proposed in TxFlash. Vijayan said that reduced complexity of the software stack makes TxFlash particularly interesting for embedded devices and specialized operating systems such as those in game consoles. Another attendee asked whether there was a way to implement transactions on existing interfaces. How about exposing a fuller view of flash including metadata to software? Vijayan believes that it is not sufficient to expose the metadata. One would either have to export all the functionality to the software or implement it in the disk. The latter seems preferable to hide the complexity from software.

- ***Avoiding File System Micromanagement with Range Writes***
Ashok Anand and Sayandeep Sen, University of Wisconsin, Madison; Andrew Krioukov, University of California, Berkeley; Florentina Popovici, Google; Aditya Akella, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, and Suman Banerjee, University of Wisconsin, Madison

The modern disk interface is represented by a linear address space; this aids usability but limits the opportunity for the software stack to make performance improvements. File systems try to improve disk I/O by ensuring spatial locality of related blocks; however, the interface only allows them a low-level command to write a block to a specific logical address—a classic example of micromanagement. The information gap between the file system and the disk results in I/O operations incurring unnecessary positioning overhead. This work proposes a new interface where the file system can specify a range of candidate blocks to the disk and allows it to choose the most appropriate block.

The interface allows the file system to specify a list of ranges to the disk, which, in return, specifies the result of the request as well as the chosen target block for the write request. One possible problem is that file systems might specify multiple write requests that have overlapping ranges. If the disk selects a block from the overlapping range, then it must make sure it masks it out of the range that it considers for the subsequent write request. This additional metadata must eventually be cleared. An ideal opportunity to do so is during write barriers.

This change in interface would require the in-disk scheduler to consider all options from among the list of ranges passed to it by the OS but select only one. A simple modification is the expand-and-cancel (ECS) approach. It expands a range write into a number of write requests. Once a candidate is selected, all other ranges are canceled from the request queue. This, however, is computationally expensive, since extensive queue reshuffling may be required. The authors present an alternative called Hierarchical Range Scheduling (HRS) in their paper.

Integration into existing file systems also poses a challenge. For instance, file systems try to improve sequentiality of writes. However, with range writes, related blocks could be spread farther apart. Moreover, there are complications because the file system is informed of the chosen destination later than normal. However, the presenter suggested that these were not significant impediments to using range writes, at least for a subset of block types. Simulation runs of range writes on write-heavy workloads such as Postmark and untar showed that the performance improved by 16%–35%.

An audience member wondered whether it was sufficient to create a faster write cache without modifying the filesystem, disk, and interface to improve write performance. The presenter replied that existing disks could benefit from their approach. Moreover, cooperative storage between the cache device and the backing store was also possible. Another audience member commented that if a high-speed write cache (e.g., a flash drive) was used, it is likely that the backing disks would be slow and inexpensive; therefore cooperation between the cache and backing disks through range writes might not be possible. Lastly, a few people wondered how range-writes would handle overwrites and if this approach wouldn't result in greater fragmentation. The presenter suggested that read and rewrite performance might suffer from fragmentation. He mentioned that they had not sufficiently evaluated this and that it was an area for future work.

PROGRAMMING LANGUAGE TECHNIQUES

Summarized by Vivek Lakshmanan (vivekl@cs.toronto.edu)

- ***Binary Translation Using Peephole Superoptimizers***
Sorav Bansal and Alex Aiken, Stanford University

Sorav Bansal explained that binary translators are used to run applications compiled for one ISA (Instruction Set Architecture) on another. Major challenges in binary translation are performance, ISA coverage, retargetability, and compatibility with the OS. This work explores using superoptimization to address the first three of these challenges associated with binary translators. Superoptimization is an approach in which the code generator does a brute-force search for optimal code. The peephole superoptimization approach presented here utilizes three modules: a harvester, which extracts canonicalized target instruction sequences from a set of training programs; an enumerator, which enumerates all possible instructions up to a certain length, which are checked for equivalence against the target sequences produced by the harvester; and, finally, a rule generator, which creates a mapping of target sequences to their equivalent optimal translations.

Using peephole superoptimization for binary translation complicates matters. First a register mapping is required between the source and destination architectures before equivalence can be verified. The register mapping may need to be changed from one code point to another and the choice of the register mapping can have a direct impact on the quality of translation. The authors used dynamic programming to attempt to reach a near-optimal solution. Their mapping accounts for translations spanning multiple instructions and simultaneously performs instruction selection and register mapping.

A static user-level translator from ELF 32-bit PPC/Linux to ELF 32-bit x86/Linux binaries was implemented. The prototype was evaluated using microbenchmarks and SPECINT 2000 against natively compiled equivalents, as well as other binary translators such as Rosetta and QEMU. The minimum performance of the prototype was 61% of native, but it exceeded native on three microbenchmarks. It is up to 12% faster than Rosetta and 1.3 to 4 times better than QEMU. Note, however, that both Rosetta and QEMU are dynamic translators, so the cost of translation is embedded in their runtimes while their current prototype is static.

One attendee asked about the pattern-matching algorithm used and whether optimizing this would improve performance. The response was that the current implementation used a very simple approach and a pattern-matching algorithm could help. There were questions raised about whether the translator could work on multithreaded applications and whether the use of synchronization constructs was tested. The presenter didn't know of any reason why their approach wouldn't work for multithreaded applications. Correctness on synchronization constructs had not been tested. Another asked how well their approach would translate to a dynamic translator. Sorav explained that the choice of a static translator was for simplicity. However, he wasn't aware of any reason why the approach couldn't work in a dynamic setting. The final query concerned how well the equivalence test handled aliased instructions. Soral claimed that the Boolean filter used by the satisfiability solver was resilient to aliasing. Details of this can be seen in their previous publication in ASPLOS.

- ***R2: An Application-Level Kernel for Record and Replay***
  Zhenyu Guo, Microsoft Research Asia; Xi Wang, Tsinghua University; Jian Tang and Xuezheng Liu, Microsoft Research Asia; Zhilei Xu, Tsinghua University; Ming Wu, Microsoft Research Asia; M. Frans Kaashoek, MIT CSAIL; Zheng Zhang, Microsoft Research Asia

Zhenyu Guo described R2 as a record and replay tool for debugging. Some bugs are hard to reproduce by simply reexecuting the program, whereas others make it hard to do comprehensive analysis without significant perturbation to the system at runtime. Previous approaches have used virtual machines to replay the application and the OS, but this is difficult to deploy. Other attempts that use a library-based approach do not support multithreading or asynchronous I/O. Instead, R2 allows developers to determine the interface at which record and replay will occur and tries to address these shortcomings.

Selecting a replayable set of functions with the goal of capturing nondeterminism requires the developer to make a cut through the call graph. Calls to functions above the cut are replayed but functions below the cut are not. R2 also establishes an additional isolation environment: The items above the call graph cut are termed to be in replay space, whereas the R2 library and the rest of the system are in system space. Since it is possible to choose cuts poorly, the authors propose two rules developers can follow: Rule 1 (isolation)—all instances of unrecorded reads and writes to a variable should be either all below or all above the interface; Rule 2 (nondeterminism)—any nondeterminism should be below the interposed interface.

A major challenge is to keep a deterministic memory footprint between record and replay. It is important to get the same memory addresses in both record and replay to ensure that different control flow isn't followed. R2 uses separate memory allocators for system and replay space. Allocation in replay space will get similar addresses, whereas allocations in system space are hidden from the application and don't need to get a deterministic address. Another issue is ensuring deterministic execution order in multithreaded applications. It is important to make sure that the ordering of execution of multithreaded applications is not changed between the original and replayed runs. To do this, R2 captures happens-before relationships through causality events that have designated annotations.

There are three categories of annotations in R2: data transfer, execution order, and optimization. The authors have annotated large parts of the Win32 API, in addition to all of MPI and SQLite APIs. R2 has been used to debug several large applications with moderate annotation effort. A recording overhead of 9% was measured on ApacheBench when run on a standard Apache Web server configuration.

One audience member asked whether there were plans for any static checking or automatic verification for choosing the interface cut to prevent developers from making mistakes. The presenter stated that cutting at library boundaries was a good rule of thumb. However, work on using compiler techniques to select this boundary is planned. Another attendee wondered whether the tool could be used in production for more complex workloads, since the overhead for microbenchmarks is helped by caching. The presenter claimed that they saw 10%–20% overhead for larger benchmarks. He also said that one problem with larger applications is that the log may become unmanageable; as a result, a form of checkpointing is being considered. A query regarding R2's ability to cope with data races was posed. The presenter admitted that R2 would not be able to handle such issues currently. Finally, an audience member asked whether developers had to reannotate new versions of libraries. The presenter said that API-level annotation was sufficient.

- ***KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs***
  Cristian Cadar, Daniel Dunbar, and Dawson Engler, Stanford University

  Winner of Jay Lepreau Award for one Best Paper

Cristian Cadar began by saying that systems programming is notorious for its complex dependencies, convoluted control flow, and liberal unsafe pointer operations. The ever-present threat of malicious users does not make matters easier. Testing such complex code is not trivial. Certain bugs are only tripped in obscure edge conditions, which may be missed even when comprehensive test cases are available. KLEE is designed to check such complex systems code. KLEE is a fully automatic symbolic execution tool that explores a large number of paths in a program and automatically generates test cases.

KLEE runs C programs on unconstrained symbolic input. As the program runs, constraints on the symbolic values are generated; these are then fed to a constraint solver that generates test cases. Though this seems conceptually simple, there are a few major challenges KLEE overcomes. First, most programs have an exponential number of execution paths they may take. As a result, smart search heuristics are essential. KLEE uses either random path selection or coverage-optimized search at any given point (chosen in round-robin fashion): The former prevents starvation of certain paths, whereas the latter tries to choose the path closest to an uncovered instruction. The second challenge relates to constraint solving. Since KLEE needs to invoke a constraint solver at every branch, the costs can easily become prohibitive. As a result, KLEE tries to eliminate irrelevant constraints before sending them to the constraint solver. In addition, results from the constraint solver for previous branches are cached and reused where possible. Finally, when a program being executed by KLEE reads from the environment (filesystem, network, etc.), all possible values for the operation should be ideally available for KLEE to explore. Similarly, when the program writes to the environment, subsequent reads should reflect that write. To handle this in the case of symbolic input to a system call, KLEE provides the ability to redirect environmental accesses to models that generate the necessary constraints based on the semantics of the system call invoked.

KLEE was run on each of the 89 standalone applications, combining to form Coreutils for 1 hour. The test cases generated were then run on the unmodified tools under a replay driver that recreates the environment for the tests to proceed. Overall, KLEE's tests produced high line coverage, with an overall average of 84%. In comparison, manual testing achieves an average of 68% coverage. KLEE was able to find 10 unique crash bugs in Coreutils, each accompanied by commands that could reproduce crashes. It has been run on hundreds of applications as well as Minix and HiStar, and it has found 56 serious bugs in total.

An audience member wondered how KLEE would work against large systems such as database servers. The presenter replied that their previous work used symbolic file systems. Something similar could be done for databases. Another question involved the location of the bottleneck for KLEE when scaling the complexity of the checked system. The authors believe the number of states is a major problem. KLEE uses COW to minimize memory footprint. An audience member noticed that bullet-proof code could be a problem (last-minute checks in the code that return incorrect values but don't let the program crash). The presenter believed that the assert framework could be of help in such situations. in response to a question about the path lengths for some of the 56 bugs found, Cristian explained that the depth varied, but in some cases hundreds of branches needed to be hit. Another audience member asked whether they had attempted to use some of the tools from the related work to see if they could catch any of the bugs as well. Cristian replied that since many of the tools were not available or not comparable to their approach, they did not perform a comparison. The last question was regarding the name KLEE and what it meant. Cristian credited Daniel Dunbar with the name and believed that perhaps it was a Dutch word, a theory swiftly put to rest by a Dutch member of the audience.

SECURITY

Summarized by Periklis Akritidis (pa280@cl.cam.ac.uk)

- ***Hardware Enforcement of Application Security Policies Using Tagged Memory***
  Nickolai Zeldovich, Massachusetts Institute of Technology; Hari Kannan, Michael Dalton, and Christos Kozyrakis, Stanford University

Hari Kannan started by observing that traditional operating systems' lack of good security abstractions forces applications to build and manage their own security mechanisms. This bloats the TCB and makes it hard to eliminate bugs. But recent work has shown that application policies can be expressed in terms of information flow restrictions and enforced in an OS kernel, and this work explored how hardware support can further facilitate this.

Unfortunately, current hardware mechanisms are too coarse-grained to protect individual kernel data structure fields, so Hari proposed using tagged memory, where each physical memory word maps to a tag and tags map to access permissions. A trusted monitor below the OS is responsible for mapping labels expressing application security policies to tags that are enforced by the hardware, so less software needs to be trusted and some level of security is maintained even with a compromised OS.

Hari presented LoStar, a prototype system based on HiStar OS and a tagged memory hardware architecture called Loki. He discussed the operation and implementation of LoStar, including optimizations such as a multi-granular tag storage scheme to reduce memory overhead and a permission cache within the processor. The prototype had a 19% logic overhead (which would be much less for a modern CPU), had negligible performance overhead, and reduced the (already small) TCB of HiStar by a factor of 2. Hari concluded his presentation with pointers to the Loki port to a Xilinx XUP that costs $300 for academics and $1500 for industry, adding that the full RTL and the ported HiStar distribution are available.

A member of the audience asked Hari to contrast Loki with Mondrian memory protection, which also offers fine granularity. In particular, could LoStar be implemented on top of Mondrian memory protection or is there some fundamental functional difference? Hari acknowledged that Loki builds on Mondrian memory protection but argued that Loki provides more functionality and that the whole system is a carefully designed ecosystem. LoStar extends application security policies all the way to the hardware, and Loki keeps the MMU outside the TCB. Mondrian memory protection, in contrast, only extends traditional access control to offer word granularity and, furthermore, depends on the correctness of the MMU. Finally, in response to another question, he clarified that a dedicated garbage collection domain described in his presentation does not have unconditional access to kernel objects.

- ***Device Driver Safety through a Reference Validation Mechanism***
  Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gün Sirer, and Fred B. Schneider, Cornell University

Patrick Reynolds argued that device drivers should not be trusted, because they have a high fault rate and are written by third parties. These issues, in combination with running in the kernel and having hardware privileges, can lead to Trojan horses, insider attacks, and faults that take down the system. He said that Cornell University is building a new operating system, the Nexus, that is based on a trusted microkernel that leverages trusted hardware to enable trustworthy applications. Nexus aims for a small and auditable TCB, but device drivers presented a great challenge because they are so untrustworthy and keep changing.

Nexus moves drivers to user space as a first step to prevent direct attacks on the kernel, but devices can also compromise the kernel integrity in many ways, such as overwhelming the kernel with interrupts or using direct memory access. Nexus addresses this by placing a filter between the driver and the device. This filter is written once for each device, is small to audit, and, given proper specifications, can defend against broader attacks.

Direct hardware operations in drivers ported to Nexus are replaced with Nexus system calls and the sequence of operations is constrained using a specification language built around state machines. Illegal transitions kill the driver and invoke a trusted reset routine that is part of the specification. Several mechanisms, including IO ports, MMIO, interrupts, and DMA, are captured in the specification language. Finally, drivers are not trusted with disabling interrupts. Instead the driver can defer its own interrupts and pause execution of its own threads. This was sufficient in practice and has the advantage of lowering interrupt latency for other unrelated drivers.

Five drivers were ported for evaluating the system. The metrics used for the evaluation were complexity-measured by driver and specification size and performance-measured by throughput and latency and by robustness to random and targeted attacks. About 1% of the lines of code changed in each driver, and the specifications were between 100 and 150 lines of code—an order of magnitude smaller than the drivers themselves. The network throughput at 1 Gbps for user-space drivers degraded slightly for sending small packets, but no penalty was measured for sending large packets or receiving packets. The interrupt latency degraded significantly (from 5 to 50 microseconds), but that did not affect usability in day-to-day use. The CPU overhead for a benchmark streaming video at 1 Mbps increased from 1% to 2%. Finally, the resistance to attacks was evaluated by probabilistically modifying driver operations as well as using a malicious driver suite. Having the drivers in user space only caught direct reads/writes, but the security policies blocked all further attacks. Patrick concluded that the system is efficient, general, and robust against attacks.

A member of the audience objected to the use of throughput as a metric for the evaluation and highlighted that CPU load was roughly doubled. Another member wondered how this system would affect more demanding devices using Firewire or USB2.0. Patrick replied that latency and correctness were not affected for USB1.1 and USB 2.0 high-speed drives, although with USB 2.0 the CPU overhead is significant, as USB 2.0 drivers require many context switches. Patrick specifically clarified that they have not observed dropped frames with their USB disk experiments, but they have not tried Firewire. Somebody asked who would write the specifications. Patrick suggested that the device driver manufacturers or a trusted third party could write these, but it is an orthogonal problem. Asked whether performance was the reason for having the reference monitor inside the kernel, Patrick argued that it has to be part of the TCB, so it might as well be in the kernel, but he believes that the overhead of having it in user space would be small. Someone else observed that the specification can be extracted either from the driver's normal behavior or from the device's specification. Patrick said that they looked at both and considered it a shortcut to allow only observed behavior, but he agreed that for implementing a security policy normal behavior is more interesting than the device's full capabilities.

People were intrigued by the possibility of permanent damage to the hardware and solicited realistic examples. Patrick described a driver asking a device to overclock itself and turn off its fans or overwrite flash memory, rendering the device unbootable. Having observed that a generic USB policy was used for USB drivers, somebody asked for a comparison with the Windows driver model that also allows user-space drivers. Patrick acknowledged that they are very similar but argued that Windows allows only user-space devices that do not perform direct IO, whereas Nexus extends this to all kinds of devices. Asked whether an IOMMU would be useful for doing lazy trapping and batching things up, much as shadow page tables are used, Patrick observed that many memory accesses are individual commands that cannot be batched like that.

The final question was about similar description languages such as Devil IDL and Sing\#, the Singularity driver constraint language. Patrick said that they drew inspiration from the languages but did not reuse them or reuse any of the specifications that were written for them. He further argued that Devil was intended for the construction of drivers rather than constraining them, and helater added that Sing\# applies only to properties that can be checked statically.

- ***Digging for Data Structures***
  Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King, University of Illinois at Urbana-Champaign

Anthony Cozzie summarized the state of the art in the antivirus industry and the challenges faced by signature checkers against code obfuscation. He argued that all programs use data structures and targeting these could defeat code obfuscation. To this end, he presented Laika, a system that can detect programs based on their data structures.

Laika works by classifying words in memory into types such as heap pointers, zero, integers, or ASCII strings, and from these it tries to find objects in memory and their classes. Importantly, in addition to field types, it can classify objects by considering what other objects their fields point to.

Anthony presented an evaluation of Laika's accuracy using seven test programs that averaged 4000 objects and 50 classes: Laika detected object classes with 75%–85% accuracy. It was straightforward to turn Laika into an anti-virus program by checking how many structures a tested program shares with a known malware sample. They tried this with three botnets, which Laika classified with 100% accuracy, whereas ClamAV, an anti-virus program, had about 80% accuracy. Anthony observed that virus detection is an arms race, and he discussed some possible attacks on the system and countermeasures. Obfuscating reads and writes would not help the virus, as it would result in no visible data structures, which is suspicious on its own. A compiler attack could shuffle the order of fields, but that would preserve sufficient information to recover the classes. A mimicry attack could use structures from a legitimate program, such as Firefox, but a successor to Laika could try to detect unused fields, or it could detect the mismatch between the program (not Firefox) and its data structures (Firefox). The approach does not work for very simple programs, but malware with some functionality would expose some structure. Finally, being a dynamic approach, it is computationally expensive.

Stefan Savage highlighted the challenges of getting a consistent memory snapshot, with malware being careful not to expose completely at once, or waiting until a particular time of day. Anthony said that they just run the malware for five minutes and that was sufficient, but he acknowledged the problem. Another attendee observed that the attacker code might try to detect whether it is in a sandbox or a VM, to which Anthony replied that the approach relies on a working sandbox. Another point raised involved the whole class of nonmalicious applications such as DRM that use obfuscation. Anthony argued that although they could not be classified by Laika, they could be signed, thus solving this problem.

Another attendee asked whether they have tried running Laika on programs written in other languages or using more exotic compilers. Anthony said that the results would depend on the language. For example, it would not work with LISP, where everything is a list, but with some other high-level languages such as Java, the classes are readily available. Somebody observed that many programs share data structures because of libraries and runtime environments, such as the C library and the JVM, and wondered how the technique would deal with such programs. Anthony argued that the approach can still separate such programs. Somebody proposed to break Laika by encrypting data structures on the heap and decrypting them on the stack before use, but Anthony classified this as another case of an encrypted memory attack where no structure would be visible and used a graphic analogy of cutting one's fingers to avoid matching a fingerprint scan, reiterating that lack of structure is suspicious by itself. Finally, somebody was concerned with having to run Laika for five minutes and wondered whether matching against several candidate viruses would take five minutes per virus signature, but Anthony clarified that one run of the scanned program and a snapshot are sufficient for checking multiple signatures.

DEALING WITH CONCURRENCY BUGS

Summarized by Chris Frost (frost@cs.ucla.edu)

- ***Finding and Reproducing Heisenbugs in Concurrent Programs***
  Madanlal Musuvathi, Shaz Qadeer, and Thomas Ball, Microsoft Research; Gerard Basler, ETH Zurich; Piramanayagam Arumuga Nainar, University of Wisconsin, Madison; Iulian Neamtiu, University of California, Riverside

Madanlal Musuvathi presented their work on making it easier to find concurrency bugs. Concurrent applications are infamously difficult to debug: Executions are highly nondeterministic, rare thread interleavings can trigger bugs, and slight program changes can radically change execution interleavings. For these reasons, concurrency bugs can cost significant developer time to find and reproduce. Madanlal et al.'s user-mode scheduler, CHESS, controls and explores nondeterminism to trigger concurrency bug magnitudes more quickly than through stress testing.

Madanlal showed the utility of CHESS through a well-received concurrency bug demo. He first ran a bank account application test suite hundreds of times; no execution triggered a concurrency bug. Importantly, over the hundreds of runs only two unique execution paths happened to be explored. He then ran the test suite in CHESS; the bug was triggered on the fourth run. Each run in CHESS explored a unique execution path.

The high-level goals of CHESS are to find only real errors and to allow all errors to be found. This latter suite goal is difficult because of the state explosion that results from all the sources of nondeterminism; practically, Madanlal et al. wanted CHESS to be able to beat stress checking.

CHESS is a user-mode scheduler; it explores execution paths in a test (1) to trigger test suite asserts and (2) for CHESS to detect deadlocks and livelocks. CHESS inserts scheduler calls at potential semantic preemption points. Several techniques are used to reduce the number of these scheduler calls to help minimize state explosion; some of these techniques can be adjusted to trade off speed with coverage. CHESS has found bugs in Microsoft's Concurrency Coordination Runtime, IE, and Windows Explorer, among others. It is available for Win32 and .NET now and will be available later for the NT kernel. They also plan to ship it as an add-on with Visual Studio. Finally, Madanlal made a plea for abstraction designers: Specify and minimize exposed nondeterminism.

One person asked whether CHESS controls random number generators and time. Madanlal responded that CHESS guarantees reproducibility for random number generation, timeouts, and the time of day, but it will not find errors that only show up on particular return values (i.e., it captures but does not explore the nondeterminism). CHESS leaves file and network inputs to the test suite.

- ***Gadara: Dynamic Deadlock Avoidance for Multithreaded Programs***
  Yin Wang, University of Michigan and Hewlett-Packard Laboratories; Terence Kelly, Hewlett-Packard Laboratories; Manjunath Kudlur, Stéphane Lafortune, and Scott Mahlke, University of Michigan

Yin Wang presented Gadara, a principled system that provably ensures that a multithreaded C program is deadlock-free, while requiring only a modest runtime overhead (e.g., 11% for OpenLDAP). Gadara avoids circular-mutex-wait deadlocks by postponing lock acquisitions, using a program binary instrumented with control logic automatically derived from the source program using discrete control theory. Gadara transforms a C program into a control flow graph, then into a Petri net (a discrete control theory model used to describe lock availability, lock operations, basic blocks, and threads), from which Gadara derives control logic that it adds to the original source. The control logic observes and controls thread interactions. Most lock operation sites can be ruled safe and thus need not be instrumented.

Yin et al. evaluated Gadara for the OpenLDAP server, BIND, and a PUBSUB benchmark. Gadara can require source function and lock annotations; the OpenLDAP server was mostly annotated in one hour (with 28 of 1,800 functions being ambiguous). Gadara found four possible sources of deadlock: two new, one known, and one false positive (resulting from Gadara not using data flow analysis). The known bug was fixed in the OpenLDAP repository, but it was later reintroduced, and then fixed again through a code redesign. With Gadara's runtime instrumentation no fix is necessary, avoiding this frequent deadlock bug fix difficulty. This makes Gadara useful for rapid development, corner cases in mature code, and end-user bug fixes.

One person asked why one would use Gadara's instrumentation instead of fixing the bug. Yin responded that developers are not always available, that fixes can be difficult to design, and that Gadara can generate false positives, necessitating code study to determine whether the bug is real. Another person asked why adding a simple lock around the instrumentation points is insufficient; this solution works in some cases, but not all (e.g., the five dining philosophers problem). In response to why the PUBSUB experiment used only two cores. Yin answered that they found, in the BIND experiments, that 2- and 16-core experiments had similar overheads. For both PUBSUB and BIND, only workloads that saturated the CPU showed any performance overhead.

- ***Deadlock Immunity: Enabling Systems to Defend Against Deadlocks***
  Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea, École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

George Candea presented the Dimmunix deadlock avoidance system. When a system deadlocks, Dimmunix remembers the control flow that has brought the program into the current state and, through lock operation instrumentation, prevents the program from reentering that state in future program runs. George had fun describing their work in terms of antibodies (Dimmunix signatures) for pathogens (deadlock bugs): You get sick from the first infection, but you are immune from then on.

When a program deadlocks, Dimmunix saves the values for each frame in the deadlocked threads' call stacks. After the program is then restarted, if a lock operation is performed with the same callstacks Dimmunix changes the lock operation ordering to avoid the deadlock. Dimmunix is implemented for C/C++, as a modified POSIX thread library, and for Java, by rewriting Java bytecode. The talk and the paper detail the performance challenges they faced and overcame. Dimmunix avoids inducing thread starvation when reordering a program's lock operations using Dimmunix's deadlock avoidance algorithm with yield edges.

George sees Dimmunix as complementary to existing deadlock reduction techniques. For example, although a browser may have no deadlocks itself, a third-party plug-in may introduce deadlocks. An end user can use Dimmunix to avoid these, without source code access and without understanding the program's internals. George stated that Dimmunix has four important properties: (1) Someone must experience the first deadlock occurrence; (2) Dimmunix cannot affect deadlock-free programs; (3) Dimmunix cannot induce nontiming execution differences; and (4) Dimmunix must be aware of all synchronization mechanisms.

They evaluated Dimmunix by reproducing user-reported deadlock bugs for a number of large systems, including MySQL, SQLite, and Limewire, among others. For each reproducible deadlock they were able to induce deadlock, when not using Dimmunix, one hundred times in a row. With Dimmunix, each bug was triggered only the first of the hundred runs.

One person asked about extending Dimmunix to provide the stack traces to the programmer for debugging; George said one of his students is working on this for very large programs. Another asked what happens when Dimmunix misbehaves, for example, preventing a valid path from executing. George responded that Dimmunix protects against this using an upper bound on yielding. Another person asked whether Dimmunix could be generalized to segmentation faults or other symptoms; George said that another student is looking at applying their techniques to resource leaks.

POSTER SESSION

Part 1 summarized by Kyle Spaans (kspaans@student.math.uwaterloo.ca)

- ***Automatic Storage Management for Personal Devices with PodBase***
  Ansley Post, Rice University/MPI-SWS; Petr Kuznetsov, Juan Navarro, and Peter Druschel, MPI-SWS

Users don't like having to do work. PodBase is a system that tries to avoid involving the users in any way with their files. It is a system for synchronizing and replicating user data on personal devices, under the covers, for durability and availability. For example, it can use the freely available storage on the various devices in a user's household to back up files in case the original is lost.

- ***Entropy: A Consolidation Manager for Clusters***
  Fabien Hermenier, Xavier Lorca, Jean-Marc Menaud, and Gilles Muller, Ecole des Mines de Nantes; Julia Lawall, DIKU

Hermenier described Entropy, a system to optimize configuration in cluster environments for VMs. It packs and migrates VMs to reduce overhead and make best use of resources. For example, it can reduce the number of migrations needed, take advantage of parallelism, and minimize the number of nodes required.

- ***Writing Device Drivers Considered Harmful***
  Leonid Ryzhyk and Ihor Kuz, NICTA, University of New South Wales

Ryshyk said that device drivers are hard to write correctly and can compromise an entire system. This research aims to automate the process by taking a formal device driver specification along with the OS API and composing them into a driver program. The only effort necessary is to translate the device specification into a formalized finite-state machine. Ideally, this could be generated from the VHDL code of the device itself. To avoid state explosion in more complex drivers, symbolic interpretation is used.

- ***NOVA OS Virtualization Architecture***
  Udo Steinberg and Bernhard Kauer, TU Dresden

A very small microhypervisor that can run legacy OSes, NOVA, focuses on being the small trusted base (~30 KLOC) that runs in privileged mode on the hardware. It runs the maximum possible number of services in user space, monitors included, so that compromises cannot spread among VMs.

- ***CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems***
  Maysam Yabandeh, Nikola Knezevic, Dejan Kostic , and Viktor Kuncak, EPFL

CrystalBall makes it easier to find errors in distributed systems and increases their resilience. It will predict inconsistencies in live systems and then steer execution to avoid the predicted errors. It's a form of model checking and allows live distributed debugging with low overhead.

- ***Operating System Transactions***
  Donald E. Porter, Indrajit Roy, and Emmett Witchel, The University of Texas at Austin

Secure sandboxing is hard. Sandboxed systems can still be vulnerable to timing attacks. OST's approach is to atomicize and isolate access to system calls using a simple API to give more transactional behavior. It is implemented with lazy version management and eager conflict resolution.

- ***The Network Early Warning System: Crowd Sourcing Network Anomaly Detection***
  David Choffnes and Fabian E. Bustamante, Northwestern University

NEWS uses distributed clients to detect anomalies in network performance and can be useful as another tool in a network administrator's toolbox. As usual, it is a tradeoff between coverage and overhead, but network overhead is kept minimal by taking advantage of the fact that most P2P applications already implicitly monitor the network anyway. NEWS is implemented as a Vuze (BitTorrent client) plug-in, with a central reporting Web interface useful for administrators.

Part 2 summarized by Kathryn McBride (katymcbride@yahoo.com)

- ***File System Virtual Appliances: Third-party File System Implementations without the Pain***
  Michael Abd-El-Malek, Matthew Wachs, James Cipar, Elie Krevat, and Gregory R. Ganger, Carnegie Mellon University; Garth A. Gibson, Carnegie Mellon University/Panasas, Inc.; Michael K. Reiterz, University of North Carolina at Chapel Hill

No summary available.

- ***The Barrelfish Multikernel Operating System***
  Andrew Baumanny, Simon Peter, Jan Rellermeyer, Adrian Schüpbach, Akhilesh Singhania, and Timothy Roscoe, ETH Zurich; Paul Barham and Rebecca Isaacs, Microsoft Research, Cambridge

Barrelfish is a new operating system that is being built from scratch. Barrelfish uses a rapidly growing number of cores, which leads to a scalability challenge. Barrelfish gives the user the ability to manage and exploit heterogeneous hardware and run a dynamic set of general-purpose applications all at the same time. The researchers at ETH Zurich and Microsoft Research, Cambridge, are exploring how to structure an operating system for future multi- and many-core systems with Barrelfish.

- ***Dumbo: Realistically Simulating MapReduce for Performance Analysis***
  Guanying Wang and Ali R. Butt, Virginia Tech; Prashant Pandey and Karan Gupta, IBM Almaden Research

Dumbo provides a good model for capturing complex MapReduce interactions and predicts the performance of test clusters. Dumbo aids in designing emerging clusters for supporting MapReduce. Dumbo takes metadata, job descriptions, and cluster topology to the ns-2 server, where it is traced.

- ***Aggressive Server Consolidation through Pageable Virtual Machines***
  Anton Burtsev, Mike Hibler, and Jay Lepreau, University of Utah

No summary available.

- ***Scalable Fault Tolerance through Byzantine Locking***
  James Hendricks and Gregory R. Ganger, Carnegie Mellon University; Michael K. Reiter, University of North Carolina at Chapel Hill

No summary available.

- ***Eyo: An Application-Oriented Personal Data Synchronizer***
  Jacob Strauss and Chris Lesniewski-Laas, MIT; Bryan Ford, MPI-SWS; Robert Morris and Frans Kaashoek, MIT

Eyo was developed out of the interference challenges on handheld devices. All types of media are produced and consumed everywhere on laptops, phones, and MP3 players. Eyo is used to transfer files seamlessly to the right place at all times. Central servers can limit usability. There is currently no offline client synchronization. Partial replicas have vast differences in storage capacity (e.g., a phone versus a laptop). There are also bandwidth limitations. Eyo is an attempt to synchronize personal data using an application-orientated approach.

- ***ProtoGENI: A Network for Next-Generation Internet Research***
  Robert Ricci, Jay Lepreau, Leigh Stoller, Mike Hibler, and David Johnson, University of Utah

No summary available.

Part 3 summarized by John McCullough (jmccullo@cs.ucsd.edu)

- ***NetQuery: A Universal Channel for Reasoning about Network Properties***
  Alan Shieh, Oliver Kennedy, and Emin Gün Sirer, Cornell University

Information about networks and their endpoints is scarce. If a service wanted to restrict access to protected networks or if a client wanted to connect to an ISP with better provisioning or DDoS protection, they would be hard pressed to find the information on their own. NetQuery uses tuplespaces to store such useful information. The tuplespace abstraction provides for filtering and modification triggers, enabling the rapid dissemination of reputable network information.

- ***Trapper Keeper: Using Virtualization to Add Type-Awareness to File Systems***
  Daniel Peek and Jason Flinn, University of Michigan

Collecting file metadata requires intimate knowledge of the file formats. It is straightforward to read metadata from common file formats with well-known parsers. However, there are thousands of obscure file formats that don't have readily accessible parsers; it is impractical to program parsers for all of them. Trapper Keeper leverages the parsers in applications by loading the files and extracting the information from the accessibility metadata in the GUI. Manipulating a GUI for each individual file in the file system is problematic. However, using virtualization, the application can be trapped and snapshotted when it is about to open a file. Thus, the snapshot can be invoked with a file of interest. Trapper Keeper can then use this technique to extract metadata from all of the files in your file system.

- ***Gridmix: A Tool for Hadoop Performance Benchmarking***
  Runping Qi, Owen O'Malley, Chris Douglas, Eric Baldeschwieler, Mac Yang, and Arun C. Murthy, Yahoo! Inc.

As Hadoop is developed, it is important to have a set of applications that exercise the code base. Gridmix is an open set of applications useful for benchmarking, performance engineering, regression testing, cluster validation, and configuration evaluation. The current application set has been critical in the recent performance enhancements in Hadoop. Gridmix is publicly available in the Hadoop source repository.

- ***Performance Evaluation of an Updatable Authenticated Dictionary for P2P***
  Arthur Walton and Eric Freudenthal, University of Texas at El Paso

An authenticated dictionary provides key-value pairs that are certified by an authority. Such a dictionary could be used to maintain blacklists for DHT membership on end-user machines. Fern is a scalable dictionary built on Chard that uses a tree to hierarchically distribute the potentially rapidly changing data such that the data is cacheable. The values can be validated by tracing a path to the root authority. Because the validation step has latency proportional to the height of the tree, it is desirable to keep the tree as short as possible. This work explores how the branching factor of the tree can reduce the height and how per-node load is affected as a result.

- ***Miser: A Workload Decomposition Based Disk Scheduler***
  Lanyue Lu and Peter Varman, Rice University

Ensuring high-level quality of service for all disk requests necessitates a significant degree of overprovisioning. However, the portion of the requests that necessitate this overprovisioning can be less than 1% of a financial transaction workload. Relaxing the low-latency QoS requirements for a fraction of the requests greatly reduces the provisioning requirements. The differentiated service is provided by multiple queues. The priority requests are serviced from one queue and the other requests are serviced in a best-effort manner using the slack of the priority queue.

VARIOUS GOOD THINGS

Summarized by Ann Kilzer (akilzer@gmail.com)

- ***Difference Engine: Harnessing Memory Redundancy in Virtual Machines***
  Diwaker Gupta, University of California, San Diego; Sangmin Lee, University of Texas at Austin; Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat, University of California, San Diego

  Winner of Jay Lepreau Award for one Best Paper

Amin Vahdat presented Difference Engine, which was awarded Jay Lepreau Best Paper. The motivation for the work is the increasing trend toward server virtualization in the data center to support consolidation and, ultimately, cost reduction. Some hurdles to consolidation include memory limits on virtual machines and bursty CPU utilization. The work is based on two concepts: first, memory-saving opportunities extend beyond full-page sharing and, second, page faults to memory cost less than page faults to disk.

Contributions of the Difference Engine include comprehensive memory management for Xen, efficient memory management policies, and detailed performance evaluation. Difference Engine finds opportunities for memory savings through identical page sharing, page patching, and compression of pages unlikely to be accessed in the near future. Three challenges here are (1) choosing appropriate pages for sharing, patching, and compression, (2) identifying similar pages, and (3) memory overcommitment. To combat the first challenge, the system uses a global clock to see which pages have been recently modified or accessed. For the second, Difference Engine operates by keeping two hash tables, one for sharing and one for similarity. On identical hashes into the sharing table, Difference Engine verifies byte-by-byte equality before enabling page sharing. To identify similar pages with low overhead, the system hashes over subpage chunks. To deal with memory overcommitment, the system implements demand paging in the VMM.

The Difference Engine is built on top of Xen 3.0.4, and the source code is publicly available at http://resolute.ucsd.edu/hg/difference-engine. For evaluation, the authors used micro-benchmarks of the cost of individual operations. They measure memory savings and performance over 10-minute intervals after stabilization on homogenous and heterogeneous workloads. The results show a memory savings of up to 90% in homogenous workloads, gained primarily from page sharing. Heterogeneous workloads saw a memory savings of up to 65%. Performance overhead was less than 7% compared to the baseline with no Difference Engine.

One audience member inquired whether the complexity of compression outweighed the 5% benefit. Vahdat noted that compression was run after page sharing and patching and has limited complexity. Switching the order of memory saving techniques could contribute greater savings to compression. Another audience member asked about the size of the reserve of free pages kept to resolve page faults. Although Difference Engine's current implementation uses a fixed number, Vahdat noted that it would be straightforward to make this value a percentage. There was a follow-up question regarding server node paging. Vahdat explained that some amount of paging occurred during startup, but there was very little paging during the performance evaluations. One questioner asked about the benefits of only using demand paging in the hypervisor and identical page sharing, to which Vahdat answered that memory could be reduced by an additional factor of 1.6 to 2.6 when additionally employing patching and compression for heterogeneous workloads.

[See p. 24 for an article on Difference Engine by the authors of this paper.—Ed.]

- ***Quanto: Tracking Energy in Networked Embedded Systems***
  Rodrigo Fonseca, University of California, Berkeley, and Yahoo! Research; Prabal Dutta, University of California, Berkeley; Philip Levis, Stanford University; Ion Stoica, University of California, Berkeley

Prabal Dutta presented Quanto, a system for tracking energy in sensor networks, where it is the defining constraint. Sensor nodes ("motes") spend most of their time sleeping and often run on limited energy supplies, such as AA batteries. These nodes consume energy in very short bursts and they display orders of magnitude difference in current draw between active and sleep states. Three basic challenges in energy tracking include metering energy usage, breaking down energy usage by device, and tracking causally connected activities both within a node and across the network.

To measure energy usage, Quanto uses the iCount energy meter. To break down energy usage, device drivers are instrumented to expose power states, and Quanto uses this information, along with knowledge of the aggregate system energy usage, to estimate power breakdowns by device using regression. For activity tracking, Quanto labels executions, and these labels are propagated throughout the system and its device drivers, and even onto other motes. The labels help identify the origin and execution path as belonging to a particular activity.

Quanto is built on top of TinyOS, and it only needs 12 bytes per activity or energy sample. In their RAM-based logger, Quanto keeps a buffer of 800 samples. The authors evaluate Quanto regression by testing 48 seconds of the Blink application and comparing the results with ground truth obtained using an oscilloscope. The results show that, although Quanto takes a large portion of active CPU time (71%), the denominator is very small, since actual CPU active time is 0.12%, and most of the time the sensor is sleeping. Additionally, Quanto is able to monitor itself.

Quanto is very useful for measuring energy use and CPU time for sensor network applications. Dutta provided a monitoring example where Quanto was used to evaluate the cost of false alarms in low-power listening. Additionally, the application Bounce, which plays ping-pong with network packets, demonstrates Quanto's ability to track an activity propagating from node to node. Some ideas for future work include scaling Quanto from initial tests with just two nodes to a large-scale testbed of 1000 nodes, examining the energy cost of various network protocols, and developing systems for energy management.

After the presentation, an audience member observed that Blink and Bounce were simple applications and asked how Quanto scaled to larger programs. Dutta noted that the full profile logging generated quite a bit of data, even for these small programs, and complex problems were difficult to measure in the current version of Quanto, but that these issues had been addressed in subsequent work. Another person asked about analyzing energy online rather than offline. Dutta suggested keeping counters instead of profile data and calculating periodic regressions to allow online usage of Quanto. Noting the challenge of getting device drivers to model device state, one audience member asked Dutta what sorts of hardware modifications would be useful to directly extract state information. Dutta replied that it would be great to get power state information directly from the hardware, but this might be infeasible in some complex subsystems (e.g., hard disks).

- ***Leveraging Legacy Code to Deploy Desktop Applications on the Web***
  John R. Douceur, Jeremy Elson, Jon Howell, and Jacob R. Lorch, Microsoft Research

Jon Howell presented Xax, a system for running legacy code securely within Web browsers. He began by observing the popularity of Web 2.0 applications, which allow location independence, OS independence, and isolation to browser windows. Howell also observed that decades of knowledge and hard work remained in legacy C/C++ code. It would be nice to still use this code, but rewriting it or maintaining multiple code bases is often a significant obstacle. Running the code on a remote server is not a viable solution, because it would take too long. Running the code in a client is not OS independent and presents security concerns.

Xax solves the problem of running legacy code on a client by creating a Xax container. The container, called the "picoprocess," is an OS process, which provides MMU isolation, and the system call interface is filtered through a Xax monitor. Services are provided through the browser. Additionally, Xax has a platform abstraction layer to allow OS independence. It requires light code modification. Benefits of Xax include native CPU performance, legacy support, and security via isolation.

To show that Xax will work, the authors built many demos. Howell demonstrated a clock, an openGL example, and an implementation of Ghostscript. The "lightweight code modification" involved changes such as removing static dependencies, rejecting unnecessary system calls, and making I/O operate through the browser. The authors evaluated Xax by using it to support 21 libraries and 3.7 million lines of code. In conclusion, Howell summarized Xax as a secure, fast, and portable interface for running legacy code in a Web browser.

The first question regarded limitations on Xax. Howell said that the openGL example was limited because it relied on compressed PNGs for display. If Xax could get a low-level interface to the browser, it could contain its own rendering engine. Howell noted that Xax just needed the right low-level interface to provide greater functionality. Another audience member asked about the 3.7 million lines of code, and how much of it was actually being run by Xax. Howell noted that most of this code was being used, and that "shims" (parts specific to the X server or OS) were cut out.

One person asked about security and limiting resource usage. Howell noted that resource constraints would be easy to add to Xax. Another attendee asked about performance and the amount of data transferred over the Web. He also inquired whether dial-up users would be able to run Xax. Howell noted that they didn't worry about these issues when building the first version of Xax, because Web caching and other software engineering tools could be added later to make Xax more efficient.

[See p. 32 for an article on Xax by the authors of this paper.]

WIDE-AREA DISTRIBUTED SYSTEMS

Summarized by Roxana Geambasu (roxana@cs.washington.edu)

- ***FlightPath: Obedience vs. Choice in Cooperative Services***
  Harry C. Li and Allen Clement, University of Texas at Austin; Mirco Marchetti, University of Modena and Reggio Emilia; Manos Kapritsos, Luke Robison, Lorenzo Alvisi, and Mike Dahlin, University of Texas at Austin

Harry Li presented FlightPath, a peer-to-peer system for media streaming applications that is able to maintain low jitter in spite of Byzantine or selfish peers. This work is motivated by the observation that most of today's cooperative systems lack rigorously defined incentives, which leaves room for exploits and free-riders. The author made reference to their previous work, which used Nash equilibria from gaming theory to provide provable incentives for rational users not to deviate from the protocol. That work, as well as other related works, however, sacrificed flexibility and performance for correctness. In this work, Harry and his co-authors aimed at approximating Nash equilibria to achieve both formal incentives and efficiency.

More specifically, they propose an epsilon-Nash equilibrium scheme, in which rational peers may only gain a limited advantage (< an epsilon) from deviating from the protocol. This provides nodes with some freedom in choosing peers, which in turn allows them to steer away from overloaded peers and avoid departed peers. The author stressed that it is this flexibility that enables some of the properties of FlightPath: churn resilience, byzantine and rational peer tolerance, and high-quality streaming.

A member of the audience asked about FlightPath's resilience to collusion attacks. The author answered that they had considered collusion for all of the results reported for malicious attacks. Another member of the audience pointed out that a previous study had shown that most free-riders accounted for little bandwidth in a collaborative system. He wondered to what degree eliminating those small-resource free-riders would improve overall performance in a real collaborative system.

- ***Mencius: Building Efficient Replicated State Machines for WANs***
  Yanhua Mao, University of California, San Diego; Flavio P. Junqueira, Yahoo! Research Barcelona; Keith Marzullo, University of California, San Diego

Yanhua Mao presented a Paxos-based replication protocol specifically designed for WAN operation. The author envisions this protocol to be useful in cross-datacenter geographical replication. The author explained why current Paxos protocols (Paxos and Fast Paxos) perform poorly on WANs. On one hand, Paxos maintains a single leader and thus achieves poor latency for operations issued at nonleader replicas. On the other hand, Fast Paxos achieves good latency by allowing all replicas to behave as leaders, but it suffers from collisions, which lead to poor throughput.

The proposed system, Mencius, aims to take the best of both worlds. Very briefly, their approach consists of two mechanisms: rotating leader and simple consensus. The former allows the leader function to be assumed by the servers in a round-robin fashion, which means equal latencies and high throughput. The latter mechanism allows servers with low client load to skip their turn in Paxos rounds efficiently.

The authors evaluated the system by comparing Mencius's throughput and latency against Paxos's. A member of the audience asked the speaker to clarify a discrepancy in one of the graphs, which showed Mencius's throughput degrade gracefully after a crash, whereas Paxos's throughput was at an all-time low value before and after recovery. The author responded that this effect was due to Mencius's ability to use all of the servers' bandwidth, whereas Paxos was bottlenecked by the single leader's bandwidth.

Workshop on Supporting Diversity in Systems Research (Diversity '08)

December 7, 2008
San Diego, CA

Summarized by Ann Kilzer (akilzer@cs.utexas.edu)

- ***Succeeding in Grad School and Beyond***
  Alexandra (Sasha) Fedorova, Simon Fraser University; Claris Castillo, IBM Research; James Mickens, Microsoft Research; Hakim Weatherspoon, Cornell University

Alexandra Fedorova advised students to work towards an ideal CV, looking at CVs of recently hired professors for ideas. A good CV has publications in top conferences or journals, and quality and impact outweigh quantity. Fedorova also encouraged students to imagine the final product of research and to write as much of the paper as possible before building anything. Writing helps thinking, and this approach helps researchers develop methodology, review background material, and find gaps in their approach. Her final advice was to be ready for adjustment—research can be risky and may not turn out the way one intends.

Hakim Weatherspoon explained that his path had been filled with sharp turns, playing football as an undergrad, getting married in graduate school, and raising children. A postdoctorate inspired him to pursue an academic career. Hakim noted that being in graduate school is very different from being an undergrad. Grades matter less, but one is expected to become an expert in his or her field and learn from a variety of sources. He emphasized the importance of collaborations, noting this could be a challenge for underrepresented students. Hakim observed that "everyone has an agenda." Finally, he told students to "own their own career"—we are each responsible for our own success.

The section ended with James Mickens' presentation, in which he stressed that students should not fear adversarial growth—a lot can be learned from bad reviews. He encouraged students to network at conferences and not just associate with underrepresented colleagues. Networking can help lead to internships, teaching, or collaborative research. Regarding the thesis, Mickens noted that grad school was about producing science, and that students shouldn't let the thesis trip them up. Mickens ended with an assortment of random systems advice, which included learning a scripting language, not fearing math, looking for interesting problems outside of computer science, and interning in industry.

In the Q&A session, a student observed that international students have different views on authority and asked how to reconcile this when working with an advisor. Hakim recalled his own experience with his advisor, who told him that he could not graduate until Hakim confronted him as an equal and voiced his disagreement. Every advisor has his own agenda and wants students to further that; however, students must also consider what is best for their career. Hakim stressed compromise in the advisor-student relationship, noting that the student should learn to act as a colleague. Finally, he warned students not to focus too much on being a member of a minority, because that can lead to mistakes.

- ***Technical Talks***
  Andrea Arpaci-Dusseau, University of Wisconsin—Madison; Helen Wang, Microsoft Research

Andrea Arpaci-Dusseau explained her work on gray box systems, semantically smart disks, and IRON filesystems. She advised students to keep their eyes open for new observations and unique approaches. What is good research? According to Arpaci-Dusseau, research addresses problems general to many systems. Good research begins when one initially doesn't have the terminology to describe what one is thinking about.

Helen Wang presented her work on Web browser security. Wang showed the evolution leading to browsers as a multi-principal OS. Her research seeks to create a better security model, with multi-principal protection and communication abstractions in the browser. As for future research, Wang seeks to build a browser as an OS, enable browser support for robust Web service building, analyze Web service security, and investigate usability and security with the mobile Web.

- ***Career Paths in Systems Research***
  Bianca Schroeder, University of Toronto; Ramón Cáceres, AT&T Labs—Research; Jeanna Matthews, Clarkson University and VMware

Bianca Schroeder contrasted work in academia and research labs. Academic responsibilities include working with graduate students, teaching classes, applying for grants, and traveling to give talks. There is lots of freedom in the research, as well as variety in daily activities. Professors work closely with students, acting as teachers and mentors. Typical requirements of industry researchers include working with co-workers and interns. The research often has less freedom, as there is a focus on products. Industry researchers don't have to write grants but must sell their ideas internally.

How should one decide which route to take? Schroeder suggested trying out internships, teaching, and writing grants during one's graduate career. She advised improving one's name recognition by giving talks, attending conferences, or doing internships. Advisors can help set up talks at other schools. Schroeder ended with advice about selling oneself, noting the importance of strong writing and speaking skills.

Ramón Cáceres shared his challenges with self-doubt, noting that it is important to seek advice and support. He found strength in things he was certain about. He found satisfaction in developing or redesigning things that real people could use. Regarding diversity, Cáceres stressed that one's differences add value to the field. Diversity isn't just about fairness, but also about providing perspective from underrepresented user communities. Cáceres also contrasted work for research labs and startups. In research labs, one has the freedom to pursue multiple areas of interest. In startups, research is more likely to affect actual products. He ended by advising students to have confidence, learn from criticism and move on, and seek second opinions.

Jeanna Matthews described the challenges of working at a small university. As a graduate student at Berkeley, she grew accustomed to working in a large team. She taught briefly at Cornell and recalled working with well-prepared Ph.D. students, teaching one course per semester, working with other systems professors, and having access to teaching and administrative support. Matthews contrasted this with her current position at Clarkson University. Now she teaches two courses per semester and works with undergraduates. She has found it useful to "build a pipeline" so that every student learns and teaches other students. Matthews spends a lot of time mentoring and teaching, which leaves less time for research. She finds her work at Clarkson very rewarding, and she advises anyone who enjoys teaching and working closely with students to consider a position at a small university.

- ***Making the Best of an Internship in Systems***
  Lin Tan, University of Illinois at Urbana-Champaign; Dilma Da Silva, IBM Research

Lin Tan, who interned at IBM Watson and Microsoft Research, recommended finding internships by using advisors' connections, asking colleagues for advice, visiting career fairs, and searching online. Before the internship, Tan advised asking for a reading list. Because internships are fairly short, it's a good idea to talk with one's mentor ahead of time. Tan also emphasized setting expectations and working toward goals.

Dilma De Silva described benefits of internships, including honing skills, gathering information for one's thesis, and broadening one's research experience. For successful interviews, do homework, be able to discuss general ideas as well as specifics, and develop an "elevator speech" to summarize one's research. De Silva also emphasized interviewing the interviewer by asking questions about the position. Sometimes internship decisions have nothing to do with performance; rather, the internship may simply be the wrong fit. All interviews should be viewed as practice.

At the internship, students should track their progress, understand expectations, and find a mentor outside their group. Internships are short, so it's important to make plans and adjust them as necessary. Finally, De Silva noted that it's better to seek internships from different companies rather than returning to the same position in the future.

In the Q&A session, a commenter asked whether software development or research internships are better. De Silva noted the importance of asking lots of questions during the interview. Both experiences can be valuable, but it's important to be sure the internship is what you want.

OPEN DISCUSSION

Life as a System Researcher: Challenges and Opportunities

De Silva shared her experiences at the 2008 Grace Hopper Conference, including a panel discussion on the "imposter syndrome": underestimating oneself, doubting one's qualifications, or believing that everyone else is working harder and faster. The panelists listed tips for overcoming self-doubt. It's important to believe in oneself and to remember past successes rather than dwelling on failures. They advised speaking up, finding support, and faking confidence when necessary. Most importantly, we are responsible for making ourselves feel like impostors—we create our own experience.

An open discussion followed,. Fedorova posed a question on the work-life balance in graduate school. One commenter shared her experience of raising a child while in graduate school: "How do you manage? You just do. . . . When it comes down to it there are some basic things in life you can't put aside." Other participants shared stories of raising children while in graduate school. Regarding personal relationships, one commenter noted the bursty nature of research and the importance of letting friends and family know about work schedules.

Workshop on Power Aware Computing and Systems (HotPower '08)

December 7, 2008
San Diego, CA

Summarized by Alva L. Couch (alva@usenix.org) and Kishore Kumar (kishoreguptaos@gmail.com)

HotPower '08 depicts a very different approach to computing from that to which the average USENIX member may be accustomed. In a power-centric view of computing, units of measurement are translated into units representing power requirements. "Execution times" are converted into their corresponding power requirements, measured in watts. "Execution cycles" are converted into their corresponding energy requirements, measured in nano-joules. Power is (of course) energy over time: One joule is one watt/second. This energy-aware view of computing—while quite enlightening—takes some getting used to.

The goal of energy-aware computing is not just to make algorithms run as fast as possible, but also to minimize energy requirements for computation, by treating energy as a constrained resource like memory or disk. From a power/energy point of view, a computing system (or ensemble of systems) is "energy proportional" if the amount of energy consumed by the system is proportional to the amount of computational work completed by the system. True energy proportionality is impossible because of the baseline energy cost of keeping systems running even when idle, but one can come close to energy proportionality by powering up servers and/or subsystems only when needed and keeping them powered down (or, perhaps, running at a slower speed or power level) otherwise.

One thing that makes energy-aware computing challenging is that there is a straightforward inverse relationship between energy requirements and execution time, which often requires making a time/energy tradeoff. It is acceptable for some tasks to take longer times so that they can in turn require very little energy to accomplish (e.g., in embedded sensor systems that harvest power from RFID readers). In other cases, for time-critical tasks it is appropriate to balance task completion delay against energy requirements.

Similarly (but perhaps less obviously), there is also an inverse relationship between energy requirements and reliability. Reliability is usually implemented through hardware redundancy, and redundancy means in turn more power consumption. This redundancy can take subtle forms, such as whether a disk is powered up or its data and changes are cached in volatile memory instead.

HotPower is a gathering point for a diverse community of many kinds of researchers, ranging from software experts concentrating on algorithms for reducing power consumption to hardware designers and testers studying the effects of hardware design choices. This community has in a very short time developed its own acronyms and specialized language which can be difficult for a newcomer to grasp. For example, DVFS stands for Dynamic Voltage/Frequency Scaling, which represents the ability to run a CPU or subsystem at several different speeds and/or voltages with varying power requirements. Required background for understanding the papers includes the functional relationships among computing, power consumption, and cooling, as well as the basics of energy transfer including, for example, the relationship between the energy in a capacitor and the observed voltage difference between its contacts. [Editor's note: Rudi van Drunen's article in this issue discusses power in electrical terms.]

SCHEDULING AND CONTROL

- ***Memory-aware Scheduling for Energy Efficiency on Multicore Processors***
  Andreas Merkel and Frank Bellosa, University of Karlsruhe

Andreas Merkel and Frank Bellosa presented an energy-efficient co-scheduling algorithm to avoid memory contention problems in multi-core systems. Memory access power requirements depend upon processor architecture, including whether a set of processor cores shares one L2 cache. One approach to avoiding contention is to schedule tasks with different characteristics (memory-bound and CPU-bound) on each core. Another approach, called "sorted scheduling," is to reorder program blocks for processes so that only one memory-intensive block is scheduled at a time. Scheduling algorithms were tested by comparing their performance to DVFS with SPEC CPU2006 on Linux. DVFS performed better only for memory-bound tasks.

- ***Delivering Energy Proportionality with Non Energy-Proportional Systems—Optimizing the Ensemble***
  Niraj Tolia, Zhikui Wang, Manish Marwah, Cullen Bash, Parthasarathy Ranganathan, and Xiaoyun Zhu, HP Labs, Palo Alto

Niraj Tolia et al. showed that it is possible to use optimized techniques to approximate energy-proportional behavior at ensemble level. An "ensemble" is a logical collection of servers and could range from a rack-mount enclosure of blades to an entire datacenter. One can approach energy proportionality by using a virtual machine migration controller that powers machines up or down, in addition to dynamic voltage and frequency scaling in response to demand changes. A power- and workload-aware cooling controller optimizes the efficiency of cooling equipment such as server fans. A case study examines the balance between server power and cooling power and compares several energy-saving approaches, including no DVFS, DVFS alone, and DVFS with simulated annealing for service consolidation. The last approach shows significant improvement over the former two, with some counterintuitive results, including that the cooling effect from a fan is not a linear function of power input to the fan; for optimal efficiency, one must run the fan at about 30% of its peak load.

MODELING

- ***A Comparison of High-Level Full-System Power Models***
  Suzanne Rivoire, Sonoma State University; Parthasarathy Ranganathan, Hewlett-Packard Labs; Christos Kozyrakis, Stanford University

Suzanne et al. used a common infrastructure to evaluate high-level full-system power models for a wide range of workloads and machines. The machines (8-core Xeon server, a mobile file server, etc.) that they used span three different processor families: Xeon, Itanium, and Turion. Several models were compared, including a linear model based on CPU utilization, a linear model based on CPU and disk utilization, and a power-law model based on CPU utilization. To evaluate the models, they used SPEC CPU, SPEC JBB, and also memory-stress and IO-intensive benchmarks. The results of these tests illustrate tradeoffs between simplicity and accuracy, as well as the limitations of each type of model. Performance-counter-based power models give more accurate results compared with other types of power models, though these are processor-specific and thus nonportable. Counterintuitively, using a parameter in a model that is not utilized (e.g., disk in a memory-intensive application) leads to overprediction and error.

- ***Run-time Energy Consumption Estimation Based on Workload in Server Systems***
  Adam Lewis, Soumik Ghosh, and N.-F. Tzeng, University of Louisiana

Adam Lewis et al. showed statistical methods to develop system-wide energy models for servers. They developed a linear regression model based on DC current utilization, L2 cache misses, disk transactions, and ambient and die (CPU) temperatures. When evaluated with the SPEC CPU2006 benchmark programs, their model exhibited prediction errors between 2% and 3.5%. This technique shows promise, but the audience questioned whether this would apply as accurately to uncontrolled, real-world loads. Results suggest that additional performance data—beyond the performance counters that are provided by a typical processor—are needed to get a more accurate prediction of system-wide energy consumption.

POWER IN EMBEDDED

- ***Getting Things Done on Computational RFIDs with Energy-Aware Checkpointing and Voltage-Aware Scheduling***
  Benjamin Ransford, Shane Clark, Mastooreh Salajegheh, and Kevin Fu, University of Massachusetts Amherst

A *computational RFID unit* (CRFID) is a computational unit with no battery that utilizes power harvesting from RFID readers to accomplish computational tasks. CRFIDs utilize extremely low-power hardware, such as the Intel WISP, which consumes 600 micro-amperes when active and 1.5 micro-amperes when sleeping. Units such as the WISP can accomplish useful work in multiple steps—as power becomes available—by dynamic checkpointing and restore. The checkpointing strategy assumes a linear relationship between input voltage and available power, such as that from a capacitor used as a power storage device. A voltage detector senses remaining power and checkpoints the processor's current state to flash memory when the power available drops below a given threshold. This strategy has promise in several application domains, including medical electronics, sensor networks, and security.

- ***The True Cost of Accurate Time***
  Thomas Schmid, Zainul Charbiwala, Jonathan Friedman, and Mani B. Srivastava, University of California, Los Angeles; Young H. Cho, University of Southern California

Maintaining a highly accurate concept of wall clock time for otherwise autonomous wireless nodes has a high power cost. To reduce that cost, a hybrid architecture is proposed in which a relatively higher-power but highly accurate crystal clock circuit is paired with a low-power, low-frequency oscillator on a single chip. The more accurate clock sleeps much of the time and is polled to reset a less accurate LFO clock when needed. The result is a low-power clock chip in a 68-pin configuration that has about 125,000 gates, consumes 20 microwatts on average, and has a 1.2-volt core voltage.

POWER IN NETWORKS

- ***Greening the Switch***
  Ganesh Ananthanarayanan and Randy H. Katz, University of California, Berkeley

Network switches are often provisioned for peak loads, but this is not power-efficient. To reduce power requirements, one can selectively power-down idle switch ports, utilize a separate "shadow port" to accept traffic from a set of powered-down ports, or utilize a "lightweight switch" as a slower, low-power alternative to a fast, higher-power-consumption main switch. If a port is powered down, one loses incoming traffic and queues outgoing traffic on the port. A "shadow port" can receive data from other powered-down ports. A shadow port can reduce but not eliminate data loss, because it can only receive one packet at a time from a group of ports. By contrast, a "lightweight alternative switch" replaces the regular switch and allows it to be completely powered down during nonpeak times. The authors compare these two strategies through trace-driven simulation of the results of the strategy on seven days of network traces from a Fortune 500 company. Lightweight alternative switches turn out to be the most cost-effective of these two strategies, saving, according to the simulation, up to 32% of power.

- ***Hot Data Centers vs. Cool Peers***
  Sergiu Nedevschi and Sylvia Ratnasamy, Intel Research; Jitendra Padhye, Microsoft Research

Should a service be provided in a datacenter or as a peer-to-peer application? This paper analyzes the power requirements of peer-to-peer versus centralized service provisioning in a novel way, by considering the "baseline cost" of the existing systems before the service is added. If one is going to be running underutilized desktop computers anyway, then the added power and cooling requirements from a peer-to-peer application are shown to be cost-effective. The assumptions of the paper were quite controversial to the audience, however; for example, why are the underutilized desktops still powered up when there is nothing to do?

POSTERS

- ***Analysis of Dynamic Voltage Scaling for System Level Energy Management***
  Gaurav Dhiman, University of California, San Diego; Kishore Kumar Pusukuri, University of California, Riverside; Tajana Rosing, University of California, San Diego

Dynamic Voltage/Frequency Scaling (DVFS) is commonly used to save power by lowering the clock rate of a processor at nonpeak periods. DVFS does better at saving power than putting any idle CPU into its lowest-power operating state, but it might not save as much power as shutting down an idle processor and putting memory into self-refresh mode. This perhaps counterintuitive result is predicted by trace-driven simulation.

- ***Energy Aware Consolidation for Cloud Computing***
  Shekhar Srikantaiah, Pennsylvania State University; Aman Kansal and Feng Zhao, Microsoft Research

Energy-aware consolidation is the process of migrating applications and/or services to a small number of physical servers to allow excess servers to be shut down. Simulations demonstrate that packing applications into servers at higher than 50% cumulative CPU load is actually less energy-efficient than keeping the effective load below 50% of peak, due to wasted energy from server thrashing. Similarly, co-locating services so that disk utilization exceeds 70% of peak load leads to energy loss.

- ***Energy-Aware High Performance Computing with Graphic Processing Units***
  Mahsan Rofouei, Thanos Stathopoulos, Sebi Ryffel, William Kaiser, and Majid Sarrafzadeh, University of California, Los Angeles

Low-power energy-aware processing (LEAP) can be applied to code running inside a graphics processor on the video board of a desktop computer. Power savings for CPU-bound applications (e.g., convolution) can be as high as 80%, as demonstrated via trace-driven simulation.

- ***Augmenting RAID with an SSD for Energy Relief***
  Hyo J. Lee, Hongik University; Kyu H. Lee, Purdue University; Sam H. Noh, Hongik University

A solid-state disk (SSD) can be used as a read/write cache for a log-structured filesystem on a RAID disk array. The read-write flash cache is flushed when 90% full. Simulations of this architecture predict power savings of 14% at peak load and 10% at low load.

- ***Workload Decomposition for Power Efficient Storage Systems***
  Lanyue Lu and Peter Varman, Rice University

The traditional definition of "quality of service" (QoS) defines thresholds for response time that cannot be exceeded without penalty. By redefining QoS in statistical terms, one can reduce power requirements for service provision by 50% to 70%. The new definition of QoS allows response times for some percentage of requests to exceed each QoS threshold. Power savings arising from this change are estimated via trace-driven simulation.

- ***CoolIT: Coordinating Facility and IT Management for Efficient Datacenters***
  Ripal Nathuji, Ankit Somani, Karsten Schwan, and Yogendra Joshi, Georgia Institute of Technology

CoolIt is a temperature-aware virtual architecture. A sensing subsystem monitors activity of the virtual architecture, while a cooling control subsystem solves a linear program to optimally control cooling fans. This approach is implemented for Xen in an "ambient intelligent load manager" (AILM).

POWER IN STORAGE

- ***On the Impact of Disk Scrubbing on Energy Savings***
  Guanying Wang and Ali R. Butt, Virginia Polytechnic Institute and State University; Chris Gniady, University of Arizona

Guanying et al. proposed a new metric called the "energy-reliability product (ERP)" to capture the combined performance of energy saving and reliability improving approaches of disks. This metric is a product of energy savings (by spinning down the disk) and reliability improvement in terms of "mean time to data loss." The authors used trace-driven simulations of enterprise applications, such as the Mozilla Web browser, and studied the effects of disk scrubbing and energy management on these applications. Finally, through this study, they showed that ERP can help to identify efficient ways to distribute disk idle time for energy and reliability management.

- ***Empirical Analysis on Energy Efficiency of Flash-based SSDs***
  Euiseong Seo, Seon Yeong Park, and Bhuvan Urgaonkar, Pennsylvania State University

Euiseong et al. analyzed the power consumption pattern of solid-state disk drives (SSDs) with a microbenchmark (using the "DIO tool" workload generator) to show the characteristics for read and write operations at the device level, as well as a macro-benchmark "filebench" to measure real-world behavior of the device. The authors measured differences in terms of power consumption between SSDs and hard-disk drives (HDDs) and also common characteristics shared by SSDs. One audience member asked about the role of logical block lookup tables in improving the reliability of SSDs (by minimizing erasures), and how that affects power requirements. In particular, if a traditional filesystem is written to a SSD, the superblock is not especially vulnerable, because it is logically moved each time it is updated.

CHALLENGES PANEL

Moderator: Feng Zhao, Microsoft Research
Panelists: James Hamilton, Microsoft Research; Randy Katz, University of California, Berkeley; Jeffrey Mogul, Hewlett-Packard Labs

James Hamilton (Microsoft) began his presentation with the question, "Where does power go and what to do about it?" Power losses are easier to track than cooling. Seven watts of each server watt are lost from translation inefficiency. Cooling systems employ a large number of conversions: a "catastrophically bad design." About 33% of cooling power cost is due to mechanical losses. Pushing air 50 feet is catastrophically bad. A secondary problem is evaporative water loss from cooling systems, estimated at 360,000 gallons of water a day for a site. Several creative approaches to the problem include "air-side economization" (open the window!) and cooperative expendable micro-slice servers, with four times the work per watt of current servers.

Randy Katz (Berkeley) asked, instead, "What if the energy grid were designed like the Internet?" Current energy grid technology is a remnant of the machine age, and expertise in power distribution has largely disappeared from academia. As a fresh approach, we can apply principles of the Internet to energy. First, we push intelligence to the edges and concentrate on lower-cost incremental deployment. Enhanced reliability and resilience arise from the same sources as Internet reliability and resilience. The result is the "LoCal-ized datacenter" that is based upon DC distribution rather than AC and contains battery backups (or other forms of power storage) in each rack. This allows flexible use of any kind of power with minimal conversion loss, including stored energy and solar power.

Jeff Mogul (HP) encouraged us to "look between the street lamps" for the next generation of power Ph.D. thesis topics. The street lamps include component power, control theory, and moving work around. These areas are well-explored. There are many topics that fall "between the street lamps," including tradeoffs between reliability and power use, matching customer needs to theoretical solutions, and making it easier to write energy-aware programs. Key challenges to understanding include the boundaries between areas, as well as energy inputs beyond the computer's power supply, including the energy cost of building and disposing of computing hardware.

A spirited discussion ensued in which there were many contributors.

A key principle is "Do nothing well." In other words, stop trying to optimize the effect of every joule going into our hardware; instead, look for median approaches to the problem.

One possible approach is detouring work: Instead of paying for peak energy load, store energy from nonpeak times. However, it remains very difficult to store energy. Innovative approaches include energy harvesting and even compressed-air storage.

Building more power plants to satisfy datacenter demand is not the only way to deal with increasing power demand. We don't know yet how to produce an application-independent layer that does that, and programmers may have difficulties with the resulting level of abstraction.

Another challenge is that of sharing data for mutual benefit. Data privacy is a major problem, but if someone could define what an interesting power trace might be, smaller players could contribute. Alternatively, using open-source applications such as Hadoop allows one to collect power data for one's own application.

There is also a seeming contradiction in the way people and lawmakers react to cooling strategies. If one puts heat into a river, environmentalists are concerned. If one puts heat into the air instead, no one seems to care.

Another potential savings strategy is to reevaluate how datacenters are cooled. We may not want to cool the whole datacenter to 62 degrees. We may want to cool everything to 89 degrees. But then there's no margin for error. In raising the total machine room temperature, we would be operating "nearer to the edge of the hardware function envelope" and any failure of cooling might lead to massive failures of hardware.

The recent rise of cloud computing poses its own power challenges. If everybody outsources storage to Amazon and everyone gets a surge of traffic (e.g., the day after Thanksgiving), do our computer systems have a credit meltdown? What if the whole "ecosystem" undergoes the same set of unforeseen changes?

First Workshop on I/O Virtualization (WIOV '08)

San Diego, CA
December 10–11, 2008

I/O ARCHITECTURE

Summarized by Mike Foss (mikefoss@rice.edu)

- ***Towards Virtual Passthrough I/O on Commodity Devices***
  Lei Xia, Jack Lange, and Peter Dinda, Northwestern University

Lei Xia delivered the first presentation of the workshop, explaining how one might use a model-based approach to allow virtual passthrough I/O on commodity devices. The current approaches to allow high-performance I/O in guest operating systems are limited. In one approach, the virtual machine provides full emulation of the device in order to multiplex it to each guest operating system; however, this requires significant overhead in the VM. To reduce the performance penalty, a guest might bypass the virtual machine altogether in direct-assignment I/O. However, this approach is less secure, since a guest could affect the memory of other guests or the VM itself. Some devices are multiplexed in the hardware and allow each guest to directly access the device while preserving security, but this feature is not available on commodity I/O devices, nor do these devices currently allow migration of guests.

Xia introduced virtual passthrough I/O (VPIO), which allows a guest to have direct access to the hardware for most operations and also allows a guest to migrate. VPIO assumes that there is a simple model of the device that can determine (1) whether a device is reusable, (2) whether a DMA is about to be initiated, and (3) what device requests are needed to update the model. VPIO also assumes that the device can be context-switched, that is, that the device can deterministically save or restore the state pertaining to a guest operating system. For the best performance, the goal of VPIO is to have most guest/device interactions complete without an exit into the VM.

Under VPIO, each access to the device must go through a Device Modeling Monitor (DMM). The purpose of DMM is twofold: (1) It saves enough state about the guest and the device that a guest could migrate to a new VM, and (2) it ensures that the VMM enforces proper security. It also keeps track of a hooked I/O list, which is a set of I/O ports that require VM intervention if accessed by a guest. Unhooked I/O ports may be used by the guest directly. The device is multiplexed by performing a context switch on the device (restoring the guest-specific state into the device). Currently, if the DMM disallows the guest to continue with an operation (e.g., in the case of a DMA to an address out of bounds), the DMM delivers a machine-check exception to the guest. If the device issues an interrupt, it may not be clear to which guest to forward the interrupt, as in the case of receiving a packet on a NIC. Currently, Xia's team is working on finding a general solution to this problem.

Xia's team did implement a model of an NE2000 network card and had it running under QEMU. The model was under 1000 lines of code, and only a small fraction of I/Os (about 1 in 30) needed VM intervention. The remaining challenges for this project include the following: moving more of the model into the guest in order to reduce the cost of a vm_exit; handling incoming device input, such as interrupts without a clear destination guest; and obtaining a device model from hardware manufacturers.

- ***Live Migration of Direct-Access Devices***
  Asim Kadav and Michael M. Swift, University of Wisconsin—Madison

Asim Kadav presented the second paper of WIOV, explaining how to migrate direct-access I/O devices from one virtual machine to another. While direct, or passthrough, I/O offers near-native performance for a guest OS, it inhibits migration, because the VM does not know the complete state of the device. Furthermore, the device on the destination machine may be different from that on the source machine. Asim proposed to use a shadow driver in the guest OS in order to facilitate migrating guests that take advantage of passthrough I/O.

The challenge of the shadow driver is to simultaneously offer both low constant overhead and short downtime during migration. The shadow driver listens to communication between the kernel and the device driver via taps. In its passive mode, the shadow driver keeps track of the state of the driver. It intercepts calls by the driver, tracks shared objects, and logs any state-changing operations.

During migration, or active mode, the shadow driver is responsible for making sure that migration occurs without the need to modify the existing device driver or hardware. First, the shadow driver unloads the old device driver and monitors any kernel requests during the period where there is no driver. Next, it finds and loads the new driver into the appropriate state.

Asim's team modified Xen and Linux in order to implement a prototype shadow driver. The shadow driver implements taps by substituting functions in the kernel/driver interface with wrapper functions. These wrappers were generated by a script that would accommodate any network device. Asim showed that the shadow driver method worked and only cost a percentage point of both extra CPU overhead and network throughput during passive mode. Migration to a new virtual machine took four seconds, but most of the time was spent in the initialization code of the network driver. Asim also showed that migration between heterogeneous NICs was possible by enabling the lowest common denominator of features between the participating NICs. No device driver or hardware modifications were needed in order to use the shadow driver.

- ***Scalable I/O—A Well-Architected Way to Do Scalable, Secure and Virtualized I/O***
  Julian Satran, Leah Shalev, Muli Ben-Yehuda, and Zorik Machulsky, IBM Haifa Research Lab

Muli Ben-Yehuda presented the final paper of the first session of WIOV, a position paper on how I/O should be scaled for any system. The current device driver model presents a unique problem in the OS for several reasons. First, communication with the hardware consists of only register transfer and DMA operations. Furthermore, each driver is vendor-specific and must be maintained by the vendor. They are often the source of bugs in the OS. These problems are pronounced in virtualized systems. Muli proposed to virtualize the entire I/O subsystem rather than each driver or device, in order to enhance the scalability and security of I/O in virtual machines.

The scalable I/O architecture consists of device controllers, I/O consumers, and host gateways. A device controller (DC) is responsible for communicating directly with the device. It implements I/O services and can serve many I/O consumers simultaneously. It also protects devices from unauthorized access. An I/O consumer is any process on the host that wishes to access the device. The I/O consumer accesses the proper DC by first sending the request through a host gateway. The host gateway (HG) is in charge of granting protected I/O mechanisms to all the I/O consumers on the host. It can be thought of as an elaborate DMA engine that provides a DMA to virtual memory. The HG and DC are connected by shared memory or any I/O interconnect in general, which is completely abstracted away from the I/O consumer.

Protected DMA (PDMA) is the mechanism by which the HG and DC communicate. The HG generates a memory credential whenever an I/O consumer wishes to use the DC. This credential is later validated by the HG whenever the DC accesses the consumer's memory.

The scalable I/O protocol grants several benefits over current I/O mechanisms. I/O consumers may submit I/O programs to the DC, which gives a high-level I/O interface to consumers. Furthermore, the I/O subsystem is now isolated from the rest of the operating system, which may improve performance and robustness. A programmable I/O interface also allows for enhanced flexibility and scalability.

Another benefit of scalable I/O is that memory pinning becomes unnecessary. Memory pinning is expensive and puts an error-prone burden on the programmer. In scalable I/O, devices ignore pinning and assume that the memory is always present. In the unlikely case that the desired memory is not present, the device takes an I/O page fault, and the DC and HG communicate in order to resolve the page fault.

STORAGE VIRTUALIZATION

Summarized by Asim Kadav (kadav@cs.wisc.edu)

- ***Block Mason***
  Dutch T. Meyer and Brendan Cully, University of British Columbia; Jake Wires, Citrix, Inc.; Norman C. Hutchinson, University of British Columbia; Andrew Warfield, University of British Columbia and Citrix, Inc.

Block Mason by Dutch Meyer addresses the problem of developing agile storage systems for virtualization by proposing a high-level declarative language to manage blocks. Dutch began by describing the file system interface as basically a block interface but with accessibility issues in practice, as the kernel hides it. However, in virtualized interfaces the block layer becomes more important since shared storage can leverage significant functionality from the block layer. At block level one can add many features such as compression, encryption, deduplication, or even advanced gray box techniques. The key idea of this talk is to provide a user-level framework for building reusable modules at the block level that one can connect to perform more complex tasks. Block Mason helps developers build fine-grained modules and assemble and reconfigure them to build high-level declarative verifiable block manipulation. The implementation of Block Mason was done in Xen using the blktap interface. In user mode, a new scheduler, parser, and driver API were implemented. There were also some minimal updates to blkback.

Dutch further detailed the implementation, discussing the basic building blocks (elements/modules) and connectors (ports/edges). An element example would be as simple as recording I/O requests. Any details of elements can be added to configuration files. The connectors are the ports, identified by names. Block Mason also supports live reconfiguration of the modules built by draining outstanding requests and initializes new ones as they arrive. The architectural support implemented includes message passing and dependency tracking.

As an example of a service using these various constituents, Dutch suggested migrating storage from a local disk to another storage device. The two subservices that are using the

Block Mason interface in this example are I/O handling and background copying, implemented using low-level Block Mason constituents. More complex modules such as cloud-backed disks were also described briefly. Block Mason can be also used to perform other tasks such as correctiveness verification. Block Mason will be integrated into the new blktap2 interface in Xen. Future work will include developing declarative languages to perform block tasks.

Dan Magenheimer from Oracle commented on how powerful Block Mason is and inquired about the things that can be done with Block Mason. Dutch answered that, using Block Mason, one can build simple features and aggregation of simple features such as disk encryption. Himanshu from Microsoft asked about synchronization issues with Block Mason. Dutch explained synch issues with the copy example. In response to a question about whether synchronous write would work properly, Dutch explained that only one port is used to perform I/O in his example. Another questioner asked about block failures and their handling by Block Mason. Dutch replied that one can trap failures and perform recovery actions and explained it in his disk copy example. Muli Ben-Yehuda posited that this may be similar to using pipes, but Dutch said that pipes would give the same expressiveness but coarse-grained modules.

- ***Experiences with Content Addressable Storage and Virtual Disks***
  Anthony Liguori, IBM Linux Technology Center; Eric Van Hensbergen, IBM Research

Eric Van Hensbergen gave a talk on his research on how to reduce redundancy in virtual disk images using content addressable storage. The motivation here is that virtualization causes lot of image duplication with many common files, libraries, etc. In a cloud scenario, the problem is even more severe, with many thousands of disk images on the server. The first part of his talk consisted of analyzing the existing duplication at file and block levels. The results had filtered out duplicates due to hard links and sorted the results to obtain self- and cross-similarity separately. Eric showed considerable overlap in terms of the same blocks in various Linux distributions for their 32/64-bit versions. There are also similar overlaps in different distributions of the same operating system (Linux) and in different versions of the same distribution. All results show considerable overlap in the binaries that can be exploited. Even in analyzing different images created from different install options, there is a large degree of overlap (duplication). These results are also the same for Windows (factoring out swap/hibernation files). Analyzing the deduplication efficiency, the results show a slightly higher efficiency for 8k blocks, but this is primarily due to error associated with partial blocks and the discounting of zero-filled blocks. A disk-based scan was able to identify approximately 93% of the duplicate data.

The second part of the talk compared existing solutions and their solution. The common existing solution is to use Copy-On-Write (COW) disks. The problem with the COW approach is that there is a drift to higher disk usage with application of the same updates to different disks. This is because, as the same updates are applied to similar images, since updates are applied one after another the images get out of sync. Eric's solution is to use a an existing Content Addressable Storage (CAS) system (Venti) as a live backing store and use a filesystem interface on top of it to present to virtual disks. The hypervisor was modified to use these virtual disks. Further, Eric gave some background on CAS and then described some related work including Foundation (CAS to archive VM for backup), Mirage (file-based CAS), and Internet Suspend Resume (CAS to access personal computing environments across a network by using virtualization and shared storage). He also cited a work from Data Domain in which a filesystem-based approach is used to leverage dedupulication in backups.

In terms of implementation, Eric reused an existing block-based CAS, vbackup in Plan 9. QEMU/KVM ran the virtual machines and provided a hook via vdiskfs that uses vbackup as the underlying store.

The evaluation consisted of measuring block utilization before and after same updates on two similar disk images. There was also an additional micro-benchmark using the dd command. The results show better results with CAS and compressed CAS than those with COW. The performance, however, takes a hit and the boot time to bring up the system using CAS is much higher. In terms of the micro-benchmark, CAS performs much worse, running at 11 Mbps compared to 160 Mbps (without CAS) in raw mode. To summarize the evaluation, the space efficiency is great but performance is bad, since Venti is single-threaded and synchronous and also was configured with a small cache for these experiments. Their future work includes reworking CAS for live performance, experimenting with flash disks for index storage, and building in support for replication and redundancy.

A questioner asked about the case of dirty blocks and Eric replied that he was using a write buffer to avoid using them for CAS; however, dirty blocks can be used as a single large cache for all virtual machines. Another person from the audience pointed out a related work from CMU about finding similarities using encryption system. When asked whether this was even the right approach to the problem, Eric said he didn't know, but it was easy to implement and took only two weeks. Jake Oshins from Microsoft wondered whether it would be advantageous for the file system to know what blocks are being deleted. Eric said it would definitely help and pointed out a related work that addresses this.

- ***Paravirtualized Paging***
  Dan Magenheimer, Chris Mason, Dave McCracken, and Kurt Hackel, Oracle Corporation

Dan's talk covered a new type of cache, called hcache, aimed at resolving memory issues in virtualization. Memory is cheap but, currently, memory systems are running unutilized and are being wasted. Most recent work on virtualization has concentrated on efficient CPU and I/O utilization, but little work has been directed toward memory utilization. Described in the talk was a proposed approach to resolving memory issues in virtualization by allocating a separate pool in the virtual machine's memory space, called transcendental memory. Dan then focused on the basics of physical memory concepts in a single machine and virtualization servers and discussed the common memory issues there. In a single machine with a single operating system, the memory is basically a huge page cache that is never optimized, and a lot of idle memory in page cache is simply wasted. This is because the operating system has no way of determining which areas of page cache are being utilized and which are not. The pages are moved out of page cache using a page cache replacement algorithm (PRFA); even PRFA cannot determine the correct working set of page cache, resulting in many false-negative page evictions, making the matter worse.

The situation is no different in virtualization servers, where the physical memory is still used inefficiently. Memory allocations to guest virtual machines are either by static partitioning of memory or by dynamic partitioning of memory. Neither of them is helpful. Static partitioning has problems such as fragmentation of memory and memory holes resulting from machine migration. There is also almost no load balancing of memory by the hypervisor in static partitioning. The second method, dynamic partitioning (also known as ballooning), uses a balloon driver in the guest virtual machines. Ballooning tunnels pages across balloon drivers to transfer memory from one virtual machine to another to perform load balancing. This scheme also has many issues, since OS/virtual machines rarely voluntarily give up memory and always demand more memory. There are further difficulties in determining which virtual machine is the neediest here. Also, ballooning is not instantaneous for large or fast changes in balloon size.

The solution here aims to answer unanswered questions such as how to reclaim I/O without increasing disk I/O. Also, the problems of ballooning and memory just mentioned can be alleviated by using the solution described. The solution provided is to reclaim all idle memory into a pool called the transcendental memory pool. All guests access memory via the hypervisor using transcendental APIs, which cause memory operations that are synchronous, page-oriented, and copy based. This memory pool is subdivided into four subpools: private ephemeral, private persistent, shared ephemeral, and shared persistent. The private ephemeral memory is labeled as "hcache." The false evictions now fall into hcache and have a low cost now. Also, any memory in hcache can be thrown away quickly, resulting in faster memory allocations to virtual machines so that operations such as ballooning can be done quickly. Dan also pointed out that very minimal changes are needed to implement this solution. He further described hswap, which is a persistent, private cache that works like a pseudo swap device. It helps to balloon fast, as ops from pool are faster and there is no thrashing as memory is allocated from the pool. He further pointed out that shared ephemeral/persistent pools can act as shared memory for inter-VM communication; this is a part of future work.

Transcendental memory can also be used in a single OS, because API is clean, as a useful abstraction (NUMA memory, hot-swappable memory, or SSDs). It can also be used as a cache for network file systems. In conclusion, transcendental memory is a new way to manage memory for single operating system and virtualization servers and reduces many of the existing memory issues.

To the question of whether one needs contiguous memory for transcendental memory, Dan replied that the transcendental memory solution works even with fragmented memory.

DEVICE VIRTUALIZATION

Summarized by Jeff Shafer (shafer@rice.edu)

- ***GPU Virtualization on VMware's Hosted I/O Architecture***
  Micah Dowty and Jeremy Sugerman, VMware, Inc.

Micah Dowty presented a paper on how to virtualize a GPU. In this talk he introduced a taxonomy of GPU virtualization strategies and discussed specifics of VMware's virtual GPU.

A GPU can provide significant computation resources, and both desktop and server virtualized environments seek to take advantage of these resources. Virtualizing a GPU poses many different challenges, however. There are multiple competing APIs available, and these APIs are complicated with hundreds of different entry points. In addition, the APIs and GPUs are programmable. Every GPU driver is also a compiler, and each API includes a language specification. Hardware GPUs are all different, covering a wide range of architectures that are often closely guarded secrets that change frequently between product revisions. Finally, the hardware state of the GPU chip and associated memory is large, covering gigabytes of data in a highly device-specific format including in-progress DMAs and computation.

There are several potential options to virtualize a GPU, as presented in the taxonomy. These strategies include capturing application API calls at a high level and proxying them to another domain for execution (API remoting), providing a virtual GPU for each guest with which the native software stack communicates (device emulation), and various passthrough architectures to tunnel GPU commands down to the actual hardware. Each technique has various tradeoffs in terms of performance and isolation.

The VMware virtual GPU uses a combination of techniques, most notably device emulation and API remoting, to provide accelerated GPU support in a virtualized environment on top of any physical GPU. With this architecture, interactive graphics applications can now be run at a usable performance level, whereas it was not possible to run them in a virtualized environment before. Future work will focus on new pass-through techniques as well as the development of virtualization-aware GPU benchmarks that stress, not the raw GPU hardware performance, but, rather, the API-level paths that are at issue in a virtualized system.

One audience member asked about the challenges involved in migrating virtual machines across different GPUs. Dowty replied that migration requires reading all the state from the GPU and memory, but this is not always possible considering that some state is generated by the graphics card itself and is not always accessible to the driver or API. GPU vendors have a lot of flexibility in implementing new technologies (such as SR-IOV) to make virtualization and migration simpler and more complete.

- ***Taming Heterogeneous NIC Capabilities for I/O Virtualization***
  Jose Renato Santos, Yoshio Turner, and Jayaram Mudigonda, Hewlett-Packard Laboratories

Jose Renato Santos from HP Labs presented a paper on a network I/O virtualization (IOV) management system that can translate high-level goals into low-level configuration options. In addition, methods for efficient guest-to-guest packet switching were discussed.

In recent years, different vendors have provided many mechanisms for I/O virtualization, such as software virtualization, multi-queue NICs, and SR-IOV multifunction NICs. In the process, however, they have created significant challenges for managing networks of heterogeneous devices, each with different hardware and software approaches to virtualization. Configuration becomes more complex and fragile with increasing diversity in IOV mechanisms. What is needed is a higher-level abstraction for I/O configuration, where users specify logical networks and a mapping of virtual interfaces to logical networks, and then the system selects and configures the appropriate mechanism.

This configuration can be done statically or dynamically. Although a static system may be simpler, consider a case where there are more guests than hardware NIC contexts available to support them. Then a dynamic management system that looks at current workload levels may be needed for optimal assignment. In addition to a new configuration mechanism, a spectrum of methods for efficient intranode guest-to-guest packet switching were also discussed, including switching in software, on the NIC, or in external network devices. All these techniques have tradeoffs in terms of CPU, I/O bandwidth, link bandwidth, and memory use, and this must be considered by the high-level management tool depending on constraints input by the user.

One audience member asked how frequently the system can change its configuration based on these high-level policy guidelines. Jose answered that this is an open question, but certainly not on every packet. There are several concerns involving maintaining packet ordering and minimizing setup/teardown overheads.

- ***Standardized but Flexible I/O for Self-Virtualizing Devices***
  Joshua LeVasseur, Ramu Panayappan, Espen Skoglund, Christo du Toit, Leon Lynch, and Alex Ward, Netronome Systems; Dulloor Rao, Georgia Institute of Technology; Rolf Neugebauer and Derek McAuley, Netronome Systems

Rolf Neugebauer spoke about some of the limitations of the SR-IOV standard for virtualizing complex network devices and proposed a new approach, software configurable virtual functions, to provide increased flexibility for virtualization.

In today's networking environment, multi-queue NICs are an emerging standard, and some include hardware support for virtualization. Hypervisors allow assignment of PCI device functions to virtual machines by virtualizing the PCI configuration space. Moreover, modern chipsets include I/O MMUs to provide DMA isolation and address translation. The SR-IOV (Single Root I/O Virtualization) standard ties these three trends together and allows endpoints such as network cards to be enumerated as PCI virtual functions. Because this is performed in hardware through the device configuration space, however, the SR-IOV has limits to its flexibility.

The new Software Configurable Virtual Functions (SCVF) is proposed for highly programmable network devices such as the Netronome NFP3200. SCVF is built on the same base PCI Express technologies and provides isolated access to virtual functions using IOMMUs. Rather than using the hardware-based configuration space and device support to provide virtualization, however, it performs device enumeration by host OS software. In SCVF, the PCI configuration space is not used to enumerate virtual functions. Rather, SCVF simply presents a PCI device to the host OS. The OS loads a card driver for the physical device function. This driver acts as a privileged control driver and implements a virtual PCI bus on which SCVFs are enumerated as full PCI devices. The operating system recognizes the virtual PCI bus, and then everything "just works." When implemented in Linux, the kernel loads the physical function control driver just as for other devices, and no changes were required to Linux or Xen. The existing software stack is used for hot-plugging, device discovery, and PCI device assignment.

An audience member asked how, after an interrupt is generated, it is routed and virtualized. Rolf answered that the host sets up a list of MSI interrupts via the card driver, which emulates MSI, and then relies on the hypervisor to route those interrupts to CPU cores normally. Thus there are two levels of PCI virtualization: their driver, and then the Xen back-end/front-end interrupt virtualization.

- ***SR-IOV Networking in Xen: Architecture, Design and Implementation***
  Yaozu Dong, Zhao Yu, and Greg Rose, Intel Corporation

Greg Rose presented a paper on the SR-IOV specification and its application to network devices to provide direct I/O in a virtualized environment.

SR-IOV (Single Root I/O Virtualization and Sharing) is a PCI-SIG standard released in September 2007 for sharing device resources on virtualization-capable hypervisors or kernels. It specifies how a single physical function (PF) device should share and distribute its resources to many virtual functions (VFs). It is not networking-specific but, when properly employed in a network device, should provide the full native I/O bandwidth to a virtualized guest operating system and improve scalability over emulation/paravirtualization as more virtual machines are added.

Greg presented a network architecture that includes the SR-IOV NIC, Xen hypervisor, and individual guest domains. Domain 0 runs the physical function device driver and accesses the physical functions of the NIC, while each guest domain runs a virtual function device driver and accesses a virtual function of the NIC. The physical function device driver is responsible for controlling all of the virtual function capabilities and providing configuration services. It maintains administrative control of all the Tx/RX queues, and thus it has ultimate responsibility for device security. The virtual function device driver, in contrast, is similar to a normal NIC driver. It serves as an I/O engine in the virtual machine to "pump packets" to the NIC and depends on the physical function driver for most configuration and notification of events.

The presentation concluded with a demo of a functional SR-IOV NIC running in the lab and a discussion of future work. Areas that need further effort include handling a physical function driver reset (such as one caused by a power state transition), because that reset also affects all the virtual function drivers that depend on it. In addition, network-specific management tools are needed to set parameters such as replication, loopback, MAC addresses, and more. One audience member asked where the packets go when two virtual machines on the same host are communicating. Greg replied that the packets are going down to a layer-2 switch fabric in the physical NIC and then going back up to the other guest. Domain0 never sees the intra-VM packets.

USENIX Upcoming Conferences

8th International Workshop on Peer-to-Peer Systems (IPTPS '09)

Co-located with NSDI '09

APRIL 21, 2009, BOSTON, MA, USA
http://www.usenix.org/iptps09

2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET '09)

Co-located with NSDI '09

APRIL 21, 2009, BOSTON, MA, USA
http://www.usenix.org/leet09

6th USENIX Symposium on Networked Systems Design and Implementation (NSDI '09)

Sponsored by USENIX in cooperation with ACM SIGCOMM and ACM SIGOPS

APRIL 22–24, 2009, BOSTON, MA, USA
http://www.usenix.org/nsdi09

12th Workshop on Hot Topics in Operating Systems (HotOS XII)

Sponsored by USENIX in cooperation with the IEEE Technical Committee on Operating Systems (TCOS)

MAY 18–20, 2009, MONTE VERITÀ, SWITZERLAND
http://www.usenix.org/hotos09

2009 USENIX Annual Technical Conference

JUNE 14–19, 2009, SAN DIEGO, CA, USA
http://www.usenix.org/usenix09

2nd Workshop on Cyber Security Experimentation and Test (CSET '09)

Co-located with USENIX Security '09

AUGUST 10, 2009, MONTREAL, CANADA
http://www.usenix.org/cset09
Paper submissions due: May 15, 2009

3rd USENIX Workshop on Offensive Technologies (WOOT '09)

Co-located with USENIX Security '09

AUGUST 10, 2009, MONTREAL, CANADA
http://www.usenix.org/woot09
Paper submissions due: May 26, 2009

2009 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections (EVT/WOTE '09)

Co-located with USENIX Security '09

AUGUST 10–11, 2009, MONTREAL, CANADA
http://www.usenix.org/evtwote09
Paper submissions due: April 17, 2009

4th USENIX Workshop on Hot Topics in Security (HotSec '09)

Co-located with USENIX Security '09

AUGUST 11, 2009, MONTREAL, CANADA
http://www.usenix.org/hotsec09
Paper submissions due: May 4, 2009

18th USENIX Security Symposium

AUGUST 12–14, 2009, MONTREAL, CANADA
http://www.usenix.org/sec09

22nd ACM Symposium on Operating Systems Principles (SOSP '09)

Sponsored by ACM SIGOPS in cooperation with USENIX

OCTOBER 11–14, 2009, BIG SKY, MT, USA
http://www.sigops.org/sosp/sosp09/

23rd Large Installation System Administration Conference (LISA '09)

Sponsored by USENIX and SAGE

NOVEMBER 1–6, 2009, BALTIMORE, MD, USA
http://www.usenix.org/lisa09
Submissions due: April 30, 2009

ACM/IFIP/USENIX 10th International Middleware Conference

NOV. 30–DEC. 4, URBANA CHAMPAIGN, IL
http://middleware2009.cs.uiuc.edu/

USENIX: The Advanced Computing Systems Association

TECHNICAL SESSIONS AND TRAINING PROGRAM INFORMATION AND HOW TO REGISTER ARE AVAILABLE ONLINE AND FROM THE USENIX OFFICE:
http://www.usenix.org/events | Email: conference@usenix.org | Tel: +1.510.528.8649 | Fax: +1.510.548.5738

Innovation co-creation

with Infosys

SETLabs (Software, Engineering and Technology labs) is the research and innovation arm of Infosys. Over 500 strong, SETLabs is focused on leveraging cutting edge technologies in the areas of Software Engg, Convergence, Knowledge Driven Systems, Distributed and High Performance Computing, Cloud Computing and Security & Privacy.

We invite you to partner with us on the Infosys Technology Forum to interact and co-create with us.

Register at http://forums.infosys.com/technology

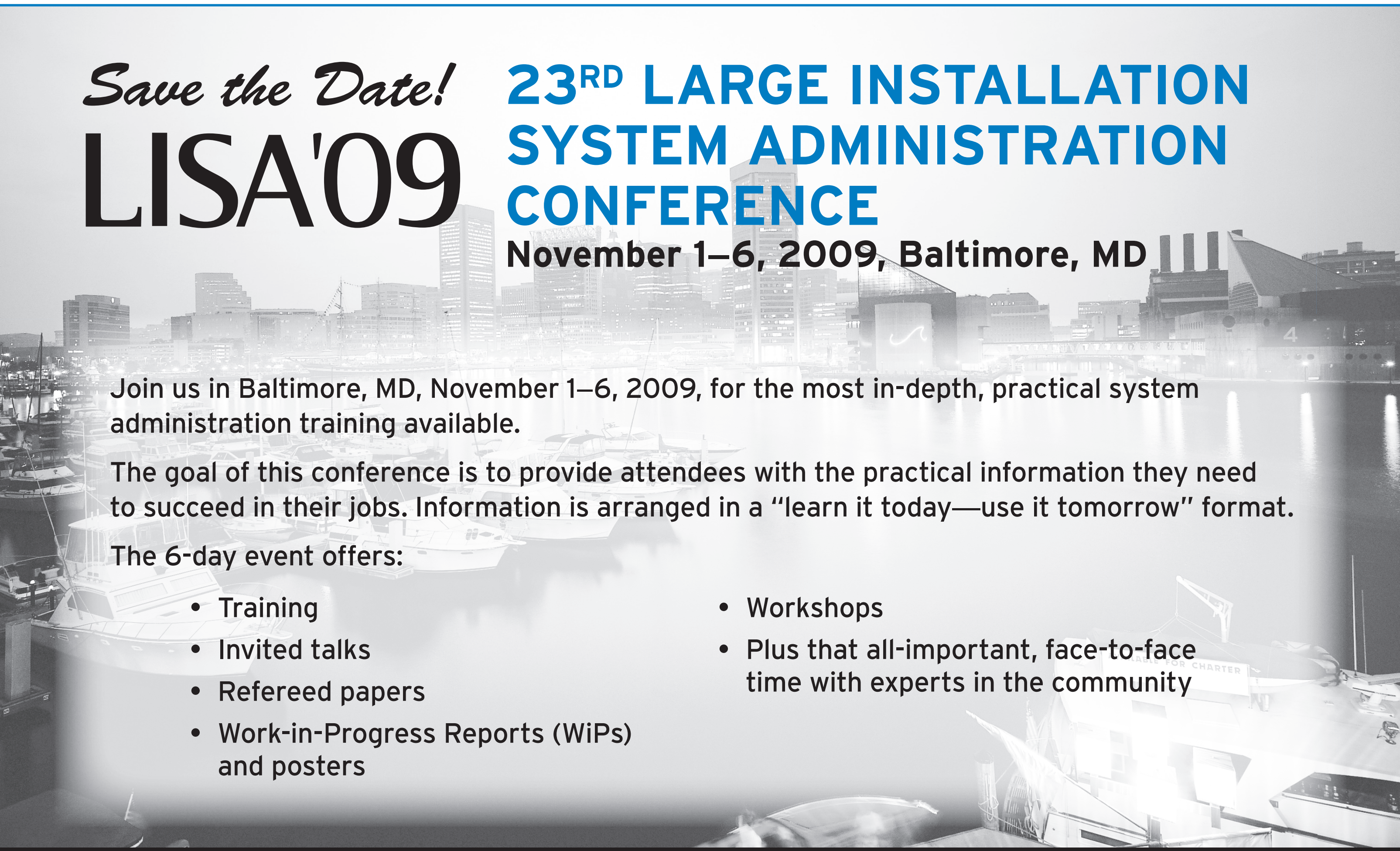
Infosys®

www.infosys.com

;login:
USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710

POSTMASTER
Send Address Changes to *;login:*
2560 Ninth Street, Suite 215
Berkeley, CA 94710

PERIODICALS POSTAGE
PAID
AT BERKELEY, CALIFORNIA
AND ADDITIONAL OFFICES



Save the Date!

LISA'09

23RD LARGE INSTALLATION SYSTEM ADMINISTRATION CONFERENCE

November 1–6, 2009, Baltimore, MD

Join us in Baltimore, MD, November 1–6, 2009, for the most in-depth, practical system administration training available.

The goal of this conference is to provide attendees with the practical information they need to succeed in their jobs. Information is arranged in a "learn it today—use it tomorrow" format.

The 6-day event offers:

- Training
- Invited talks
- Refereed papers
- Work-in-Progress Reports (WiPs) and posters
- Workshops
- Plus that all-important, face-to-face time with experts in the community

http://www.usenix.org/lisa09/loa