Conference Reports

In this issue:

11th USENIX Symposium on Operating Systems Design and Implementation

October 6–8, 2014, Broomfield, CO

Summarized by Radu Banabic, Lucian Carata, Rik Farrow, Rohan Gandhi, Mainak Ghosh, Yanqin Jin, Giorgos Kappes, Yang Liu, Haonan Lu, Yu Luo, Amirsaman Memaripour, Alexander Merritt, Sankaranarayana Pillai, Ioan Stefanovici, Alexey Tumanov, Jonas Wagner, David Williams-King, and Xu Zhao

Opening Remarks

Summarized by Rik Farrow (rik@usenix.org)

OSDI '14 started with a beautiful, sunny day outside of Boulder, Colorado. The nice weather lasted the entire conference, which was also brightened by record attendance.

The program chairs, Jason Flinn (University of Michigan) and Hank Levy (University of Washington), told the crowd that they had moved to shorter presentations, just 22 minutes, so they could fit four presentations into each session. By doing this, they raised the acceptance rate from the old range of 12–14% to 18%. There were 242 submissions, with 42 papers accepted. The PC spent one-and-a-half days in meetings in Seattle and used an External Review Committee to help reduce the workload on the PC. Each PC member reviewed 30 papers (on average), down from 45.

USENIX and OSDI '14 sponsors made it possible for 108 students to attend the conference via grants. Overall attendance was also at record levels.

The Jay Lepreau Best Paper Awards were given to the authors of three papers: "Arrakis: The Operating System Is the Control Plane" (Peter et al.), "IX: A Protected Dataplane Operating System for High Throughput and Low Latency" (Belay et al.), and "Shielding Applications from an Untrusted Cloud with Haven" (Baumann et al.). There was no Test of Time award this year.

Who Put the Kernel in My OS Conference?

Summarized by Giorgos Kappes (gkappes@cs.uoi.gr) and Jonas Wagner (jonas.wagner@epfl.ch)

Arrakis: The Operating System Is the Control Plane

Simon Peter, Jialin Li, Irene Zhang, Dan R.K. Ports, Doug Woos, Arvind Krishnamurthy, and Thomas Anderson, University of Washington; Timothy Roscoe, ETH Zürich

Jay Lepreau Best Paper Award

Simon began his talk by explaining that traditional operating systems like Linux do not take advantage of modern hardware that supports I/O virtualization, and they impose significant overheads because the kernel mediates all data accesses.

Simon introduced Arrakis, a server OS that splits the role of the kernel into the control and the data plane. The control plane lies in the kernel and is responsible for functionalities like naming, access control, and resource limits. These functionalities are used infrequently, to set up a data plane, for example. On the other hand, the functionality of the data plane is moved into applications. Applications perform I/O processing themselves by taking advantage of hardware I/O virtualization, while protection, multiplexing, and I/O scheduling are directly performed by the hardware. The copying of data between the kernel and the user space is no longer needed. A per application dynamically linked library implements the data plane interfaces which are tailored to the application. The network data plane interface allows applications to directly talk with the hardware in order to send and receive packets. The storage data plane interface allows the applications to asynchronously read, write, and flush data into its assigned virtual storage area (VSA). The storage controllers map this virtual area to the underlying physical disks.

There is also a virtual file system (VFS) in the kernel that performs global naming. In fact, the application has the responsibility to map data onto its VSA and register names to the VFS. The storage data plane also provides two persistent data structures: a log and a queue. These allow

operations to be immediately persistent, protect data against crash failures, and reduce the operations' latency.

To evaluate Arrakis, the authors implemented it in Barrelfish OS and compared its performance with Linux. By using several typical server workloads and well-known key-value stores, they show that Arrakis significantly reduces the latency of set and get operations while increasing the write throughput 9x. Arrakis also scales better than Linux to multiple cores, because I/O stacks are local to applications and are application specialized.

John Criswell (University of Rochester) asked what would happen if the Linux kernel made the hardware devices directly available to applications. Simon replied that there is a lot of related work that does try to keep the Linux kernel intact. However, it does not provide the same performance as Arrakis, since the kernel has to be called eventually. System call batching can mitigate this, however this trades off latency for higher throughput. Geoff Kuenning (Harvey Mudd College) asked whether Redis must be running in order to mediate disk I/O through its dedicated library and what would happen if someone damaged the Redis config file preventing it from starting up. Simon answered that the idea behind the indirection interface is provided by the libIO stack in Redis's dedicated library. The stack includes a server that receives I/O operations and directs them to the config file. Aaron Carol (NICTA) first pointed out that it seems that Arrakis designates a process as a host for a collection of files, and then asked what performance implications would come with accessing these files from a different process. Simon replied that the process to which the file belongs will have faster access. Different processes need to perform IPC, which typically has some costs, but Barrelfish introduced fast IPC. Finally, Peter Desnoyers (Northeastern University) asked how Arrakis performs for very high connection rate applications, e.g., a large Web server. Simon said that not every connect operation needs a control-plane call. For example, a range of port numbers can be allocated to a server with a single control-plane call.

Decoupling Cores, Kernels, and Operating Systems

Gerd Zellweger, Simon Gerber, Kornilios Kourtis, and Timothy Roscoe, ETH Zürich

Gerd motivated his work with current hardware and software trends: In an era of Turbo Boost, core fusion, and dark silicon, power management is crucial. Tomorrow's OSes need to switch between specialized cores based on their workload. They should be able to dynamically reconfigure or update themselves. All this is possible in Barrelfish/DC, a multikernel architecture where OS state has been carefully factored per CPU. This allows a separate kernel to control each CPU and to treat CPUs as pluggable devices.

The main challenge is to cleanly and quickly shut down a core. Barrelfish/DC's solution is to move CPU state out of the way quickly, then dismantle it lazily, if needed. State is encapsulated in an "OSnode" containing capabilities that represent application and OS state, as well as a Kernel Control Block representing hardware state. OSnodes can be moved to another CPU, where they are "parked" until dismantled or restarted.

Experiments show that Barrelfish/DC can shut down nodes very quickly (in <1 ms). Moreover, the time to do so does not depend on the system load. The Barrelfish/DC team has demonstrated that their system enables various new use cases, e.g., live updates of the OS kernel or temporarily switching to a real-time kernel for a specific task.

Chris Frost (Google) asked how interrupts interacted with moving cores. Gerd explained that Barrelfish/DC handles three types of interrupts: for timers, inter-process communication, and devices. When a device driver is moved to another core, device interrupts must be disabled on the source core before the move, and the driver must poll the device state once it is running on the destination core. Srivatsa Bhat (MIT) asked whether Barrelfish/DC's energy savings could also be achieved by the power modes in today's CPUs. Gerd answered that this is possible, but that his work goes beyond power savings to explore completely new ideas. Someone from Stanford asked about the cost of dismantling a state. Gerd explained that this depends on the application (e.g., whether it uses shared message channels) and that it was impossible to give a specific number. Malte Schwarzkopf (Cambridge) asked whether this would work on non-cache-coherent architectures. We don't know, said Gerd, because such a system has not yet been built.

Jitk: A Trustworthy In-Kernel Interpreter Infrastructure

Xi Wang, David Lazar, Nickolai Zeldovich, and Adam Chlipala, MIT CSAIL; Zachary Tatlock, University of Washington

Today's kernels execute untrusted user-provided code in several places: BSD Packet Filter (BPF) programs to filter packets or system calls, DTrace programs for profiling, etc. Xi Wang started his talk by showing how hard it is to build correct and secure interpreters for such user-provided code. He and his colleagues created Jitk, a verified JIT compiler for programs in the BPF language, to eradicate interpreter bugs and security vulnerabilities once and for all.

Jitk models both a BPF program and the CPU as state machines and comes with a proof that, whenever Jitk successfully translates a BPF program to machine code, all its state transitions correspond to transitions at the machine code level. Jitk's proof builds on the CPU model and an intermediate language, Cminor, from the CompCert project. The main proof is complemented by a proof that decoding BPF bytecode is the inverse operation of encoding it, and by a high-level specification language that simplifies the creation of BPF programs. Putting these components together, users can have confidence that the semantics of well-understood, high-level programs are exactly preserved down to the machine code level.

Jitk consists of 3510 lines of code, two thirds of them proof code. The JIT's performance is comparable to the interpreter that ships with Linux. Due to the use of optimizations from CompCert, it often generates code that is smaller.

Rich Draves (Microsoft) enquired how Jitk compares to proof-carrying code. Xi Wang answered that Jitk proves strong correctness properties, whereas proof-carrying code usually demonstrates only its memory safety. Also, Jitk's proof holds for any translated program. Malte Schwarzkopf (Cambridge) wondered about the value of Jitk's proof, given the large trusted code base. Xi Wang answered that all theorem proofing techniques share this problem. The trusted code base consists of layers that build on each other, and we can gain confidence by analyzing layers in isolation and trusting that errors would not propagate. Volodymyr Kuznetsov (EPFL) asked whether proven code could be isolated from untrusted code. Xi Wang pointed to the related Reflex project (PLDI 2014) where, for example, isolation between different browser tabs has been proven.

IX: A Protected Dataplane Operating System for High Throughput and Low Latency

Adam Belay, Stanford University; George Prekas, École Polytechnique Fédérale de Lausanne (EPFL); Ana Klimovic, Samuel Grossman, and Christos Kozyrakis, Stanford University; Edouard Bugnion, École Polytechnique Fédérale de Lausanne (EPFL)

Jay Lepreau Best Paper Award

Adam started by mentioning the increasing mismatch between modern hardware and traditional operating systems. While the hardware is very fast, the OS becomes the bottleneck. This results from the complexity of the kernel and its interface, while interrupts and scheduling complicate things even further. Instead, today's datacenters require scalable API designs in order to support a large number of connections, high request rates, and low tail latency.

To achieve these goals, the authors designed IX, a data-plane OS that splits the kernel into a control plane and multiple data planes. The control plane consists of the full Linux kernel. It multiplexes and schedules resources among multiple data planes and performs configuration. Each data plane runs on dedicated cores and has direct hardware access by utilizing hardware virtualization. Additionally, IX leverages VTX virtualization extensions and Dune (OSDI '12) to isolate the control plane and the data planes as well as to divide each data plane in half. The first half includes the IX data-plane kernel and runs in the highest privilege ring (ring 0), while the other half comprises the user application and libIX and runs in the lowest privilege ring (ring 3).

libIX is a user-level library that provides a libevent-like programming model and includes new interfaces for native zero-copy read and write operations. Describing the IX design, Adam briefly presented the IX execution pipeline and mentioned its core characteristics. The IX data plane makes extensive use of adaptive batching, which is applied on every stage of the network stack. Batching is size-bounded and only used in the presence of congestion. This technique decreases latency and improves instruction cache locality, branch prediction, and prefetching, and it leads to higher packet rates. Additionally, the IX data plane runs to completion of all stages needed to receive and transmit a batch of packets, which improves data cache locality. It also removes scheduling unpredictability and jitter, and it enables the use of polling.

The authors evaluated a prototype implementation of IX against a Linux kernel and mTCP, and showed that IX outperforms both in terms of throughput and latency. Additionally, IX achieves better core scalability. The authors also tested memcached and showed that IX reduces tail latency 2x for Linux clients and by up to 6x for IX clients. It can also processes 3.6 times more requests.

Brad Karp (UCL) asked whether the technique used to achieve data cache locality affects instruction cache locality. He also asked whether integrated layer processing conflicts with the techniques used in IX. Adam answered that they didn't observe that data cache locality adversely affects instruction cache locality. If the amount of data that accumulates between processing phases fits in data cache, then the instruction cache is not a bottleneck. An upper limit on the batch size also helps. Simon Peter (University of Washington) asked how the batching used in IX affects tail latency, especially with future, faster network cards. Adam said that batch limits have no impact at low throughputs because batching is not used. But even at high throughputs, batching leads to low latency because it reduces head-of-line blocking. The next question was about the run-to-completion model. Michael Condict (NetApp) asked whether no one is listening on the NIC when the core is in the application processing stage. Adam replied that while the application performs processing, the NIC queue is not being polled. Michael also asked whether this technique can be used on servers that have high variability in processing time. Adam said that IX distinguishes between I/O and background threads. Applications could use background threads for longer-duration work. They also want to use interrupts to ensure that packets are not unnecessarily dropped. However, interrupts should only be a fallback. Steve Young (Comcast) asked whether they encountered dependencies between consecutive operations due to batching. Adam answered that this was a big issue when they designed their API, but careful API design can prevent such problems. They also use a heuristic: the batch is a set of consolidated network requests from a single flow. If one fails, they skip the other requests in the flow.

Data in the Abstract

Summarized by Yang Liu (yal036@cs.ucsd.edu) and Yanqin Jin (y7jin@cs.ucsd.edu)

Willow: A User-Programmable SSD

Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson, University of California, San Diego

Steve Swanson first highlighted that Willow aims to make programmability a central feature of storage devices and to provide a more flexible interface. Then he gave a retrospective view of the evolution of storage technology, starting from slow hard disks to emerging PCIe attached SSDs backed by flash or phase change memory (PCM). Unlike conventional storage, these new SSDs promise much better performance as well as more flexibility, urging people to rethink the interface between storage soft-

ware and storage device. Previously, SSDs had been connected to the host via rigid interfaces such as SATA, SAS, etc., while SSDs have flexible internal components. Thus the challenge is to expose the programmability to application developers who want to build efficient and safe systems.

Steve presented Willow, a system that (1) provides a flexible interface that makes it easy to define new operations using the C programming language, (2) enforces file system permissions, (3) allows execution of untrusted code, and (4) provides OS bypass so that applications can invoke operations without system calls.

Willow is especially suitable for three types of applications: data-dependent logic, semantic extensions, and privileged execution. Intensive and complex data analytics is not the sweet spot for Willow's design, mainly because of the wimpy CPUs inside the SSD, limited by power and manufacturing cost.

Steve then presented Willow's system architecture. Willow, which is implemented on a BEE3 FPGA board, has similar components to a conventional SSD: It contains eight storage processor units (SPUs), each of which includes a microprocessor, an interface to an inter-SPU interconnect, and access to an array of non-volatile memory. Each SPU runs a small SPU-OS, providing basic functionality such as protection. Willow is connected with the host machine via NVM Express (NVMe) over PCIe.

Willow allows application programmers to download SSD apps to the storage device. An SSD app has three components: a host-side user-space library, the SPU code, and an optional kernel module. Willow's interface is very different from that of a conventional SSD. Host threads and SPUs rely on a RPC mechanism to communicate with each other. The RPC mechanism is sufficiently flexible so that adding new interface is easy. There is nothing storage-centric about the RPC since SPUs and host can send RPCs in any direction, from host to storage and vice versa.

Steve also introduced the trust and protection model adopted by Willow in which a file system sets protection policy while Willow firmware enforces it. In particular, he pointed out that, thanks to hardware-written processID information in the message headers, RPCs cannot be forged.

To demonstrate the usefulness of Willow, Steve guided the audience through a case study and invited them to read the paper for further details. In the talk, he showed that by implementing moderate amount of transaction support inside Willow, some applications become easy to write, with a noticeable performance gain. He also emphasized that the programmability of Willow actually makes tweaking and tuning the system faster and more convenient.

Pankaj Mehra (SanDisk) asked whether future programmable SSD can work with the new NVMe standard, given the evolution of non-volatile memory. Steve said that they are actually doing some work to answer that question, and the simple answer is yes. One of the possible ways to do that is to go through the NVMe standard and add some extensions, such as allowing generic calls from the device to the host, which will fit in the NVMe framework. Peter Chen (University of Michigan) asked whether Steve saw any technological trends that could reduce the need for programmable SSDs, when faster buses emerge. Steve said that he doesn't see trends in that direction because latency doesn't decrease much even though PCIe bandwidth continues to grow. Thus, it is still a problem if there is too much back-and-forth communication between the host and the SSD. In addition, the programming environment on the SSD is much simpler than that on the host, making the SSD more reliable and predictable. He said he can see a consistent trend towards pushing more capable processors on SSDs, and similar trends on GPUs and network cards as well. In his opinion, this is a broad trend.

Physical Disentanglement in a Container-Based File System

Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, University of Wisconsin-Madison

Isolation is vital for both reliability and performance, and is widely used in various computer systems. Lanyue Lu pointed out current system design does not have isolation in the physical on-disk structures of file systems, resulting in poor reliability and performance. He further showed several problems caused by such physical entanglement, and then proposed a novel file system called IceFS to solve the problem.

Without isolation of the on-disk structures, logically distinct files may well use co-located metadata, thus the corruption of one file can affect another unrelated file or even lead to global errors, such as marking a file system as read-only. File systems also use bundled transactions to commit changes of multiple files, causing the performance of independent processes to be entangled.

IceFS introduces a new abstraction called cubes, which are implemented as special isolated directories in a file system. The abstraction of cubes enables applications to specify which files and directories are logically related. Multiple cubes do not share the same physical resources. Any cube does not contain references to any other cube. Lanyue showed that IceFS offers up to eight times faster localized recovery and up to 50 times higher performance. He also told the audience that IceFS can reduce downtime of virtualized systems and improve recovery efficiency of HDFS.

The design of IceFS follows three core principles: (1) no shared physical resource across cubes, (2) no access dependency (one cube will not cross-reference other cubes), and (3) no bundled transactions. IceFS uses a scheme called "transaction splitting" to disentangle transactions belonging to different cubes. Lanyue demonstrated the benefits within a VMware virtualized environment and a Hadoop distributed file system, achieving as much as orders of magnitude performance gain.

Bill Bolosky (MS Research) was curious to know how block group allocation is done in IceFS and was mainly concerned about whether IceFS really got rid of global metadata. Lanyue

said that each block group is self-defined with its own bitmap and does not share metadata with other block groups. Then Bill asked how block groups themselves are assigned and suggested that there might be some global metadata at a higher level to indicate the allocation status of each block group. Lanyue agreed with him in that for each block group they store a cube ID that needs to be examined when traversing the block groups, but such information is not shared.

Shen Yang (Penn State University) asked how IceFS handled a case when there is a link across cubes—for example, hard links. Lanyue replied that first IceFS doesn't support hard links. And IceFS can detect many other cases of cross reference. When I/O is performed at runtime, IceFS can check whether source and destination belong to the same cube. Another person thought that the practice of isolation was nice, but that performance tweaks might violate POSIX security checks under failures. Lanyue responded that they store the most strict permission at the root directory, which has to be examined to access any subdirectory. This can enforce the original protection. Yang Tang (Columbia University) suggested that ideally he would want a separate cube for each file. He was curious to know whether this would work in IceFS. If not, is it better to just have partitions for complete isolation? Lanyue replied that partitions would lead to wasted space, and if one file system on one of the partitions panics, it might lead to a global crash. Partitions cannot solve the problem of slow recovery either. Finally, in the case of HDFS, it is hard to make partitions on a local node.

Ziling Huang (NetApp) wondered what the performance would be like atop an HDD where jumping across different cubes might incur more disk seek operations. Lanyue confirmed that running Varmail and SQLite on HDD with IceFS would lead to worse performance. Although their system would also work for HDD, it would be more likely to yield better performance on faster devices such as SSDs.

Customizable and Extensible Deployment for Mobile/Cloud Applications

Irene Zhang, Adriana Szekeres, Dana Van Aken, and Isaac Ackerman, University of Washington; Steven D. Gribble, Google and University of Washington; Arvind Krishnamurthy and Henry M. Levy, University of Washington

Modern applications have to handle deploying code across different environments from mobile devices to cloud backends. Such heterogeneity requires application programmers to make numerous distributed deployment decisions, such as how to coordinate data and computation across nodes and platforms, how to hide performance limitations and failures, and how to manage different programming environments and hardware resources. In addition, application programmers have differing requirements: for example, some ask for reliable RPC, while others demand caching, etc. All of these contribute to complicating the development and deployment of applications. Irene Zhang introduced a system called Sapphire, aiming to free application developers from such complex but tedious tasks. Sapphire is a distributed programming platform, which separates application logic from deployment code. Furthermore, it makes it easy to choose and change application deployment.

Sapphire has a hierarchical structure and three layers. The top layer is the distributed Sapphire application. The bottom layer is the deployment kernel (DK), which provides as basic functionality as possible. DK provides only best-effort communication and is not fault-tolerant. The key part of Sapphire architecture is the middle layer, which is a library of deployment managers and offers control over placement, RPC semantics, fault-tolerance, load balancing and scaling, etc.

An important entity in Sapphire is a Sapphire Object (SO). The SO abstraction is key to managing data locality, and provides a unit of distribution for deployment managers (DM). A Sapphire application is composed of one or more SOs in communication with each other using remote procedure calls (RPCs).

Each SO can optionally have an attached DM. Sapphire also provides a DM library. The programmers select a DM to manage each SO, providing features such as failure handling and data cache among many others. Thus, programmers can easily compose and extend DMs to further choose, change, and build deployment.

Kaoutar El Maghraoui (IBM Research) asked how flexible Sapphire is for programmers to specify what kind of deployment they want. In addition, programmers sometimes don't really know the correct deployment requirements for their applications. Irene replied by giving an example of how code offloading can work with the DM. The code offloading DM is adaptive, and it can measure the latency of the RPC to figure out the best place to place the application. Sapphire only asks the programmer to tell whether the piece of code is computationally intensive, and it will do the rest. In contrast, the current practice is either implementing the code twice, once for the mobile side and once for the cloud side, or using some code offloading systems to do pretty complicated code/program analysis to just figure out what portion of the code can or should be partitioned out. Sapphire gets a little bit of information from the application programmer and then does something really powerful.

Howie Huang (George Washington University), asked whether Sapphire also deals with other issues such as security, scalability, and energy consumption, which are important to mobile applications. Irene replied that they haven't looked at energy yet and encouraged the building of a DM that could take energy into account. That would require the DK to monitor energy consumption of the system; right now the DK can only provide latency information. As for privacy and security issues, Irene revealed that they are actually looking at a similar system that provides the user with improved data privacy.

Phil Bernstein (Microsoft Research) asked whether Irene could give the audience an idea of how Sapphire would scale out in a cluster environment, given that the experiment was done on a single server. He noted, in addition, that the DM is centralized

and may become a bottleneck. Irene replied that there are actually some evaluation results in the paper in which they tested scalability with multiple servers and DMs. DMs themselves are not centralized: instead, there is another rarely used but centralized service to track SOs when they move or fail.

Phil then asked about multi-player games with hundreds of thousands of gamers coming and going: Can Sapphire handle the creation and destruction rate scaling up from there? Irene thought it would definitely be a problem if this happens a lot. The other thing she imagined is that Sapphire objects are shared virtual nodes, and most of them are created on cloud servers as files, just like you would for most of the games today.

Pebbles: Fine-Grained Data Management Abstractions for Modern Operating Systems

Riley Spahn and Jonathan Bell, Columbia University; Michael Lee, The University of Texas at Austin; Sravan Bhamidipati, Roxana Geambasu, and Gail Kaiser, Columbia University

Riley Spahn first used an interesting email deletion as an example to motivate this work. In the example, he showed the audience how an email app can perform unexpectedly in managing user data: Although a user may believe that the email has been deleted, it is actually not and is still sitting somewhere in persistent storage and prone to malicious manipulations. Riley pointed out that this is a prevailing problem because a number of popular Android apps suffer from this. Conventional OSes lack the ability to understand high-level user data abstraction and understand plain files instead. Such a mismatch between the OS and the user's perception of data can lead to difficulty in building data protection systems on smartphones. To address this problem, Riley and his colleagues built and proposed Pebbles.

Pebbles is a system-level tool designed to recognize and manage fine-grained user data. It does not require program change, because the authors don't expect application programmers to change their code. This may seem impossible at first glance, but they made the amazing discovery that in real life, user data on smartphones have quite good uniformity, making it feasible to detect.

Logical Data Objects (LDOs) are different from ordinary files in that they can be hierarchical and span multiple data stores: e.g., plain file, key-value store, and SQLite database. To address the aforementioned problem, they made several important assumptions. First, protection tools are trusted. Second, applications which produce LDOs will not maliciously act against Pebbles by manually obfuscating them. Finally, they limit their scope to persistent data, leaving main memory aside. Given these assumptions, they want Pebbles to be able to hide some data, audit access to data, and restrict access to some data.

Pebbles is plugged into Android and modifies Android in three ways: (1) Dalvik with TaintDroid to track dataflows and discover relationships, (2) three modified storage APIs to generate relationships between them, and (3) a new system service called Pebbles Registrar to record all the relationships and create object graphs. This graph of LDOs, or object graph, is the most significant piece of Pebbles since it represents Pebbles' understanding of application data. They used several mechanisms to build the graph, with details presented in the paper. They also built four different applications leveraging the service provided by Pebbles. Evaluation results show that Pebbles is quite accurate in constructing LDOs without supervision. The performance overhead is relatively low, and Pebbles provides reasonably good performance to application users.

Ashvin Goel (University of Toronto) was curious about whether relations other than foreign key relations and file name relationships could be detected. Riley pointed out that basically all relations that can be found are dataflow relationships. By tracking data being written to a certain file that generates a bi-directional relationship because of data sharing, Pebbles could detect a unidirectional relationship from there based on the access. Jonas Wagner (EPFL) commented that many applications want to encrypt their storage. Riley said they didn't evaluate applications that used encryption, although several hundred use a library to encrypt their SQL storage.

My Insecurities

Summarized by Radu Banabic (radu.banabic@epfl.ch) and David Williams-King (dwk@cs.columbia.edu)

Protecting Users by Confining JavaScript with COWL

Deian Stefan and Edward Z. Yang, Stanford University; Petr Marchenko, Google; Alejandro Russo, Chalmers University of Technology; Dave Herman, Mozilla; Brad Karp, University College London; David Mazières, Stanford University

Deian Stefan began by observing that today's Web apps entrust third-party code with the user's sensitive data, leaving browsers to prevent mistreatment of that data. Basic protection is provided by Web browsers' same-origin policy (SOP), where content from different sites is separated into browsing contexts (like tabs and iframes), and scripts can only access data within their own context. But SOP has two problems: (1) it is not strict enough, since a site (or libraries like jQuery) can arbitrarily exfiltrate its data, and (2) it is not flexible enough, because third-party mashup sites are prevented from combining information from multiple source Web sites. So browsers have extended SOP with discretionary access control: The Content Security Policy (CSP) allows a page to communicate with a whitelist of sites, and Cross-Origin Resource Sharing (CORS) allows a server to whitelist sites that can access its data. However, this is still not a satisfactory solution. Taking CORS as an example, if a bank grants access to a mashup site, that site can still do anything with the data (e.g., leak it through buggy or malicious software). So the bank will be unlikely to whitelist such a site, and the mashup may instead fall back on the dangerous practice of requesting the user's bank login credentials.

Deian explained that the challenge addressed by COWL is to allow untrusted third-party code to operate on sensitive data. His motivating example is an untrusted password strength checker. Ideally, the code should be able to fetch lists of common passwords from the Internet to compare against the user's

password, but as soon as the code gains access to the real user's password, it should no longer be able to interact with the network. This is a form of mandatory access control (MAC) known as confinement, for which there exists prior work, but existing confinement systems (Hails, Jif, JSFlow) are overly complex for a Web environment. COWL's design goal is to avoid changing the JavaScript language runtime and Web security model to avoid alienating Web developers.

COWL adds confinement to the existing model of the Web in a natural way. Just as browsers enforce separate execution contexts by different origins (source domains), COWL introduces data protection through labels that specify the origins which care about the data. COWL tracks labels across contexts (iframes, workers, servers). Any context may add a new origin to its label in order to read data labeled with some origin, but then can only communicate with that origin (and cannot communicate at all if its label contains two or more origins). COWL enforces label semantics for outgoing HTTP requests as well as for communication between browser contexts.

For evaluation, Deian mentioned four applications implemented on COWL (including a mashup like mint.com which summarizes banking info). COWL itself is implemented for Firefox and Chromium (by modifying 4k lines of code in each), changing the gecko and blink layout engines, and adding parameters to communications infrastructure like postMessage() and XMLHttpRequest. In terms of performance, there is no additional overhead for sites that do not use COWL (it is only enabled the first time the COWL API is called), while the overhead for the mashup example, excluding network latency, is 16% (16 milliseconds). Deian claimed that COWL can be easily deployed, given its backwards compatibility, reuse of existing Web concepts, and implementation in real Web browsers. One limitation of COWL is that it does not deal with covert channels. In addition, apps must be partially redesigned with compartmentalization in mind (simply adding labels to sensitive variables is insufficient). Some related work includes (1) BFlow, a coarse-grained confinement system for the Web which does not handle the case where two parties mutually distrust each other, and (2) JSFlow, which does fine-grained confinement, is better suited for tightly coupled libraries, and has high overhead (100x). Deian concluded by saying that today we give up privacy for flexibility to allow apps to compute on sensitive data, but the mandatory access control provided by COWL—a natural extension of the existing Web model—allows developers to do better.

The first question was about covert channels: Couldn't information be leaked by sending labeled data to another context and having it respond with one of two messages, leaking one bit of the protected data, and couldn't this process be repeated to leak the entire data? Deian answered that the intent of COWL is to close off all overt communication channels, and while covert channels might still be possible, COWL's approach is better than the current approach where a site is given all-or-nothing access to the data through discretionary access control. Mike Freedman (Princeton) mentioned that mandatory access control systems often have trouble with declassification, and was this ever necessary with COWL, or are browsers essentially stateless? Deian answered that a site can read its own data labeled with its own origin, and this is a sufficient form of declassification. Another attendee asked about the ramifications of defaulting to open access instead of defaulting to closed access before COWL becomes universally deployed. The answer is that a site must opt-in to COWL's mandatory access control by adding a label to some data in order to loosen mechanisms like CORS, and clients that do not support COWL would fall back on the default discretionary access control as deployed today.

Code-Pointer Integrity

Volodymyr Kuznetsov, École Polytechnique Fédérale de Lausanne (EPFL); László Szekeres, Stony Brook University; Mathias Payer, Purdue University; George Candea, École Polytechnique Fédérale de Lausanne (EPFL); R. Sekar, Stony Brook University; Dawn Song, University of California, Berkeley

Volodymyr started by explaining control-flow hijack vulnerabilities: By exploiting a memory safety error in a program, an attacker can overwrite function pointers in program memory and divert the control-flow of the program to execute any code the attacker wants. Despite this being a known problem for 50 years, it is still relevant today; there are more and more control-flow hijack vulnerability reports in the CVE database every year. Code written in high-level languages avoids this problem, but such code often requires millions of lines of C/C++ code to run (language runtimes, native libraries, etc.). There are techniques to retrofit precise memory safety in unsafe languages, but the overhead of such techniques is too high for practical deployment. The control-flow integrity technique provides control-flow hijack protection at lower overhead, but many of control-flow integrity implementations were recently shown to be bypassable.

The authors proposed Code-Pointer Integrity as a technique to eliminate control-flow hijack vulnerabilities from C/C++ programs, while still keeping the overhead low. The key insight is to only protect code pointers in the program; as these are only a minority of all the pointers in the program, the overhead due to the memory safety instrumentation for just these pointers is low.

The implementation of the technique separates memory into two regions: safe and regular memory. The isolation between the two is enforced through instruction-level isolation and type-based static analysis. Instructions that manipulate program data pointers are not allowed to change values in the safe memory region, even if compromised by an attacker. This ensures that attackers will not be able to exploit memory safety errors in order to forge new code pointers. This protection mechanism is called code-pointer separation (CPS). However, this still leaves the potential of an attack, where attackers manipulate pointers that indirectly point to code pointers (such as through a struct) and are thus able to swap valid code pointers in memory, causing a program to call a different function (only a function whose address was previously taken by the program). In order to protect against this type of attack, the authors also propose the code-pointer integrity (CPI) mechanism, which also puts in the safe

memory region all sensitive pointers, i.e., all pointers that are used to indirectly access other sensitive pointers (essentially, the transitive closure of all direct and indirect code pointers). CPI has a bit more overhead than CPS but has guaranteed protection against control-flow hijacks.

In the evaluation, the authors showed that both CPS and CPI protect against all vulnerabilities in the RIPE benchmark and that CPI has a formal proof of correctness. The authors compared CPS and CPI to existing techniques, such as CFI (both coarse- and fine-grained) and ASLR, DEP, and stack cookies, showing that CPS and CPI compare favorably to all: The proposed techniques have either lower overhead, higher guarantees, or both. CPS provides practical protection against all existing control-flow hijack attack and has an average overhead of 0.5%–1.9%, depending on the benchmark, while CPI provides guaranteed protection at an average overhead of 8.4%–10.5%, depending on the benchmark. The stack protection (a component of both CPS and CPI that protects the stack) has an average overhead of 0.03%. The authors released an open-source implementation of their techniques; the protection can be enabled using a single flag in the LLVM compiler.

Jon Howell (MSR) commented that the overhead of CPI is still relatively high, at 10%, but that the attacks against CPS cannot easily be dismissed, since ROP attacks are known. Volodymyr answered that CPS still protects against ROP attacks, since the attacker cannot manipulate the return addresses in any way, and can only change function pointers to point to a function whose address was already taken by the program (and not to any function in a library used by the program). Úlfar Erlingsson (Google) commented that the authors misrepresented prior work. He argued that there is no principled attack against fine-grained CFI and that the cited overheads were true 10 years ago, but a current implementation in GCC has an overhead of 4% (instead of 10% as was cited). Finally, Úlfar asked how the proposed technique is not affected by conversions between pointers and integers, which affected PointGuard several years ago. Volodymyr answered that the analysis handles such cases, and the authors successfully ran the tool on all SPEC benchmarks, which shows the robustness of the analysis.

The next question was about switch statements: Some compilers generate jump tables for such code; is this case handled by the tool? Volodymyr answered that compilers add bound checks for the generated jump table, and they are fully covered by the tool. David Williams-King (Columbia) asked about the 64-bit implementation of the tool, where the lack of hardware support for segmentation forced the authors to use alternative techniques. David asked whether OS or future HW support would help avoid any information leak attacks. Volodymyr answered that the authors have two mechanisms that work on 64-bit, one stronger and one faster. The faster support relies on randomization that is not vulnerable to information leaks, while the stronger approach relies on software fault isolation. Joe Ducek (HP Labs) asked how much of the performance overhead in CPI is due to the imprecision in the analysis and how much to the actual instrumentation. Volodymyr answered that most overhead comes from handling of char* and void* pointers, which in C/C++ are universal, but char* is also used for strings; the tool needs to protect all occurrences of these types of pointers, which leads to the majority of the overhead.

Ironclad Apps: End-to-End Security via Automated Full-System Verification

Chris Hawblitzel, Jon Howell, and Jacob R. Lorch, Microsoft Research; Arjun Narayan, University of Pennsylvania; Bryan Parno, Microsoft Research; Danfeng Zhang, Cornell University; Brian Zill, Microsoft Research

Bryan Parno started by pointing out the very weak guarantees that users have today when submitting private data online. The only guarantees come in the form of a promise from service providers that they will use good security practices, but a single mistake in their software can lead to a data breach. In contrast, Ironclad, the approach proposed by the authors, guarantees that every low-level instruction in the service adheres to a high-level security specification. Ironclad relies on HW support to run the entire software stack in a trusted environment and on software verification to ensure that the software respects a high-level specification. The hardest part is software verification of complex software; in Ironclad, the authors managed to go a step beyond the verification of a kernel (the seL4 work), by verifying an entire software stack with a reasonable amount of effort (without trusting OS, drivers, compiler, runtime, libraries, etc.). To allow this, the authors had to abandon the verification of existing code and rely on developers to specifically write their software with verification in mind.

First, developers write a trusted high-level specification for the application. Then they write an untrusted implementation of the application. Both the specification and implementation are written in a high-level language called Dafny. The implementation looks like an imperative program, except that it has annotations, such as contracts and invariants. The specification is translated by Ironclad to a low-level specification that handles the low-level details of the hardware on which the application will run. Similarly, the implementation is compiled to a low-level assembly language, where both code and annotations handle registers, instead of high-level variables; the compiler can also insert some additional invariants. Finally, a verifier checks whether the low-level implementation matches the low-level specification, and then the app can be stripped of annotations and assembled into an executable.

Bryan gave a live demo of how the system works. The demo showed that Ironclad provides constant, rich feedback to the developer, significantly simplifying the process of writing verifiable code.

The system relies on accurate specifications of the low-level behavior of the hardware that is used. Writing such specifications seems like a daunting task; the Intel manual, for instance, has 3439 pages. The authors bypassed this issue by only specifying a small subset of the instructions, and enforcing the rule that

the system only use those instructions. Similarly, the authors developed specifications for the OS functionality and for libraries. In order to prevent information leaks, Ironclad uses a Declassifier component, which checks whether any data output by the implementation to the outside world (e.g., over the network) would also be output by the abstract representation of the app in its current state.

When discussing the evaluation of Ironclad, Bryan first pointed out that the development overhead for writing a verifiable system wasn't too bad: The authors wrote 4.8 lines of "proof hints" for every 1 line of code in the system. Moreover, part of this effort will be amortized further over time, since a bulk of the proof hints were in reusable components, like the math and standard library. Even as it is now, the ratio of 4.8:1 is significantly better than previously reported ratios of 25:1. In terms of total number of lines of code, the trusted specification of the system has ~3,500 lines of code, split just about evenly between hardware and software, and ~7,000 lines of code of high-level implementation. The latter get compiled automatically to over 41,000 assembly instructions, which means that the ratio of low-level code to high-level spec is 23:1. In terms of performance, initial versions of the implementation were much slower than their non-verifiable counterparts but are amenable to significant manual optimizations; in the case of SHA-256 OpenSSL, the verifiable application is within 30% of the performance of the native, unsafe OpenSSL. The code, specification, and tools of Ironclad will be made available online.

One attendee asked the presenter to say a few words on concurrency. Bryan answered that the authors are currently working on a single processor model; some colleagues are working on multicore, but the state of the art in verification for multicore processors is way behind that for single-threaded programs. The next question was whether the authors have any experience with more complex data structures, such as doubly linked lists. The answer was that the data structures used so far were fairly simple, and most of the time was spent on number-theoretic proofs. Someone from Stanford asked whether the verification could be extended to handle timing-based attacks. Bryan answered that they have not looked into that yet, but there are other groups that are considering timing: for example, the seL4 project. Gernot Heiser (NICTA and UNSW) commented that the entire verification relies on the correctness of the specification and that the authors' approach to ensure correctness is a traditional top-down approach, which is known not to work for real software. He then asked how it is possible to ensure specification correctness for software that uses more modern software development approaches. Bryan answered that there is always a human aspect in the process and that the authors found spec reviews particularly useful. Also, one can pick a particular property of interest, such as information flow, and prove that property against the specification. Finally, Kent Williams-King (University of British Columbia) asked what happens if an annotation is wrong. Bryan replied that the annotations are only used by the verifier as hints. If an annotation is invalid, the verifier will complain to the user and discard the annotation for the rest of the proof.

SHILL: A Secure Shell Scripting Language

Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong, Harvard University

Scott Moore opened by describing how difficult it can be to figure out what a shell script does. Such comprehension requires fully reading the script's code and understanding its execution environment. Yet, going against the Principle of Least Privilege, every script runs with far broader privileges than it needs to do its job. This has security implications since, as in the case of the Shellshock vulnerability, every instance of bash that runs a malicious script can open network connections and execute with the full privileges of its invoking user. Scott then asked two questions: How can the authority of scripts be limited, and how can the authority necessary for a script's operation be determined? To answer these questions, Scott presented Shill, a scripting language where every script comes with its own declarative security policy. In addition to enforcing runtime restrictions, the policy can also be examined by the user to decide whether the script seems safe. Shill scripts can recursively call other Shill scripts if the policy allows, or invoke native executables, which are run inside a sandbox to ensure end-to-end enforcement of the original security policy.

Besides sandboxing, Shill's implementation relies on capabilities, which make the script's authority explicit rather than an artifact of its environment, and contracts, which are the declarations describing how capabilities can be used. Capabilities are an unforgeable token of authority, the possession of which grants the right to perform some action (like keys open doors). This contrasts with existing mandatory access control mechanisms, like UNIX file permissions, which are a property of the environment. There has been a great deal of related work on capabilities. In Shill, functions take capabilities as parameters: files, directories, pipes, sockets are each represented as a capability. Operations like opening or reading a file require privileges on the capability (and "opening" a file in a directory returns a derived capability for the file). All resources are represented as capabilities, and the only capabilities a script has are the ones passed in, making it easy to reason about a script's effects; this is termed "capability safety."

Software contracts in general essentially specify pre- and post-conditions that can be executed to verify that a program runs as it should. In Shill, every function has a grammatically compatible specification written before it; the Shill interpreter checks for contract violations at runtime, and if any are found, terminates the script. The contracts may list the privileges required by the script for each capability (e.g., list files in directory). A callee may assume it has these privileges; the caller can use the privileges to reason about the possible side effects of the call. This aids in reasoning about composing scripts together. This reasoning can extend to any native binaries the Shill script invokes, because Shill sandboxes binaries (without modifying them) to

continue checking for contract violations. In the presentation, Scott gave an example of a sandboxed program opening a file in another directory, and all the privilege checks that occur. Since running most programs requires a large number of (file) capabilities, Shill supports capability wallets, which are composite capabilities necessary to perform common tasks. Bootstrapping—providing the capabilities necessary to run the original script—is achieved with ambient scripts, which are limited and can only create capabilities to pass to Shill scripts.

Shill is implemented in a capability-safe subset of the Racket programming language, and Shill's sandbox is implemented in TrustedBSD's MAC framework. Scott described several case studies for Shill, including an Emacs installer script (which can only create/remove files in certain directories), a modified Apache server (which can only read its configuration and content directories), find-and-grep, and a grading script. In the last case, a TA is tasked with grading programming assignments from many different students using a test suite. The TA might create a script to automate the compiling, running, and test verification process; but it is difficult to ensure that each student's assignment doesn't corrupt other files, leak the test suite, or interact with other students' code. When the script is written in Shill, the security policy allows the TA to reason that the students' assignments will behave themselves, even if the TA's own script is buggy. In terms of performance, Shill generally has less than 20% overhead on these four examples, except find-and-grep which may spawn many sandboxes, leading to 6x overhead. The overhead is proportional to the security guarantees. In summary, Shill allows the Principle of Least Privilege to be applied to scripting, using a combination of capabilities, contracts, and sandboxing. The code and a VM with Shill are available online.

Xu Zhao (University of Toronto) asked about the motivating example of downloading a large untrusted script from the Internet, because such a script might have a very complex security policy. Scott's answer was that with existing scripts, the whole script must be scrutinized along with its execution environment, whereas the security policy provides a single place to focus one's attention. A student-attendee from Columbia University asked why use capabilities instead of access control, and how does Shill compare with SELinux. Scott answered that SELinux creates system-wide policies, while Shill allows more fine-grained control, and Shill turns the question about whether a script is safe to run into a local decision instead of a question about the environment

Brian Ford (Yale) asked about confused deputy attacks on the sandbox, where an incorrect capability is used to gain access to a resource. Scott answered that to mitigate such attacks, components could be rewritten in Shill to leverage its security checks. Stefan Bucur (EPFL) asked about the development time overhead for programmers writing Shill (since many scripting languages are used for quick prototyping). Scott answered that it is similar to writing in Python instead of in bash; one has to think in terms of data types instead of paths. But it is possible to start with broad, permissive security policies and refine them later. Someone from UC San Diego asked whether the authors had applied security policies to existing scripts to see how many misbehave. Scott replied that the closest they got was translating some of their own bash scripts into Shill. Someone from CU Boulder asked about enforcing security policies across multiple machines through ssh. Scott explained that Shill will not make guarantees about the remote machine, but it will control whether a script is allowed to create ssh connections in the first place.

Variety Pack

GPUnet: Networking Abstractions for GPU Programs

Sangman Kim, Seonggu Huh, Yige Hu, Xinya Zhang, and Emmett Witchel, The University of Texas at Austin; Amir Wated and Mark Silberstein, Technion—Israel Institute of Technology

Summarized by Alexander Merritt (merritt.alex@gatech.edu)

Mark Silberstein presented GPUnet, a socket API for GPGPU applications. The aim is to target the development efforts of server-based GPGPU applications: Due to both their complex designs and the burden of manually managing data flow between the GPGPU, network, and host memory, such systems require more time and effort to develop and make efficient. Costs arise from the coordination of data movement between the GPGPU, DRAM, and NIC using only low-level interfaces and abstractions (e.g., CUDA). As is the nature of modern GPGPUs, a host CPU is required to hand off data or work between the GPGPU and other devices, adding complexity and overhead. Mark argues that such a CPU-centric design is not optimal, and that GPGPUs should be viewed as peer processors instead of as co-processors. The lack of I/O abstractions for GPGPU applications, however, makes this challenging.

To avoid costs of data movement and synchronization complexity placed on developers, GPUnet provides a socket API for GPGPU-native applications. Additional contributions include a variety of optimizations enabling GPGPU-based network servers to efficiently manipulate network traffic, and the development and evaluation of three workloads using GPUnet: a face verification server, a GPGPU-based MapReduce application, and a matrix product workload.

Underlying their interface are two example designs that evaluate where one can place the execution of the network stack. The first resembles modern approaches where the CPU processes network packets, utilizing the GPGPU for accelerated parallel processing of received data, and scheduling data movements between the NIC and the GPGPU. A second design exports the network stack to execute natively on the GPGPU itself, where most of the effort involved was in porting the CPU code to the GPGPU. The latter design removes CPU-GPGPU memory copies, as the host CPU can schedule peer-to-peer DMA transfers using NVIDIA's GPUDirect. Their implementation provides two libraries exporting a socket API, one for CPU-based code and the other for GPGPU-based codes.

The design of an in-GPGPU-memory MapReduce application uses GPUfs (prior work) to load image and other data from the host disk to GPGPU-resident memory buffers. Experiments were performed using both one and four GPGPUs each, showing speedups of 3.5x for a K-means workload, and 2.9x for a word-count workload. The latter has a lower performance gain due to its I/O-intensive nature, leaving little room for opportunities from the use of GPUnet. A second application was implemented representing a face verification workload. One machine hosts the client code, a second an instance of memcached, and a third the server implementation. Three implementations for the server were compared: CPU-only, CUDA, and GPUnet. Results show the GPUnet implementation to provide less variable and overall lower response latencies.

Their code has been published on GitHub at https://github.com/ut-osa/gpunet.

Rodrigo Fonseco (Brown University) questioned the use of GPUnet over something like RDMA, which already bypasses the socket interface. Mark responded that there is a usability vs. performance tradeoff, and that they are currently working on a socket-compatible zero-copy API. Pramod Bhatotia (Max Planck Institute for Software Systems) asked for a comparison of this work with the use of GPUDirect. Mark clarified that their work leverages GPUDirect for registering GPGPU memory directly with the InfiniBand network device. Another attendee asked for thoughts on obtaining speedup for the more general case of applications, as opposed to the event-driven, asynchronous workload designs presented. It is a more philosophical discussion, Mark responded; GPUnet gives the freedom to choose where to host the network API. If a workload is parallel and suited for execution on a GPGPU, then you are likely to achieve speedups.

The Mystery Machine: End-to-End Performance Analysis of Large-Scale Internet Services

Michael Chow, University of Michigan; David Meisner, Facebook, Inc.; Jason Flinn, University of Michigan; Daniel Peek, Facebook, Inc.; Thomas F. Wenisch, University of Michigan

Summarized by Yu Luo (jack.luo@mail.utoronto.ca)

Michael Chow presented a technique (the Mystery Machine) to scale familiar performance analysis techniques such as critical path analysis on complex Web sites such as Facebook. The approach to deriving a causal relationship between different components is divided into four steps: identifying segments, inferring a causal model, analyzing individual requests, and aggregating results. Identifying segments refers to coming out with a waterfall diagram of segments executed in a request. Existing logs are aggregated and components are identified. To infer a causal model from the waterfall diagram, we can automatically analyze a large number of traces to find relationships such as happens-before, mutual exclusion, and pipelines. Through the generated causal model, we then apply it to the individual traces. The final step aggregates the results and builds up statistics about the end-to-end system. An earlier method to derive a causal model is through instrumentation of the entire system. Another method is to have every engineer on the team draw up a model of the entire system. Both methods do not scale well. The Mystery Machine applies the four-step approach to provide a scalable performance analysis on large complex Web sites such as Facebook, which allows it to do daily performance refinements.

Greg Hill (Stanford) asked how to deal with clock drifts between machines. Michael answered that there are techniques outlined in the paper. A short answer is that the Mystery Machine assumes the round-trip time (RTT) is symmetric between client and server. It then looks at repeated requests and calculates the clock drift. Someone from the University of Chicago asked how to deal with request failure. Michael answered that this is a natural variation in the server processing time and is taken into consideration. Ryan (University of San Diego) asked how to deal with inaccurate lower level logging messages. Michael replied that we cannot do anything about it.

End-to-End Performance Isolation through Virtual Datacenters

Sebastian Angel, The University of Texas at Austin; Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska, Microsoft Research

Summarized by Alexander Merritt (merritt.alex@gatech.edu)

Sebastian Angel began by stating that tenants are moving away from enterprise datacenters into multi-tenanted cloud datacenters. The key benefits are cost saving and elasticity. Datacenters offer services implemented via appliances. Because these appliances are shared among the tenants, the performance is degraded and unpredictable at times. Aggressive tenants tend to consume a majority of resources. Tenants should get end-to-end guarantees regardless of any bottleneck resources. To achieve the end-to-end guarantees, Sebastian introduced the virtual datacenter (VDC) abstraction and associated virtual capacities. The VDC abstraction is implemented by an architecture called Pulsar. It requires no modification to appliances, switches, guest OSes, and applications. Pulsar can allocate virtual resources based on policies from both tenants and provider. Tenants may specify how VDC resources are divided to VMs. The provider may specify the distribution of resources to maximize profit or to attain fair distribution. The VDC data plane overhead is 2% (15% for small requests) and 256 bytes/sec in the control plane for each VM.

The first questioner (Stanford University) asked how often the application needs to reevaluate relationships between tokens and requests. Sebastian answered that cost functions are fixed. The same person then asked if VDC offers latency guarantees. Sebastian answered that VDC does not offer latency guarantees. Tim Wood (George Washington University) asked how to map performance to tokens. Sebastian answered that tenants can use research tools to take high-level requirements, such as job completion time, and map them to tokens. Henry (Stanford University) asked when there are a lot of short-lived applications, have they considered what would happen during the dip in performance during the initial Pulsar capacity estimation phase?

Sebastian answered that the estimation phase time is configurable and thus the dip in performance can be adjusted.

Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems

Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm, University of Toronto
Summarized by Alexander Merritt (merritt.alex@gatech.edu)

Ding Yuan presented a study of the most common failures found within distributed services. The key finding was that failures are a result of a complex sequence of events between a small number of machines, mostly attributed to incorrect error handling within the source code.

The group developed a tool called Aspirator, a simple rule-based static checker, that uncovered 143 confirmed bugs in systems evaluated by the study, such as HBase, HDFS, and Redis. Ding illustrated the severity of failures in current systems; some can even be catastrophic. As an example, when Amazon's AWS experienced a brief outage, it took down many unrelated Web sites during that time. Why would such services fail like this? They studied end-to-end failure propagation sequences between elements composing such services, as prior studies had examined failures of elements individually in isolation, such as correlations between failures and configuration file mistakes. This study examined interactions between elements themselves.

The findings were very interesting. Starting with 198 user-reported failures, they studied the discussions between developers and users via the mailing lists and were able to reproduce 73. Of these, 48 were catastrophic—those which led to system instability or downtime affecting at least a large majority of users; 92% of the catastrophic failures were a result of largely trivial bugs, such as incorrect error handling of non-fatal errors in Java code (try/catch blocks). An example of such was given in the use of ZooKeeper, where a race condition caused two events to signal the removal of a ZooKeeper node. One attempt resulted in an error, the handling of which led to an abort. Seventy-seven percent of failures required more than one input event, leading to very complex scenarios mostly found on long-running systems. Other interesting results uncovered included: 88% of failures were due to the specific order of events in multi-event sequences, and 26% were non-deterministic (the ZooKeeper example falls into this classification).

By focusing on the 92% of failures that were a result of bad error handling, Ding said, they built a static checker to detect such bugs by feeding it the Java bytecode. Three rules were employed by the checker to signal a possible bug: the error handler was empty, aborted, or had some comment such as "TODO" or "FIXME". The checker was run on nine real-world systems and uncovered a multitude of bugs. Developers gave mixed feedback after having been notified of the group's findings: 17 patches were rejected, but 143 confirmed fixes were adopted. Responses included, "Nobody would have looked at such a hidden feature" and "I fail to see why every exception should be handled." The reason for mixed responses is due to prioritization of developer responsibilities, among other things, such as developers thinking errors will not happen, evolving code, and bad judgment.

Many audience members praised this work prior to asking their questions. A researcher from IBM Research asked whether the problem of ignoring exceptions could be solved through static analysis. Ding asked in return whether she thought she meant to remove the burden of handling all exceptions from the developers. She clarified to mean just the gaps should be filled. Ding responded with skepticism. Error handling is messy, as seen from the examples. Doing so is definitely burdensome for developers, but automating this may lead to even more catastrophic failures. A researcher at NC State asked Ding to explain the 8% of the bugs that were not caused by mishandled exceptions. Ding replied that this 8% represented silent errors not seen by developers. An example in Redis was a failure that resulted from too many file descriptors, a scenario not verified by developers. The researcher followed up with a thought that error masking at the lower levels in the software stack may affect this. Ding suggested that it is hard to suggest more generally which layer in the stack is responsible for handling any given error, except to suggest that an error should be returned to the layer that is most apt to deal with it. He said he was unsure if either silent handling or masking are appropriate techniques in the general case. It might be best to just return the error to the developers, but it is a profound question to which he really can't provide a definite answer. Finally, a researcher (John) asked Ding to compare the tool against something like FindBugs. Ding replied that FindBugs has checks for around 400 scenarios but not for the specific patterns they looked for in this study.

Posters I

Summarized by Alexander Merritt (merritt.alex@gatech.edu)

Unit Testing Framework for Operating System Kernels

Maxwell Walter, Sven Karlsson, Technical University of Denmark

New operating system kernels need to be tested as they are being developed, before hardware becomes available, and across multiple hardware setups. To make this process more streamlined and manageable for kernel developers, Maxwell's proposed system leverages system virtualization with a new testing API. A kernel is booted inside a virtualized environment using QEMU and is presented as virtual hardware configurations, or devices configured as pass-through, e.g., using IOMMUs. A kernel testing API they develop enables a client to use their framework to specify means for creating and executing tests. Capturing state for post-analysis is accomplished via Virtual Machine Introspection (VMI), enabling users to inspect kernel and virtual machine state to locate sources of bugs or race conditions. One limitation is that the virtual implementation of devices and hardware presented to kernels within QEMU behave ideally, unlike real hardware.

Head in the Cloud

Summarized by Alexey Tumanov (atumanov@cmu.edu) and Rohan Gandhi (gandhir@purdue.edu)

Shielding Applications from an Untrusted Cloud with Haven

Andrew Baumann, Marcus Peinado, and Galen Hunt, Microsoft Research

Jay Lepreau Best Paper Award

Andrew Baumann introduced Haven, a prototype to provide shielded execution of an application on an untrusted OS and hardware (except CPU). The motivation for Haven stems from limitations of existing cloud platforms to support trusted computing, where the cloud user has to trust the cloud provider's software, including privileged software (the hypervisor, firmware, etc.), the management stack, which could be malicious, administrative personnel or support staff who can potentially have access to the cloud user's data, and law enforcement. The existing alternatives to this problem are also severely limited for general purpose computing, e.g., hardware security modules (HSMs) that are expensive and have a limited set of APIs.

Haven aims to provide similar guarantees as guarantees provided by "secure collocation of data," where the only way for an outsider to access the data is through the network and not through other hardware and software. In this environment, the cloud provider only provides resources and untrusted I/O channels. Haven ensures confidentiality and integrity of the unmodified application throughout its execution.

Haven makes several key contributions. First, it provides shielded execution using Intel's SGX that offers a process with a secure address space called an "enclave." Intel SGX protects the execution of code in the enclave from malicious code and hardware. SGX was introduced for protecting execution of a small part of the code and not large unmodified applications. Haven extends the original intent of SGX to shield entire applications, which requires Haven to address numerous challenges, including dynamic memory allocation and exception handling. Second, Haven protects the applications from Iago attacks where even the OS can be malicious and the syscalls can provide incorrect results. Haven uses an in-enclave library to address this challenge. Third, Haven presents the limitations of the SGX as well as a small set of suggestions to improve shielded execution.

Haven was evaluated based on the functional emulator, as the authors don't have any access to the current SGX implementation. The authors constructed a model for SGX performance considering TLB flush latencies, variable delay in instruction executions, and a penalty for accessing encrypted memory. In the pessimistic case, Haven can slow down execution by 1.5x to 3x.

John Stalworth asked whether they used a public key for attestation and who owns the key. Andrew replied that Intel provides attestation through a group signature scheme and suggested an Intel workshop paper for details. The owner of the key will be the processor manufacturer (Intel). Nicky Dunworth (UIUC) asked about the programming model and about legacy applications. Andrew again redirected the questioner to the Intel workshop paper with a remark that they still need to support legacy applications due to their large number. Another questioner wondered about Haven's limitations, especially about the memory size/swapping. Andrew said that the size of the memory is fixed, and paging is supported in hardware. John Griswald (University of Rochester) asked about the impact of the cloud provider disabling the SGX. Andrew responded that applications can still run but the attestation will fail.

Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing

Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, and Jingren Zhou, Microsoft; Zhengping Qian, Ming Wu, and Lidong Zhou, Microsoft Research

Eric Boutin introduced Apollo as a cluster scheduler deployed at Microsoft to schedule cloud-scale big data jobs. These jobs are typically specified in a higher-level (SCOPE) SQL-like language that compiles to a DAG of tasks. Apollo sets the goal to minimize job latency while maximizing cluster utilization given the challenges of scale and heterogeneous workloads. Specifically, the target scale of this system is 170k tasks over 20k servers with 100k scheduling requests per second.

First, Apollo adopts a distributed coordinated architecture approach, with one independent scheduler per job making independent decisions but with access to global cluster information, aggregated and provided by the Resource Monitor (RM). Second, Apollo introduces the abstraction of the wait-time matrix, which captures estimated wait time for a given <CPU, Mem> resource request. It masks the underlying heterogeneity of the hardware by grouping servers of the same capacity (e.g., quad-core, 8 GB nodes) and reporting the estimated wait time to acquire that type of server. The wait-time matrix allows Apollo to minimize the total task completion time, simultaneously considering both the queueing delay and the effect of a given node on execution runtime of the scheduled task. The authors refer to this as estimation-based scheduling.

Apollo has the ability to reevaluate prior scheduling decisions. When the scheduler's updated placement decision differs from the one previously made, or there's a conflict, Apollo issues a duplicate task to the more desired server. Lastly, Apollo uses opportunistic scheduling to allow currently running jobs to allocate additional tasks above their allowed quota. This helps Apollo reach their desired goal of maximizing cluster utilization.

Christos Kozyrakis (Stanford) asked about interference of tasks co-located on the same node, sharing cache, disk/flash bandwidth, etc. Christos was specifically interested in whether the problem of interference was measured. The answer was no; Apollo depends on robust local resource management to isolate performance, like JVM and Linux Containers. Henry (Temple University) asked whether the authors considered utilization for other resource types, like disk I/O or memory. Eric replied that the CPU was the primary resource they optimized for in their particular environment. Memory utilization was recognized as important, but no numbers for memory utilization were

published in the paper. Lastly, disk I/O was also recognized as important for good performance, but Christos repeated that Apollo, relied on the local node performance isolation mechanisms "to do their job."

Malte Schwarzkopf (University of Cambridge) pointed out that the formulation used for the wait-time and runtime assumed batch jobs. Related work (Omega) looked at placing other types of jobs, such as service tasks. The question was whether Apollo had support for that or could be extended to support it. At a first approximation, Eric argued that long-running services could be modeled as infinite batch jobs. It would simply block out the corresponding rows and columns of the wait-time matrix. Malte's follow-up concern was that it would not lead to a good schedule, as there are no tradeoffs to be made if tasks are assumed to be running forever. Additionally, the quality of service task placement also varies. Eric responded that Apollo was targeted at cloud-scale big data analytics, with the architecture generally supportive of other schedulers, such as a service scheduler.

Vijay (Twitter) asked about dealing with discrepancies in load across clusters, wondering whether Apollo could make considerations across multiple cluster cells, based on load. The answer was no, as Apollo's workload is such that data locality dominates the decision about which cluster the job would run in.

The Power of Choice in Data-Aware Cluster Scheduling

Shivaram Venkataraman and Aurojit Panda, University of California, Berkeley; Ganesh Ananthanarayanan, Microsoft Research; Michael J. Franklin and Ion Stoica, University of California, Berkeley

Shivaram Venkataraman stated that the context for this work is that the volume of data and jobs that consume it grows, while job latency is expected to drop, down to the near-interactive range of seconds. One specific feature of jobs exploited by this work is the ability to work with a subset or a sample of the data instead of the whole data set. Applications that exhibit these properties include approximate query processing and ML algorithms. The key insight is that the scheduler should be able to leverage the choices afforded by the combinatorial number of *K*-samples out of *N* units of data. In this work, the authors' goal is to build a scheduler that's choice-aware. Shivaram presented KMN Scheduler, which is built to leverage the choices that result from choosing subsets of data blocks to operate on.

Making systems aware of application semantics, finding a way to express application specifics to the scheduler, in other words, is shown to be highly beneficial for this narrow class of applications that benefit from operating on the subset of their data. The authors explore ways to propagate the choices available to the applications to the scheduler and thus leverage the flexibility that is present. Using such a system was shown to improve locality and also balance network transfer, with evaluation evidence that it benefits this emerging class of applications.

Bill Bolosky (Microsoft Research) pointed out that statistical theorems about sampling assume random sampling. Data locality makes that pseudo-random, but particularly concerning is the fact that the data-dependency in the execution time of map tasks coupled with picking first map finishers could really skew results. The authors found no discernible difference between picking the first finishers versus waiting for all map tasks to finish. The data skew is likely the one that exhibits the most amount of determinism. System effects on stragglers are otherwise mostly non-deterministic, as also supported by prior literature. Non-determinism allegedly helps the authors get away from the issues that arise as a result of map task duration skew.

Callas (VMware) was happy to see a DB talk at OSDI. He had a follow-up question about randomness and sampling based on random distributions. The issue is that depending on the partitioning of data across racks (range partitioning) may also skew results. Shivaram pointed out that KMN only counts the number of blocks that are coming from each rack, not their size, which was left for future work. Being agnostic to size gives KMN the advantage that partition size differences do not bear as much of an effect.

Malte Schwarzkopf (University of Cambridge) pointed out that Quincy may yield better results. A major fraction of this work is about trading off locality and minimizing cross-rack transfers. Quincy is actually very closely related to this work, taking the approach of modeling this problem as an optimization problem, instead of using heuristics. Quincy does not support "n choose k" in the general case, because it does not support combinatorial constraints. In the KMN scheduler, however, getting m > k out of n is allowed, and Quincy does have the ability to model this, by carefully assigning increasing costs. The question is how well does Quincy compare to the KMN scheduler, given that Quincy's optimization approach may actually yield better results than KMN Scheduler? Shivaram admitted that the authors haven't tried to apply optimization on top of their solution. No comparison with Quincy was made. The starting point was the "n choose k" property, which was subsequently relaxed to help with cross-rack transfers. This discussion was taken offline.

Heading Off Correlated Failures through Independence-as-a-Service

Ennan Zhai, Yale University; Ruichuan Chen, Bell Labs and Alcatel-Lucent; David Isaac Wolinsky and Bryan Ford, Yale University

Ennan presented work on providing independence-as-a-service. Previously, reliability against failures was provided through redundancy, but seemingly independent systems may share deep and hidden dependencies that may lead to correlated failures. Ennan highlighted multiple examples of correlated failures, e.g., racks connected to the same aggregation switch, the EBS glitch that brought multiple physical machines down in Amazon, the correlated failures across Amazon and Microsoft clouds due to lightning.

The focus of this work is to prevent unexpected co-related failures before they happen. The authors propose INDaaS (independence-as-a-service) that tells which redundancy configurations are most independent. INDaaS calculates an independence score based on the notion of the data-sources (servers/VMs or even

cloud providers) that hold the copy of the data. INDaaS automatically collects dependency data from different data-sources and produces a dependency representation to evaluate the independence of the redundancy configuration. However, collecting data-sources might not be possible across cloud service providers, as cloud service providers won't share all the details about the sources in their infrastructure. To address this, INDaaS offers a privacy preserving approach to privately calculate independence. Each data source can privately calculate the independence score and relay it to the INDaaS.

INDaaS faces several non-trivial challenges including: (1) how to collect dependency data, (2) how to represent collected data, (3) how to efficiently audit the data to calculate independence score, and (4) how to do it privately, when dependency data cannot be obtained. INDaaS calculates dependency data using existing hooks provided by the cloud provider (details in paper). Dependency representation uses fault graphs that consist of DAG and logic gates (AND/OR gates). To efficiently audit the data, INDaaS provides two algorithms with the tradeoff between cost and accuracy. Lastly, INDaaS privately calculates the independence score using Jaccard similarity.

INDaaS was evaluated using (1) case studies based on its deployment at Yale (detailed in paper); (2) efficiency and accuracy tradeoffs between the two algorithms (minimum fault set, failure sampling) using the fat tree network topology, in which the failure sampling algorithm detected important fault sets in 1 million sampling runs in 200 minutes; and (3) network and computation overhead compared to the KS protocol (details in paper).

Mark Lillibridge (HP Labs) asked about other (ill) uses of the system. Amazon can use INDaaS to see what infrastructure and dependencies they have in common with Microsoft. Ennan answered that might be hard since INDaaS only provides the independence score and not any specifics. Someone from Columbia University asked about handling the failure probability of each individual component. Ennan said that in practice it is hard to get correct probability numbers. INDaaS relies on the numbers provided by the cloud service provider. Another questioner wondered how to find configuration that achieves 99.9999% uptime. Ennan noted that it might not be straightforward because INDaaS ranks different configurations based on their independence score.

Storage Runs Hot and Cold

Summarized by Amirsaman Memaripour (amemarip@eng.ucsd.edu) and Haonan Lu (haonanlu@usc.edu)

Characterizing Storage Workloads with Counter Stacks

Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas Harvey, and Andrew Warfield (Coho Data)

Jake started his presentation with a demonstration of storage hierarchy and how it has changed over the years, with the aim of better performance for lower cost. We have been adding more layers to this hierarchy in order to bridge the latency gap between different technologies, making provisioning of storage systems challenging and not-optimized. A major problem in this area is data placement that requires knowledge about future accesses, which is speculated based on previous data access patterns. An example of such speculation techniques is LRU, which tries to move least recently accessed data to lower layers of the storage hierarchy. However, it does not always result in optimum placement decisions. Additionally, its accuracy is time-variant and varies from application to application. Jake then posed the question, "Can we do reuse-distance computing for each request in a more efficient way?" and answered with a "Yes."

He proposed a new data structure, called Counter Stacks, and a set of matrix calculations that will be applied to this structure to compute reuse-distance. The basic idea is to have a good approximation of miss ratio curves with less memory usage. The initial version of the algorithm was quite expensive, so he went through a set of optimizations, including down sampling, pruning, and approximate counting, to make the algorithm run online. He also introduced a method to reduce the memory usage of the algorithm, making it possible to keep traces of three terabytes of memory within a 80 megabytes region. He concluded his talk by going over a list of applications and pointing out that the accuracy of their algorithm is related to the shape of the miss ratio curve.

Michael Condit (NetApp) asked about their memory usage and how they can perform computation while only keeping a portion of the access matrix. Jake pointed out that the algorithm only requires non-zero elements to do the required computations. Scott Kaplan (Amherst) asked about how the proposed method compares to the previous work in this area. Jake pointed out that those methods only maintain workload histories over a short period, not for the entire workload. Consequently, those methods will not be applicable to their cases. Tim Wood (George Washington University) suggested using the elbows in the miss ratio curve to improve the effectiveness of the proposed algorithm.

Pelican: A Building Block for Exa-Scale Cloud Data Storage

Shobana Balakrishnan, Richard Black, Austin Donnelly, Paul England, Adam Glass, Dave Harper, and Sergey Legtchenko, Microsoft Research; Aaron Ogus, Microsoft; Eric Peterson and Antony Rowstron, Microsoft Research

Starting his presentation with a chart comparing different storage technologies in terms of latency and cost, Sergey argued that we need to have a new storage abstraction to efficiently store cold data, which is written once and read rarely. Currently, we are storing cold data on hard disk drives, where we are accustomed to storing warm data. Pelican is going to fill this gap by providing better performance than tape but with similar cost. Their prototype provides 5+ PB of storage connected to two servers with no top-rack switch.

Due to power, cooling, bandwidth, and vibration constraints, only 8% of disks can be active at any given time. In this architecture, disks are grouped into conflict domains where only one domain can be active at a time. Having these conflict domains, they designed data placement mechanisms to maximize concurrency while keeping conflict probability minimized at the same

time. Applying a set of optimizations such as request batching, Pelican can provide a throughput close to having all disks active concurrently. In terms of performance, Pelican consumes 70% less power on average compared to the case that all disks are active concurrently. However, it will add 14.2 seconds overhead for accessing the first byte of an inactive group.

Someone pointed out that there are more recent works that they have not considered in their related works. He mentioned that a startup (Copan Systems) had actually built a similar system a couple of years ago. They decided to take the conversation offline. Someone from Cornell pointed out that most disks die after switching on and off hundreds of times. In response, Sergey mentioned that they have optimized disks for this process but due to confidentiality, he cannot disclose the changes they have made.

A Self-Configurable Geo-Replicated Cloud Storage System

Masoud Saeida Ardekani, INRIA and Sorbonne Universités; Douglas B. Terry, Microsoft Research

Doug Terry presented Tuba, a geo-replicated key-value store that can reconfigure the sets of replicas when facing changes like access rate, user locations, etc., so as to provide better overall services. It's an extension work from Azure with several consistency model choices.

Doug started his presentation with a funny point about the recent shutdown of one of Microsoft Research's labs (in Silicon Valley, where he had been working). He posed the question, "What if someone decides to get rid of the data stored in California without any warning?" which would result in wrong configurations on all other clusters outside California. He proposed a solution based on a configuration service for such situations.

The aim of this service is to choose a better configuration for better overall utility, and to install a new configuration while clients continue reading and writing data. He presented Tuba, which extends Microsoft's Azure Storage and provides a wide range of consistency levels and supports consistency-based SLAs. Tuba's design is based on Pileus and maintains two sets of replicas: primaries and secondaries. Primaries are mutually consistent and completely updated at all times, while secondaries are lazily updated from primary replicas.

Doug then talked about how configuration selection and installation work. For instance, to select a configuration, it takes as input SLAs, read/write rate, latencies, constraints, cost, and the results of a configuration generator module. Applications can declare their acceptable level of consistency or latency and Tuba will generate all possible configurations satisfying the requested service, which is reasonable as the number of datacenters is usually small. Based on constraints defined by the application, such as cost model or data placement policies, the configuration manager will try to pick a configuration and put primary replicas for maximized consistency. Next, it will start moving data based on the new configuration. In this system, clients run in either Slow or Fast mode. Running in the Fast mode, clients read from the best replica and write data to all primaries. Running in the Slow mode, clients do speculation for reading and then check the configuration to make sure data is read from a primary. In order to write data in Slow mode, clients should acquire a lock that guarantees no reconfiguration is in progress. He showed an example to demonstrate how to move the primary datacenter with Tuba.

Doug provided a quick evaluation setup. One cluster in the U.S., one in Europe, and one in Asia, and they used the YCSB benchmark to evaluate their system. He showed that the results on latency and utility were promising, and he also showed that Tuba can increase the overall number of strongly consistent reads by 63%.

A questioner asked about the way that Tuba takes into account future changes in client workloads. Doug mentioned this as the reason that reconfiguration improvements fade after some time. Another reconfiguration can solve the issue and its cost can be amortized.

f4: Facebook's Warm BLOB Storage System

Subramanian Muralidhar, Facebook; Wyatt Lloyd, University of Southern California and Facebook; Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, and Viswanath Sivakumar, Facebook; Linpeng Tang, Princeton University and Facebook; Sanjeev Kumar, Facebook

Sabyasachi Roy presented f4, an efficient warm BLOB storage system. Based on access patterns, warm BLOB content is isolated from hot content and f4 is used to store these contents. By being efficient, f4 lowers effective-replication-factor significantly and also provides fault tolerance in disk, host, rack, and datacenter levels. It's been deployed at Facebook and hosts a large amount of warm BLOB data.

Sabyasachi started his presentation with the definition of BLOB content, mentioning that most of the data stored in Facebook are photos and videos, which are immutable and unstructured. Moreover, these types of data cool down over time, making existing systems like Haystack less efficient for storing them. Basically, they split data into two rough categories, hot and warm. They do replication in various tiers to handle failures of disks, hosts, and racks. In order to make data access and recovery fast, their previous system stores 3.6 bytes for each byte of data. As this level of replication is too much for warm data, they tried to build a system that reduces space without compromising reliability. Using Reed-Solomon error correction coding and a decoder node to retrieve data and handle rack failures, they reduced the replication cost to 2.8x. Additionally, applying XOR on two BLOBs and storing the result in a third datacenter allows them to reduce the overhead down to 2.1x. In production, they consider hot data to be warm after three months or below the querying threshold of 80 reads/sec, then move it to a designated cluster designed for storing warm data.

Doug Terry highlighted the amount of data that would be transferred between datacenters when a failure happens. Sabyasachi mentioned that it would be a very rare event, but it might happen and they have already considered the bandwidth required for such situations. Someone from NetApp mentioned that one

of the charts in the paper does not make sense as 15 disks can saturate 10 Gb connections. They preferred to discuss this question offline. Finally, Jonas Wagner (EPFL) asked how f4 handles deletes. Sabyasachi replied that writes are handled by their old Haystack system, but a different system will take care of deletes.

Pest Control

Summarized by Jonas Wagner (jonas.wagner@epfl.ch)

SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems

Tanakorn Leesatapornwongsa and Mingzhe Hao, University of Chicago; Pallavi Joshi, NEC Labs America; Jeffrey F. Lukman, Surya University; Haryadi S. Gunawi, University of Chicago

Tanakorn Leesatapornwongsa pointed out that serious bugs hide deep within today's distributed systems and are triggered only by combinations of multiple messages and a specific ordering of events. Model checking has been proposed as a systematic solution for bug finding, but it cannot find the critical event interleavings in an exponentially large search space. Tanakorn presented SAMC, a tool that exponentially reduces the size of the search space through semantic knowledge about which event orderings matter for the application.

SAMC's users need to specify a set of rules (~35 lines of code for the protocols in SAMC's evaluation) that describe conditions where event order matters. SAMC evaluates these rules to prune unnecessary interleavings. Experiments show that this leads to speedups of 2–400x compared to state-of-the-art model checkers with partial order reduction. SAMC reproduces known bugs in Cassandra, Hadoop, and ZooKeeper, and also found two previously unknown bugs.

Ivan Beschastnikh (U. of British Columbia) asked whether relying on semantic information could cause bugs to be missed. Tanakorn replied that SAMC could not find those bugs that depended on non-deterministic behavior. SAMC also requires the developer-provided policies to be correct. Jonas Wagner wondered why SAMC did not find more previously unknown bugs. Tanakorn answered that this was because each bug requires a specific initial environment to be triggered, and this needs to be set up before running SAMC.

SKI: Exposing Kernel Concurrency Bugs through Systematic Schedule Exploration

Pedro Fonseca, Max Planck Institute for Software Systems (MPI-SWS); Rodrigo Rodrigues, CITI/NOVA University of Lisbon; Bjørn B. Brandenburg, Max Planck Institute for Software Systems (MPI-SWS)

Concurrency bugs are hard to find and reproduce, because they need specific event interleavings to be triggered. Existing tools can explore possible interleavings for user-mode programs, but not in the kernel. Pedro Fonseca presented SKI, the first systematic approach for finding concurrency bugs in unmodified OS kernels. It runs the kernel and guest applications in a modified VMM, where each thread is pinned to a virtual CPU. By throttling these CPUs, SKI can exercise a diverse range of schedules.

SKI detects which CPUs/threads are schedulable by analyzing their instruction stream and memory access patterns. It uses the PCT algorithm (ASPLOS 2010) to assign priorities to CPUs and systematically explore schedules. A number of heuristics and optimizations speed this up and are described in the paper. SKI supports several existing bug detectors to find data races, crashes, assertion violations, or semantic errors such as disk corruption.

SKI's authors used it to successfully reproduce four known bugs on several different kernels. This only takes seconds, because SKI explores 169k–500k schedules per second. SKI also found 11 new concurrency bugs in Linux file system implementations.

Stefan Bucur (EPFL) asked whether SKI could detect deadlocks. Pedro replied that they did not try this, although it is supported in a number of OSes that run on top of SKI. Bucur also asked how the effort of building SKI compares to the effort needed to instrument the kernel scheduler. Pedro pointed to related work, DataCollider, that, like SKI, avoided modifying the kernel because this was presumed to be very complicated. Srivatsa Bhat (MIT) asked about the maximum number of threads, to which Pedro replied that the number of virtual CPUs in QEMU (and thus SKI) is not limited.

All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications

Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, University of Wisconsin—Madison

Many applications, like databases and version control systems, employ techniques such as journaling, copy-on-write, or soft updates to keep their data consistent in the presence of system crashes. Yet their implementation is often incorrect. Thanumalayan Sankaranarayana Pillai identified file systems as the main cause for this, because they provide very weak guarantees that differ subtly for various configuration settings. He presented a study on file system guarantees and how these are used by applications.

Thanumalayan's tool, BOB (Block-Order Breaker), stress-tests a file system to produce an Abstract Persistence Model (APM), a compact representation of all the guarantees that have been observed not to hold. The companion tool ALICE (Application-Level Intelligent Crash Explorer) runs a user-provided workload, collects a system call trace, and uses the APM to enumerate possible intermediate states at various instants in this trace. If one of these states leads to inconsistent data after crash recovery, a bug has been found.

Alice found 60 such bugs, many of which lead to data loss. About half of these vulnerabilities are possible even for common file system configurations like btrfs.

Stephane Belmon (Google) asked what it meant for a "rename" system call to not be atomic. Thanumalayan explained that one can end up in a state where the destination file is already deleted but the source file is still present. Geoff Kuenning (Harvey Mudd College) asked for ideas for a better API that would make consistency easier for developers. Thanumalayan said related work had

attempted this, but no API other than POSIX is actually being used. POSIX is hard to use because there is no good description of the possible states after an API call. Emery Berger (Amherst) compared the situation to weak memory models. He asked if this is an API issue or a file system issue. Thanumalayan believes it's a combination of both. The current API is comparable to assembly language; a higher-level API would help. David Holland (Harvard) asked whether databases were more robust than other types of applications. Thanumalayan affirmed and said that databases in general fared better than version control systems.

Torturing Databases for Fun and Profit

Mai Zheng, Ohio State University; Joseph Tucek, HP Labs; Dachuan Huang and Feng Qin, Ohio State University; Mark Lillibridge, Elizabeth S. Yang, and Bill W Zhao, HP Labs; Shashank Singh, Ohio State University

Mai Zheng presented a system to test the ACID properties that we expect our database systems to provide. He showed that all examined database systems fail to guarantee these properties under some cases. At the heart of his work is a high-fidelity testing infrastructure that enforces a simple fault model: clean termination of the I/O stream at a block boundary. Because this model is simple (it does not consider packet reordering or corruptions), the errors it exposes are very relevant.

The system generates a workload that stresses a specific ACID property. It records the resulting disk I/O at the iSCSI interface level. It then truncates the iSCSI command stream at various instants to simulate an outage, performs database recovery, and scans the result for violations of an ACID property. The system augments the iSCSI command trace with timestamps, system calls, file names etc., and it uses this information to select fault points that are likely to lead to ACID violations. Such points are tried first to increase the rate at which problems are found. Once a bug is found, a delta debugging approach minimizes the command trace to narrow down the root cause of the problem.

The system was applied to four commercial and four open-source databases running on three file systems and four operating systems. It found ACID violations in all of them, especially durability violations. The speedups from using pattern-based fault point selection were instrumental for finding some of these bugs.

The first question concerned the configuration of the tested databases. Mai Zheng said that, whenever they were aware of options, his team configured the databases for maximum correctness. When asked why the work found mostly durability violations and few isolation violations, Mai Zheng explained that, although their workloads were designed to catch all types of violations, it is possible that isolation violations went undetected. Philip Bernstein (Microsoft Research) asked how likely the found bugs were to happen in practice. Mai Zheng replied that, in their traces, about 10–20% of the fault points led to a bug.

Award Announcements

Summarized by Rik Farrow (rik@usenix.org)

Rather than attempt to announce more awards while people were conversing during an outdoor luncheon, annual awards, as opposed to the ones specific to this particular OSDI, were announced before the afternoon break. Mona Attariyan (University of Michigan) received the SIGOPS Dennis Ritchie Doctoral Dissertation Award for her work on improving the troubleshooting and management of complex software. An ACM DMC Doctoral Dissertation Award went to Austin Clements of MIT for his work on improving database performance on multicore systems.

Steven Hand said that there would be no SIGOPS Hall of Fame awards this year. Instead, a committee will elect a large number of papers into the Hall of Fame at SOSP in 2015. Franz Kaashoek and Hank Levy will be committee chairs. At OSDI '16, they will focus on papers from 10–11 years previous to make things simpler. SOSP 2015 will be in Monterey and include an extra day for history. The first SOSP was in 1965.

Eddie Kohler won the Weiser Award for his prolific, impactful work on Asbestos, routing, and performing improvements in multicore databases among other things. Mark Weiser had asked that people make their code available, and Eddie has certainly done this, said Stefan Savage, who presented the award. Eddie also maintains HotCRP. Kohler, who wasn't present, had prepared a short video in which he said that what might not be obvious to people who know how cranky he is is that this community means a lot to him. Kohler thanked various people and the places where he had worked, like MIT, ICSI (Sally Ford), UCSD, Meraki (please use their systems). Kohler, now at Harvard, thanked everybody there, including Franz Kazakh, whom he described as close to an overlord, adding that it's lucky that he is so benevolent. Kohler ended by saying there are "a lot of white men on the Weiser award list. I am perhaps the only one that is gay. I hope we get a more diverse group of winners for the Weiser award."

Transaction Action

Summarized by Mainak Ghosh (mghosh4@illinois.edu)

Fast Databases with Fast Durability and Recovery through Multicore Parallelism

Wenting Zheng and Stephen Tu, Massachusetts Institute of Technology; Eddie Kohler, Harvard University; Barbara Liskov, Massachusetts Institute of Technology

Wenting motivated the problem by pointing to the popularity of in-memory databases due to their low latency. Unfortunately, they are not durable. Thus the goal of the work was to make an in-memory database durable with little impact on runtime throughput and latency. In addition, failure recovery should be fast. Wenting identified interference from ongoing transactions and slow serial disk-based recovery techniques as a major challenge. She proposed SiloR, which builds on top of Silo (a high performance in-memory database) and described how it achieves durability using logging, checkpointing, and fast disk recovery.

To make logging and checkpointing fast, SiloR uses multiple disks and multiple threads to parallel write. Wenting pointed out that SiloR logs values as opposed to operations because it enables parallel writes. SiloR also facilitates fast recovery because the need to preserve order among different log versions in operation logging is obviated. Recovery can be done in parallel as well. In the evaluation, Wenting showed the achieved throughput for SiloR to be less than vanilla Silo since some cores are dedicated for persistence. Checkpointing adds minimal overhead to the system. Recovery is also fast because SiloR can consume gigabytes of log and checkpoint data to recover a database in a few minutes.

Mark Lillibridge (HP Labs) asked about SiloR's performance if the system replicates logs and checkpoints. Wenting replied by admitting that replication is something that they are hoping to address in the future. Brad Morrey (HP Labs) asked about the lower per-core performance of SiloR in comparison to Silo. Wenting pointed out that SiloR does some additional work during logging and checkpointing which creates that difference. Brad's second question was about bottleneck during recovery. Wenting replied that SiloR tries to avoid being I/O bound by having multiple disk-based architecture.

Salt: Combining ACID and BASE in a Distributed Database

Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan, University of Texas at Austin

Chao started his talk by pointing out how transaction abstraction eases programming and is easy to reason about in ACID databases. Unfortunately, they are slow because providing isolation requires concurrency-control techniques like locks. In a distributed setting, complex protocols like 2PC make it even worse. The alternative, BASE, which does away with transactions for weaker forms of consistency, provides performance at the cost of code complexity. To bridge this gap, Chao proposed a new abstraction, Salt, which provides the best of both worlds. At this point, Chao made a key observation: following the Pareto principle, in a modern day application only a small set of transactions lead to performance limitation. This is because many transactions are not run frequently and a lot of them are lightweight. Their solution tries to BASE-ify this small set after identifying them.

Chao discussed the tradeoff between performance and complexity by using a bank transfer balance as an example. Splitting the transfer balance transaction such that deduction is in one half and addition is in another will lead to an inconsistent state being exposed for any transaction that tries to read the balance after the first transaction completes but before the second one begins. Since these transactions are critical, parallelizing them will lead to a lot of gains. Chao proposed BASE transactions, which consist of smaller alkaline transactions. Alkaline transactions can interleave with other alkaline transactions, but an ACID transaction cannot. This guarantees that the critical transaction provides the performance improvement without exposing inconsistent state to other transactions. The multiple granularities are provided by Salt isolation. Chao introduced three types of locks: ACID, alkaline, and saline, which together provide the Salt isolation.

For evaluation, the whole abstraction was implemented on top of a MySQL cluster, and three workloads were used. Chao reported a 6.5x improvement in transaction throughput with just a single transaction BASE-ified. Thus, their goal for achieving performance with minimal effort while ensuring developer ease was met.

Marcos (Microsoft Research) asked about guidelines for developers on which transaction to BASE-ify. To identify a long-running, high-contention transaction, Chao proposed running the transaction with larger workloads and spot those whose latency increases. Marcos followed up by asking how to ensure BASE-ifying a transaction will not affect invariants like replication. To that Chao put the responsibility on the developer for ensuring this consistency. Henry (Stanford University) sought a clarification on the "Performance Gain" slide datapoints. Chao said it represented number of clients. Dave Andersen (CMU) asked about the future of their solution. Chao said he was very optimistic.

Play It Again, Sam

Summarized by Lucian Carata (lucian.carata@cl.cam.ac.uk)

Eidetic Systems

David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen, University of Michigan

David Devecsery introduced the concept of eidetic systems: systems that can "remember" all the computations they have performed (recording things like past memory states, network inputs and communications between processes), together with the relationships between the different pieces of data involved in those computations (provenance, or how data came to be—i.e., what inputs or data sources were involved in creating a given output).

David presented Arnold, an implementation of such a system based on deterministic record and replay techniques. During normal process execution, Arnold records sufficient information to allow for later replay at the granularity of a replay group (where a group is the set of threads/processes that share memory). At the same time, it maintains a dependency graph of inter-group data flows in order to support the tracking of provenance across replay groups. The more expensive operation of tracking the provenance of a piece of data within a group is left for the replay stage, when processes are instrumented using PIN to perform fine-grained taint tracking.

To reduce the storage overhead of recorded data, Arnold employs multiple data reduction and compression techniques (model-based compression, semi-deterministic time recording, and gzip). With those in place, the storage overhead for the typical utilization scenario (desktop or workstation machines) is predicted to be below 4 TB for four years.

Two motivating use cases were detailed, the first referring to tracking what data might have been leaked on a system with the

Heartbleed vulnerability, and the second covering the backward tracing for the source of an incorrect bibliographical citation (through the PDF viewer, LaTeX/BiBteX processes and eventually to the browser window from where the data was copied). Forward tracing from the point of the mistake (in what other places was this wrong citation used?) is also possible using the same underlying mechanisms.

Fred Douglis (EMC) asked whether the system does anything to avoid duplication of data, such as copy-on-write techniques. David answered affirmatively. Arnold employs a copy-on-read-after-write optimization. As a follow-up, Fred asked whether the replay continues to work if the original inputs to a process are deleted. David replied that Arnold will continue to store those inputs for replay, employing deduplication and caches to reduce overheads. Ethan Miller (UC Santa Cruz) asked what happens when Arnold encounters programs that exhibit inherent randomness. David answered that they haven't found lots of programs with this behavior, but that such randomness would be treated as any other non-deterministic input that needs to be recorded in order to assure correct replay. Someone asked what happens if the users don't want provenance tracking for some pieces of data. David noted that Arnold itself can be used to determine all the places where a piece of data was used—and subsequently use that information to remove any trace of that data. Gilles Muller (INRIA) asked whether temporary files are kept, whether they are useful or a problem when trying to understand the source of information. David answered that temporary files are something like intermediate states (written out and read later). So Arnold will cache them in the file cache or regenerate them if needed.

Detecting Covert Timing Channels with Time-Deterministic Replay

Ang Chen, University of Pennsylvania; W. Brad Moore, Georgetown University; Hanjun Xiao, Andreas Haeberlen, and Linh Thi Xuan Phan, University of Pennsylvania; Micah Sherr and Wenchao Zhou, Georgetown University

Ang Chen presented the general idea behind covert timing channels, with an example of a compromised application sending normal packets over the network but encoding some sensitive data in the timing of those packets. The motivation behind the work in the paper is that existing state-of-art systems for detecting such covert timing channels look for specific statistical deviations in timing. However, those can be circumvented by attackers creating new encoding schemes or by altering the timing of just one packet in the whole encoding so that no statistical deviation exists.

The proposed solution relies on determining the expected timing of events and then detecting deviations from it. The insight is that instead of predicting the expected timing (a hard problem), one can just reproduce it using record and replay techniques: recording the inputs to an application on one machine and replaying them on a different one.

However, Ang explained that existing deterministic replay systems are not sufficient for the stated purpose, as they reproduce functional behavior of an application, but not its timing behavior (e.g., XenTT shows large time differences between actual program execution and replay). In this context, time-deterministic replay is needed. To achieve this, various sources of "time noise" must be handled.

During the presentation, the focus was placed on time noise generated by different memory allocations and cache behavior. The solution presented for that problem aims to maintain the same access patterns across record and replay. This is achieved by flushing caches before record and replay and by managing all memory allocations to place all variables in the same locations in both cases.

A prototype of time-deterministic replay, Sanity, was implemented as a combination of a Java VM with limited features and a Linux kernel module. The evaluation of this prototype shows that Sanity managed to achieve a very small timing variance across multiple runs of a computation-intensive benchmark (max 1.2%) and provided accurate timing reproduction (largest timing deviation of 1.9%) on a separate workload. Sanity was also able to detect timing channels without any false positives or false negatives.

John Howell (Microsoft Research) asked about the scope of attacks being considered, and in particular, whether an attack payload present in the input stream wouldn't still be replayed by Sanity on the reference machine. Ang answered that for the replay, the system has to assume that the implementation of the system is correct. One of the creators of XenTT noted that XenTT provided a fairly accurate time model (in the microsecond range): Was that model used for the XenTT results? Ang answered that the same type of replay was used, and the results were the ones shown during the presentation. Gernot Heiser (University of New South Wales and NICTA) questioned the feasibility of the given solution in realistic scenarios since it relies on the existence of a reference (trusted) machine that is identical to the machine running the normal application. In this case, the application could be run directly on the trusted machine. Ang acknowledged that the current solution requires a correct software implementation and an identical machine but pointed out that there are still multiple situations where the solution might be feasible and where such identical machines exist (e.g., a datacenter).

Identifying Information Disclosure in Web Applications with Retroactive Auditing

Haogang Chen, Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek, MIT CSAIL

Haogang Chen started his presentation by highlighting the recurrent cases of data breaches leading to user data being compromised as a result of Web site vulnerabilities. Multiple solutions deal with preventing such breaches, but not with damage control solutions after a breach occurs.

However, Haogang observed that even if vulnerabilities exist, they might not be exploited, or the attackers might not actually steal all the data they obtain access to. Therefore, an important goal would be to precisely identify breached data items.

The state-of-the-art in this respect requires logging all accesses to sensitive data, and careful inspection after an intrusion, but this is often impractical. An alternative is presented in Rail, a system that aims to identify previously breached data after a vulnerability is fixed. The insight used is that Web application requests/responses can be recorded during normal execution and then replayed after an administrator fixes a vulnerability. The difference in data being sent can be attributed to the information that was leaked due to that vulnerability.

Compared to other record/replay systems, the challenge in implementing Rail is minimizing the state divergence on replay as that might lead to the reporting of false positives. The proposed solution assumes that the software stack below the application is trusted, and consists of an API for Web application developers facilitating deterministic record/replay and data identification at the object level.

The design revolves around the notion of action history graphs: An action is generated for each external application event (e.g., user request, timer), and all application code triggered by that event is executed in the context of the action. Any objects used in that code are connected to the action, resulting in a history graph. This is then used to replay each action in time order, whenever one of its inputs/outputs has changed.

Haogang also discussed the case of replay in the presence of application code changes and non-deterministic inputs. The chosen example involved changes in the components of an array (list of admins), which in turn invalidated replay data associated with particular indexes in the array (e.g., password assignments). Rail provides an input context object that can be used for associating such data with particular stable object keys.

In the evaluation of Rails, Haogang highlighted that it performed better than log-inspection approaches, giving no false negatives and (for the tested workloads) only one false positive, the result of a malicious account created by the attacker. In terms of replay, Rails needed to do so only for the fraction of requests related to the attack (max 10.7%). Overall throughput overhead varied between 5% and 22%.

Stefan (Google) raised the issue of the threat model being considered; in particular, the fact that the application must trust the Web framework and other software stack components. If one manages to inject random code into the process, logging might be bypassed. Haogang acknowledged that that is the case, as the assumption is that the framework is trusted.

Help Me Learn

Summarized by Xu Zhao (nuk.zhao@mail.utoronto.ca) and Sankaranarayana Pillai (madthanu@gmail.com)

Building an Efficient and Scalable Deep Learning Training System

Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, Karthik Kalyanaraman, Microsoft Research

Trishul began by introducing machine learning and deep learning. Deep learning differs from other kinds of machine learning by not requiring a human to extract the features of the training data. Deep learning can automatically learn complex representations (without requiring humans); an example utility is computer vision. Deep learning algorithms can be thought of as a network of multiple levels of neurons that work level-by-level. For example, in computer vision, initial levels identify simple representations such as color and edges, while higher levels automatically learn complex representations such as edges and textures. The accuracy of deep learning can be improved by increasing the size of the model (such as the number of levels in the deep learning network) and by increasing the amount of data used for training the model. Both of these require better, scalable systems.

The authors proposed Adam, a scalable deep learning system. Adam contains three parts: a data server (that supplies data to the models), a model training system (where multiple models learn), and a model parameter server (where all models store their learned weights). The overall design aims at data parallelism and model parallelism: a large data set is divided, and each part is simultaneously used for training, with multiple models also trained at the same time. Adam uses an asynchronized weight update technique, where learned weights are propagated to the model-parameter server slowly (the weight update operation is both commutative and associative). To make the models distributed, Adam partitions the models to fit a single machine; the working version of the model is fit into the L3 cache, so that memory is not a bottleneck. Furthermore, Adam optimizes communication with the model-parameter server by asynchronous batching.

The authors evaluated the accuracy of Adam using MNIST as a baseline. By turning on asynchronization, Adam gets tremendous improvement; asynchronization can help the system jump out of local minimum. On a computer vision task, Adam has twice the accuracy of the world's best photograph classifier, because it can use bigger models and a larger training data set.

John Ousterhout (Stanford) asked about the size of the model in terms of bytes and how much data is shared among how many models. Trishul answered that, usually, 20 to 30 models share terabytes of data. A student from Rice University asked how hardware improvements can help (can bigger models be combated with bigger hardware?). Trishul answered that bigger models are harder to train; hence, hardware improvement does not simply solve the problem. Another question concerned the number of machines assigned for the model replica and how a replica fits into the L3 cache. Trishul answered that there are

four machines assigned to model replicas and only the working set of the model needs to fit into the L3 cache.

Scaling Distributed Machine Learning with the Parameter Server

Mu Li, Carnegie Mellon University and Baidu; David G. Andersen and Jun Woo Park, Carnegie Mellon University; Alexander J. Smola, Carnegie Mellon University and Google, Inc.; Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su, Google, Inc.

Mu began by showing users a real-world example of machine learning: ad-click analysis, similar to Google's advertisements on its search results. While machine-learning accuracy can be improved by increasing the size of the model and by increasing the amount of data, dealing with huge model sizes and data requires distributing the work among multiple machines.

In the presented system, the training data is fit into worker machines, while the model is fit into server machines. Worker machines compute gradients and push them to the servers, which compute and update the model; the model is then pulled by the worker machines. The authors found that accessing the shared model is costly because of barriers and the network communication. To reduce the cost, the authors introduce the concept of a Task. Each Task contains separate CPU-intensive and network-intensive stages, and performance can be increased by running the stages asynchronously. The system also allows users to trade off consistency for reduced network traffic; waiting time is eliminated by relaxing consistency requirements. Mu also briefly talked about other features like user-defined filters, explained further in the paper. Apart from performance, the system achieves practical fault tolerance using consistent hashing on model partitions. By only replicating the aggregating gradient, the system reduces network traffic (while, however, introducing CPU overhead). Also, the system exposes the output using a key-value API.

For the evaluation, the authors ran sparse logistic regression with 636 terabytes of real data on 1,000 machines with 16,000 cores in total. The system outperformed two baselines; furthermore, the waiting time of training can be eliminated by relaxing the consistency requirement. Mu also presented the result of running another application, Topic Model LDA: increasing the number of machines from 1,000 to 6,000 provided a 4x speedup. Finally, Mu showed the results with 104 cores; the key here is the tradeoff between network communication and consistency.

A student from NYU asked about more quantitative details of the tradeoff between consistency and accuracy. Mu answered that it really depends on the model and the training algorithm, and that he authored an NIPS paper showing the theoretical upper-bound. Kimberly Keeton (HP Labs) asked why the authors chose Boundary Delay instead of other consistency models; Mu answered that Boundary Delay was just one of the consistency models they used; they actually used different consistency models for different applications. A questioner from Microsoft Research asked what accuracy meant in the y-axis of the accuracy graph; Mu answered that when the model gets the accuracy quantity needed, they will stop training.

GraphX: Graph Processing in a Distributed Dataflow Framework

Joseph E. Gonzalez, University of California, Berkeley; Reynold S. Xin, University of California, Berkeley and Databricks; Ankur Dave, Daniel Crankshaw, and Michael J. Franklin, University of California, Berkeley; Ion Stoica, University of California, Berkeley and Databricks

Joseph began the talk by explaining how, in modern machine learning, combining two representations of the data, tables and graphs, is difficult. For tables, there are already many existing solutions, like Hadoop and Spark. For graphs, we have GraphLab and Apache Graph. However, users have to learn both table and graph solutions, and migrating data between them is difficult. GraphX unifies tables and graphs; the authors show that there is a performance gap between Hadoop/Spark and GraphX, indicating GraphX is really needed.

Joseph showed how GraphX converts graphs into table representation, and how it represents graph operations (like gather and scatter) into table operations (like Triplet and mrTriplet). The authors did many optimizations on the system, such as remote caching, local aggregation, join elimination, and active set tracking.

The authors evaluated GraphX by calculating connected components on the Twitter-following graph; with active vertex tracking, GraphX got better performance, while with join elimination, GraphX decreased data transmission in the combining stage. The evaluation was done by comparing the performance between GraphX, GraphLab, Giraph, and naive Spark. GraphX is comparable to state-of-the-art graph-processing systems.

Greg Hill (Stanford University) asked how a graph can be updated in GraphX; Joseph answered that, currently, GraphX doesn't support updating. Schwarzkopf (Cambridge) asked why there are no evaluations and comparisons between GraphX and Naiad; Joseph answered that the authors found it difficult to express application details in Naiad. The third questioner asked whether ideas in GraphX can be backported into existing systems; Joseph answered that some techniques can be backported, but many techniques are tied to GraphX's particular design.

Hammers and Saws

Summarized by Ioan Stefanovici (ioan@cs.toronto.edu)

Nail: A Practical Tool for Parsing and Generating Data Formats

Julian Bangert and Nickolai Zeldovich, MIT CSAIL

Julian motivated Nail by describing the problems of binary data parsers today: Most of them are handwritten and highly error-prone (indeed, a number of recent parsing bugs generated high-profile security vulnerabilities in SSL certification, and code signing on iOS and Android applications). One option is to employ parser generators like Bison to generate a parser, but this would still involve extra handwritten code by the programmer to manipulate the parsed data in the application, and output it back to the binary format. In addition, this approach cannot handle non-linear data formats (such as ZIP archives). With Nail, programmers write a single grammar that specifies both the format of the data and the C data type to represent it, and Nail will cre-

ate a parser, associated C structure definitions, and a generator (to turn the parsed data back into a string of bytes).

The Nail grammar supports standard data properties (size, value constraints, etc.), but it also reduces redundancy introduced by having multiple copies of the same data by including the notion of dependent fields (values that depend on other values). Non-linear parsing (e.g., parsing a ZIP archive backwards from the header field) is supported using "streams": multiple paths that can each be parsed linearly. For unforeseen stream encodings (e.g., parsing dependent on arbitrary offset and size fields), Nail provides a plugin interface for arbitrary programmer code (which would be much smaller than an entire parser). Output generation back to binary format is not a pure bijection but, rather, preserves only the semantics of the specification, allowing Nail to discard things like padding and other redundant data.

Nail is implemented for C data types. The code generator is implemented using Nail itself (100 lines of Nail + 1800 lines of C++). To evaluate Nail's ability to handle real formats, Julian implemented various grammars: Ethernet stack (supporting UDP, ARP, ICMP), DNS packets, and ZIP archives. In all cases, the implementation with Nail consisted of many fewer lines of code than a handwritten alternative, and captured all the complexities of each respective data format. A Nail-generated DNS server also outperformed the Bind 9 DNS server in queries/sec. Nail builds on previous work in the area (e.g., the basic parsing algorithm is the Packrat algorithm described in Bryan Ford's MSc thesis).

Eddie Kohler (Harvard University) remarked that memory usage is a known disadvantage of Packrat parsers, and asked whether that was a valid reason to continue using handwritten parsers instead. Julian replied that for most parsing that does not involve backtracking (such as DNS packet parsing), memory usage and performance is not a concern.

lprof: A Non-Intrusive Request Flow Profiler for Distributed Systems

Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm, University of Toronto

Yongle opened by discussing a critical problem of distributed systems: performance anomalies. Such anomalies increase user latency and are very hard to understand and diagnose, since they typically involve understanding the flow of a request across many nodes in a distributed system. Existing solutions for distributed request profiling (MagPie, X-Trace, Dapper, etc.) typically involve intrusive, system-specific instrumentation. A key observation is that most distributed systems today already generate TBs of log data per day, much of which consists of information about the flow of requests through the system (since developers rely on this information for post-mortem, manual debugging). lprof is a non-intrusive profiler that infers request control flow by combining information generated from static analysis of source code with parsing of runtime-generated system logs. Yongle presented a sample "latency over time" graph generated by lprof that showed unusually high latency for a writeBlock request in HDFS. Combined with per-node latency information (also generated by lprof), the problem can conclusively be attributed to unnecessary network communication.

lprof generates a model by performing static analysis on application byte code. This model is then used while performing log analysis at runtime (using a MapReduce job) to profile request flow and save the information into a database (e.g., for use later in visualization). Challenges involved in log analysis include: interleaved messages from different request types, lack of perfect request identifiers, and log entries generated by the same request spread across several machines. In order to trace the flow of a request, lprof needs to identify a top-level method (that starts to process the request) and a request identifier (that is not modified during the request) and maintain log temporal order. The model generated by static byte code analysis is used during the log analysis to solve all these problems. The key intuition is that unique request identifiers are already included by developers in logs for manual post-mortem debugging. Cross-node communication pairs are identified as socket or RPC serialize/deserialize methods. Temporal order is inferred by comparing the output generated for a specific request with the order of the corresponding source code that generated it.

lprof was evaluated on logs from HDFS, Yarn, HBase, and Cassandra using HiBench and YCSB as workloads. lprof grouped 90.4% of log messages correctly; 5.7% of messages involved request identifiers that were too complicated for lprof to handle, while 3% of logs could not be parsed, and 1% of log messages were incorrectly grouped. lprof was also helpful in identifying the root cause of 65% of 23 real-world performance anomalies (the 35% where it was not helpful was due to insufficient log messages).

Rodrigo Fonseca (Brown University) asked what developers could add to their logging messages to help lprof work better. Yongle replied that better request identifiers would help. Rodrigo further asked how this could be extended to combine information across multiple applications. This remains as future work. A developer of HDFS mentioned that they log per-node metrics (e.g., bytes processed/sec) at runtime, and was wondering how lprof can be used with this information for performance debugging. Yongle replied that lprof would provide the cross-node request tracing, while the per-node metrics could complementarily be used for finer-grained debugging. Someone from North Carolina State University wondered how block replication in HDFS would affect the analysis performed by lprof, but the discussion was taken offline.

Pydron: Semi-Automatic Parallelization for Multi-Core and the Cloud

Stefan C. Müller, ETH Zürich and University of Applied Sciences Northwestern Switzerland; Gustavo Alonso and Adam Amara, ETH Zürich; André Csillaghy, University of Applied Sciences Northwestern Switzerland

Stefan began by describing the motivation for their work: large-scale astronomy data processing. Astronomers enjoy coding in Python due to its simplicity and great support for numeric and graphing libraries. Pydron takes single-threaded code written by astronomers and semi-automatically parallelizes it, deploys it on EC2 or private clusters, and returns the output to the astronomer.

Astronomers can't just use MapReduce (or similar big-data "solutions"). Compared to the large-scale commercial systems for big-data processing with thousands of deployments and millions of users, astronomy projects typically have just a single deployment, and fewer than 10 people. Furthermore, that single deployment involves never-seen-before data processing and analysis on never-seen-before kinds and volumes of data. Code reusability across such projects is very limited. Astronomers are also developers, not users: They continuously iterate the logic in the analysis code and rerun experiments until they obtain publishable results and never run the code again. The goal of Pydron is not only to reduce total execution time, but also decrease the amount of time spent by astronomers writing code (i.e., writing sequential code involves much less effort than a highly parallel MPI program).

Most astronomy code is broken up into two types of functions: those that "orchestrate" the overall analysis and those called from the orchestrator function to compute on data, with no global side-effects. Pydron only requires "orchestrator" functions to be annotated with a "@schedule" decorator, and compute functions with a "@functional" decorator. At runtime, upon encountering a @schedule decorator, Pydron will start up EC2 instances and Python processes, transfer over all the code and libraries, schedule the execution, and return the results back to the user's workstation (as if the code had been executed locally).

A random forest machine learning training example shows almost-linear speedup in execution time with an increasing number of cores using Pydron. Pydron generates data-flow graphs for functions annotated with the "@schedule" decorator, and dynamically updates the data flow graph at runtime with information that is unknown statically in Python (data-dependent control flow, dynamic type information, invoked functions, etc.). The changing data flow graph then informs decisions about the degree of parallelization possible from the code. The runtime of exo-planet detection code parallelized with Pydron decreased from 8.5 hours on a single EC2 instance down to 20 minutes on 32 instances, significantly reducing the turnaround time for astronomer tasks. Future work includes better scheduling algorithms, data-specific optimizations, pre-fetching, and dynamic resource allocation.

Brad Morrey (HP Labs) complimented the effort of improving the workflow of non-computer scientists by doing good systems work, but wondered where Pydron's approach of graph-based decomposition fails (and where performance is poor). Stefan answered by admitting that some algorithms are inherently sequential (and parallelization is not possible), and the system is designed for coarse-grained parallelization (where compute-intensive tasks take seconds/minutes), so that's where it sees the most benefit. Another limitation is that the system is currently designed for functional algorithms (that don't require changes to global mutable state). Stefan (Google) wondered whether the images in the example (specified by paths) were stored as files somewhere, and whether large data sets would limit the use of Pydron. Stefan admitted that Pydron currently uses Python's native serialization library (pickle) and sends objects over TCP, and there is room for future work. Scott Moore (Harvard University) asked whether the authors had looked into validating functional annotations dynamically (using graph updates). Stefan said that at the moment, no checks take place, but that it would be useful for developers to find bugs as well. Someone from Stanford clarified Pydron's assumption that each function must be able to run independently (no side-effects) in order for it to be parallelized and wondered how Pydron would work with stencil functions. Stefan replied that the inputs to the function would need to be passed as arguments (respecting the "no side-effects" rule).

User-Guided Device Driver Synthesis

Leonid Ryzhyk, University of Toronto, NICTA, and University of New South Wales; Adam Walker, NICTA and University of New South Wales; John Keys, Intel Corporation; Alexander Legg, NICTA and University of New South Wales; Arun Raghunath, Intel Corporation; Michael Stumm, University of Toronto; Mona Vij, Intel Corporation

Leonid began by remarking that device drivers are hard to write correctly, hard to debug, and often delay product delivery (as well as being a common source of OS failures). A key observation motivating the work is that device driver development is a very mechanical task: It boils down to taking an OS interface spec and a device spec and proceeding to generate the driver without much freedom into how the driver can interact with the device. In principle, this task should be amenable to automation. Leonid approached the device driver synthesis problem as a two-player game: driver vs. (device + OS). The driver is the controller for the device, and the device is a finite state machine where some transitions are controllable (triggered by the driver), while others are not controllable (e.g., packet arriving on the network, error conditions). The problem can be seen as a game, where the driver plays by making controllable actions, and the device plays by making uncontrollable actions. A winning strategy for the driver guarantees that it will satisfy a given OS request regardless of the device's possible uncontrollable actions.

Leonid used a LED display clock as an example. A driver that wants to set a new time on the clock to an arbitrary time must first turn off the internal oscillator of the clock to guarantee an atomic change of all the hour, minute, and second values (without oscillator-induced clock ticks changing the time in between operations). All the possible strategies are considered, and by backtracking from the winning goal (in which the new time is set correctly on the clock) to the current, initial state, you can find all the winning states in the game. Leonid proceeded to demo his device driver synthesis tool (called Termite) and showed how the tool generated code to change the time on the clock according to the winning strategy.

Crucially, Termite is a "user-guided" synthesis tool (in contrast to "push-button" synthesis tools, that generate the entire implementation automatically). The rationale is to keep developers in charge of important implementation decisions (e.g., polling vs. interrupt). The driver synthesis tool then works as

a very smart auto-complete-like tool (smart enough to generate the entire driver for you!), but maintains correctness in the face of arbitrary user code changes (and will inform the user if their proposed code changes cannot lead to a winning strategy). Termite also serves as a driver verification tool for a manually written driver.

Leonid then addressed the problem of specification generation for OS and devices: If developing the specifications for synthesis takes longer than writing the driver by hand, is it worth it? Two important observations are that OS specs are generic (i.e., made for a class of devices) and can often be reused, and device specs from the hardware development cycle can be obtained from hardware developers and used for synthesis. Some drivers synthesized to evaluate Termite include drivers for: a real-time clock, a Webcam, and a hard disk. Leonid then addressed the limitations of the work: the focus is currently on synthesizing control code (none of the boilerplate resource allocation code), it is single-threaded (current work-in-progress focuses on multi-threading the generated code), and it does not currently provide DMA support (also work-in-progress).

Ardalan Amiri Sani (Rice University) wanted to know how difficult it would be to figure out what the code that Termite generated is doing. Leonid explained that the kind of expertise you need to use Termite is the same you need to write a driver. There are tools that can help you make sense of what instructions are doing to the device, but at the end of the day you need to understand what you're doing. Someone from INRIA wondered whether it would be interesting for the tool to generate a Devil specification that described what the device does instead of generated code. Leonid replied that a better strategy would be to change the Termite specification language to include Devil-style syntax. Joe Ducek (HP Labs) wondered about the difficulty in getting DMA support and whether it would be possible to have the developer handle the DMA bits and let Termite do the rest. Leonid replied that Termite currently supports user-written DMA code in addition to the Termite-generated code. Automatically generating driver code for DMA is difficult because it generates a state explosion in the game-based framework. Brad Morrey (HP Labs) asked how exploring the state space to solve the two-player game scales. Leonid replied that a big part of the whole project was implementing a scalable game solver, and the results are published in a separate publication.

2014 Conference on Timely Results in Operating Systems

October 5, 2014, Broomfield, CO

Summarized by Timo Hönig, Alexander Merritt, Andy Sayler, Ennan Zhai, and Xu Zhang

Memory Management

Working Set Model for Multithreaded Programs

Kishore Kumar Pusukuri, Oracle Inc.
Summarized by Xu Zhang (xzhang@cs.uic.edu)

Kishore opened his talk with the definition of working set size (WSS) of multithreaded programs, which is the measure of the number of pages referenced by the program during a certain period of time multiplied by the page size. Knowing the WSS helps provide insight into application memory demand and is useful for dynamic resource allocation: for example, the page replacement algorithm in operating systems. Various approaches for approximating WSS exist, including simulation-based and program traces-based techniques. However, they only work on single-threaded programs, and more importantly, such measurements are too expensive to be applicable for effective resource management.

Characterizing WSS is also non-trivial. Not only does WSS vary from application to application, it is also affected by several factors. The author collected data from running 20 CPU-bound multithreaded programs on large scale multicore machines and identified four factors—what he denotes as "predictors"—that correlate to WSS: resident set size (RSS), the number of threads, TLB miss rate, and last-level cache (LLC) miss rate. To increase prediction accuracy and reduce the cost of approximation, Kishore further refined the predictors by pruning the ones of less importance using Akaike information criterion (AIC), avoiding overfitting and multicollinearity at the same time.

The resulting major predictors are RSS and TLB miss per instruction. Based on them, three statistical models were developed using supervised learning: linear regression (LR), K nearest neighbor (KNN), and regression tree (RT). These models are further selected using cross-validation tests, with KNN being the most accurate, which enjoys an approximation accuracy of 93% and 88% by normalized root mean squared error (NRMSE) on memcached and SPECjbb05. Notably, the developed model has very little overhead, which is in granularity of microseconds compared to hours.

Kishore also briefly talked about their ongoing work for WSS-aware thread scheduling on large scale multicore systems. He explained two existing scheduling algorithms, grouping and spreading, both using the number of threads as a scheduling metric. Grouping gangs threads together on a few cores initially and spreads them out if the number of threads exceeds the limit. By contrast, spreading distributes all threads uniformly across all cores and sockets at the start of the day. The author argued that using the number of threads for thread scheduling is not sufficient on the target system and illustrated this with two examples.

Ken Birman (Cornell) noted that approximating WSS is less important as memory has become larger and asked whether there were other contexts where such a machine-learning approach might be usable. Kishore replied yes and pointed out an application in virtual machine allocation, scheduling, finding failures, and providing high availability in the cloud. Ken followed up asking whether the author had the same findings in those cases where a small subset of the available metrics become dominant and are adequate for training. Kishore replied yes. The second questioner asked whether the errors are under- or overestimated. The author said it doesn't matter since WSS varies greatly from app to app. Someone asked what accuracy is acceptable for an application using such models. Based on his understanding of the literature, Kishore noted that above 80% is considered good. The final question was architecture related: Why not use a translation storage buffer (TSB) in the SPARC architecture, which caches the most recently used memory mappings, for WSS or data migration decisions? Kishore said their work is based on the proc file system without any kernel modifications.

MLB: A Memory-Aware Load Balancing for Mitigating Memory Contention

Dongyou Seo, Hyeonsang Eom, and Heon Y. Yeom, Seoul National University
Summarized by Alexander Merritt (merritt.alex@gatech.edu)

Dongyou Seo began by illustrating how modern CPUs are manycore and that future chips are envisioned to continue this trend; he referred to a forward-looking statement made by Intel for 1,000-core chips. To maximize the use of all cores, many systems accept more work by becoming multi-tenant, as is the case in server systems. Such scenarios, however, expose applications to possible contention on resources shared by cores, such as the last-level cache, and the limited bus bandwidth to local DRAM. A challenge, then, is to efficiently manage these shared resources to limit the effects of contention on applications, e.g., prolonged execution times. Existing OS task schedulers, such as in Linux, manage multicore chips by migrating tasks between cores, using the measured CPU load that a task places on a core as an important metric for load-balancing, while not prioritizing, or fully ignoring, the impact of memory bandwidth contention on task performance. The authors argue that memory bandwidth is one of the most important shared resources to understand, since the ratio between the number of cores and available bandwidth per core is increasing across generations of processors.

To address this challenge, their work presents a memory contention-aware task load balancer, "MLB," which performs task migration based on an understanding of a task's memory-bandwidth pressure. Their work contributes a memory-bandwidth load model to characterize tasks, a task scheduling algorithm using this model to influence when/where tasks are migrated among all processors, and an implementation of the first two contributions in the Linux CFS scheduler. An analysis was extended to compare against Vector Balancing and Sorted Co-scheduling (VBSC) and systems hosting a mix of CPU- and GPU-based applications.

Their contention model is defined by the amount of retired memory traffic measured by memory request events from last-level cache misses and prefetcher requests. Because each application's performance may be differently affected by the same level of contention, a sensitivity metric is defined for each application. Together, both metrics are used by an implementation of MLB in the Linux CFS scheduler. Tasks are grouped based on those that are highly memory intensive and those that are not. Two task run-queue lists are used (one for each category of task), with a single run queue assigned to a given core. Tasks are migrated between cores via the run queues when indicated by the memory contention model. An evaluation of their methods included a comparison with a port of the VBSC model into a modern version of Linux for a variety of applications from the SPEC benchmark suite as well as TPC-B and TPC-C. An extension of their work supports multithreaded applications. NUMA platforms, however, were not examined in this work.

Someone asked when their task migration mechanisms are triggered. Dongyou replied that a burn-in time is required to collect sufficient information for the model to stabilize (to distinguish memory intensive vs. non-memory intensive tasks). A second questioner addressed the extension of MLB to multithreaded tasks, asking whether the techniques in the paper would still apply. Dongyou responded that their methods can increase the hit rates of the last-level cache and that he plans to optimize them further. Following up, the questioner suggested it would be interesting to compare MLB to a manually optimized task-pinning arrangement to illustrate the gains provided by models presented in the paper.

A final question similarly addressed the lack of analysis on NUMA platforms, asking what modifications would be necessary. The response suggested that they would need to observe accesses to pages to understand page migration strategies, as just migrating the task would not move the memory bandwidth contention away from that socket.

Cosh: Clear OS Data Sharing in an Incoherent World

Andrew Baumann and Chris Hawblitzel, Microsoft Research; Kornilios Kourtis, ETH Zürich; Tim Harris, Oracle Labs; Timothy Roscoe, ETH Zürich
Summarized by Xu Zhang (xzhang@cs.uic.edu)

Kornilios started the talk by justifying the multikernel model—treating the multicore system as a distributed system of independent cores and using message passing for inter-process communication—a useful abstraction to support machines that have multiple heterogeneous processors. He illustrated with an example of their target platform—the Intel MIC prototype on which the Intel Xeon Phi processor is based. It has four NUMA domains or three cache-coherent islands, and transfers data with DMA between islands. Such heterogeneity breaks existing OS assumptions of core uniformity and global cache-coherent shared memory in hardware architecture. Multikernel models fit nicely since they treat the operating system as a distributed system.

One problem with the multikernel model, however, as Kornilios pointed out, is that there is no easy way to share bulk data, either for I/O or for large computation. Lacking support for shared memory, the multikernel model forces data copying to achieve message passing. This is the gap that coherence-oblivious sharing (Cosh) tries to close—to share large data with the aid of specific knowledge of underlying hardware. Cosh is based on three primitive transfers: move—exclusive read and write transfer from sender to receiver; share—read sharing among sender and receiver; and copy—share plus exclusive receiver write. And no read-write sharing is allowed by design. To make bulk sharing practical, two additional features—weak transfer and aggregate—are built on top of the primitives. Weak transfer allows the sender to retain write permission and to defer or even neglect permission changes. It is based on the observation that changing memory permission is costly and is not always necessary—for example, if the transfer is initiated from a trusted service. Aggregate provides byte granularity buffer access since page granularity doesn't work for everything, particularly handling byte data. Aggregate exports a byte API by maintaining an aggregate structure on top of page buffers. Kornilios illustrated the API with examples resembling UNIX pipes and file systems.

A prototype of Cosh was implemented on top of the Barrelfish operating system, which is an instance of the multikernel model. The prototype supports MIC cores natively. Kornilios showed weak transfer being a useful optimization for host-core transfers. And although pipelining helps, host-to-MIC transfers are bogged down with high latency. Kornilios also demonstrated results of an "end-to-end" evaluation of replaying panorama stitching from traces captured on Linux. While Cosh enjoys the same latency of Barrelfish's native file system on host-core transfers, the Cosh prototype still suffers from the high-latency of DMA transfers between MIC and host cores. With the help of perfect cache, the latency is reduced by a factor of 20 but is still 17 times greater than host-to-host copying latency. The major bottleneck is due to slow DMA transfers and slow co-processors.

Someone asked how the performance improved by using cache in host-to-MIC core transfers. Kornilios explained that the cache they implemented was a perfect cache and could be as large as possible, which reduced the number of DMA operations. The second questioner asked whether they are planning to support GPU. Kornilios said they haven't looked at GPU particularly because of the lack of support for access permissions. But GPUs are becoming more general purpose, so he is personally optimistic. Another question was whether they had explored the multiple writer concurrency model. Kornilios replied no, because there is no read-write sharing in Cosh by design. He further commented that write sharing is difficult for programmers to reason about, and cache-coherent and shared memory is hard to think about. He brainstormed that it might be feasible if data partitioning was provided or if versioning was available. The last questioner asked whether the Cosh model is used for synchronization. Kornilios answered no, since the multikernel is based on message passing and there is no shared memory.

System Structuring

Summarized by Andy Sayler (andy.sayler@colorado.edu)

Fractured Processes: Adaptive, Fine-Grained Process Abstractions

Thanumalayan Sankaranarayana Pillai, Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau, University of Wisconsin—Madison

Pillai opened by discussing the traditional OS process abstraction and noting that most applications are monolithic (single process). This poses problems when managing or debugging applications: for example, we are unable to restart just part of a process if it crashes or is upgraded but must restart the whole thing (potentially affecting GUIs and other components). Likewise, many existing debugging tools like Valgrind produce too much overhead when run on a large process, making such tools unfeasible to use in production (e.g., to monitor sensitive parts of an application for memory leaks). To counter these issues, the authors propose Fracture, a mechanism for subdividing applications into an isolated set of processes communicating via RPC. Fracture is designed to work with C programs and allows the user to flexibly isolate only the parts of the program that require it, minimizing the overhead incurred by splitting a monolithic program into multiple processes.

Using Fracture, a developer divides their program into logically organized modules (collections of functions) using annotations in the code. The developer then defines a mapping of each module into an associated FMP (fractured mini-process) that will run as an isolated process. Each FMP can accommodate one or more modules. Developers can dynamically reconfigure their applications to run in a single FMP (i.e., as they would have run without Fracture), or they can split the application into multiple FMPs to troubleshoot specific bugs, isolate sensitive code, etc. Fracture is also capable of using a min-cut algorithm and a list of developer-defined rules regarding which module must be isolated from which other modules to automatically partition a program into an optimal set of FMPs. Each FMP acts as a microserver, responding to RPC requests from other FMPs. In order to facilitate this, the developer must adhere to certain rules: no sharing of global state between modules, annotation of pointers so the data they point to may be appropriately serialized, etc.

The authors tested Fracture against a handful of existing applications: Null-httpd, NTFS-3G, SSHFS, and Pidgin, the universal chat client. The overhead imposed by using Fracture depends on the specific mapping of modules to FMPs chosen by the developer. In the base case where all modules map to a single FMP, Fracture obtains effectively native performance. At the other extreme, where each module is mapped to its own FMP, performance degrades significantly, from 10% of native performance to 80% of native performance depending on the application. Using Fracture's intelligent partitioning to automatically produce an optimal map of modules to FMPs minimizes performance degradation while also supporting the isolation of application components into easy to monitor, test, and debug components.

Liuba Shrira (session chair, Brandeis University) asked whether developers leveraging Fracture must manually define inter-module interaction in order for Fracture to compute an optimal partitioning. Pillai answered that no, a developer must only specify which modules must be kept isolated and provide the necessary annotations. Tomas Hruby (Vrije Universiteit) asked whether Fracture must maintain significant state in order to facilitate restarting FMPs, etc. Pillai answered that Fracture doesn't maintain significant internal state, but only requests state between module RPCs. This has some memory cost but does not impose significant computational overhead. Another attendee asked how correctness was affected when a program is split into multiple FMPs, e.g., when a single FMP crashes and is restarted. Pillai answered that Fracture requires the developer to build/divide modules in a manner that supports being safely restarted. Ken Birman (program chair, Cornell University) asked whether Fracture's concepts apply to languages like Java that have some existing tools for purposes similar to Fracture's. Pillai answered that there might still be value in porting the Fracture ideas to higher level languages, but that it is primarily designed for C, where no such tools exist.

Leo: A Profile-Driven Dynamic Optimization Framework for GPU Applications

Naila Farooqui, Georgia Institute of Technology; Christopher Rossbach, Yuan Yu, Microsoft Research; Karsten Schwan, Georgia Institute of Technology

Farooqui opened by introducing the audience to the basics of general purpose GPU computing, including presenting the GPU as a SIMD machine with a tiered memory hierarchy. GPU performance is often limited by two main factors: irregular control flow and non-sequential memory access patterns. Traditionally, optimizing GPU code to overcome these factors is difficult, requiring many manually applied architecture-specific tweaks. High level frameworks (i.e., Dandelion, Delite, etc.) exist to reduce the manual optimizing burden, but these frameworks still fail to capture the performance quirks of specific architectures, especially in terms of dynamically manifested performance bottlenecks. Farooqui et al. created Leo as an attempt to automate this performance-tuning process.

Leo automates performance tuning for GPUs by using profiling to optimize code for dynamic runtime behaviors. Farooqui presented a standard manual GPU performance optimization: Data Layout Transformation (DLT)—e.g., switching from row major to column major ordering to promote sequential memory access. To demonstrate the value Leo provides, Farooqui explained the challenge of applying the DLT optimization to an example program: SkyServer. In some cases the DLT optimization improves SkyServer performance, but in other cases performance is reduced. SkyServer requires dynamic optimization via a system like Leo to properly apply DLT when it helps while also forgoing it when it does not.

Leo employs dynamic instrumentation to drive runtime optimization. First, Leo generates a GPU kernel, then it analyzes, instruments, and executes this kernel. Next, Leo extracts profiling results and identifies candidate data structures for optimization. Finally, Leo applies the identified optimizations. Leo iteratively repeats this process, regressing to the previous state if no benefits occur. Leo leverages GPU Lynx for instrumentation and Dandelion for GPU cross-compilation. Leo's performance approaches that of an "oracle"-derived solution (e.g., one that is hand-optimized for a known input) with gains from 9% to 53% over the unoptimized version. And since Leo's optimization is fully automated, these speedups are effectively "free."

Kishore Papineni (Google) asked whether optimization challenges in SkyServer were only related to cache misses. Farooqui answered that there were also likely other factors at play (e.g., control flow). Ken Birman asked whether compile-time analysis can provide better layout predictions than Leo's dynamic analysis. Farooqui answered that it may be possible, but there are many challenges since compile-time optimizations can't, for example, see cache impacts. Kornilios Kourtis (ETH Zürich) asked Farooqui to comment on data-flow models vs. other models (e.g., which are easier to optimize). Farooqui answered that you can profile and optimize using non-data-flow models, but that data-flow models make generating multiple code versions easier. John Reiser asked the bit width at which data coalescing happens and whether the number of workers affects performance. Farooqui tentatively answered that the width was 8 bytes, but needed to verify that. The optimal number of worker threads is hardware dependent. Liuba Shrira asked how Leo handles multiple independent optimizations while avoiding a combinatorial explosion of profiling results. Farooqui answered that applying multiple independent optimizations is complicated and that they are working on how best to handle such optimizations in Leo.

Proactive Energy-Aware Programming with PEEK

Timo Hönig, Heiko Janker, Christopher Eibel, Wolfgang Schröder-Preikschat, Friedrich-Alexander-Universität Erlangen-Nürnberg; Oliver Mihelic, Rüdiger Kapitza, Technische Universität Braunschweig

Hönig opened by highlighting the challenges developers face today trying to optimize their code for minimal power consumption. Modern hardware has many power-related levers (e.g., sleep states, toggling peripherals, processor throttling, etc.), but developers don't have good tools for creating code that optimally utilizes these levers. Instead, developers are limited to infinite iteration loops, requiring them to hand-tweak code for power consumption and then manually measure power consumption using either the limited built-in measurement systems or traditional tools like oscilloscopes and multimeters. The process of optimizing code for power consumption today involves (1) writing and compiling code, (2) running code with a well-defined set of inputs, (3) performing a manual energy usage analysis, and (4) optimizing code and repeating the cycle. This is a very time-consuming process for the developer.

Hönig et al. set out to improve this situation by integrating energy measurement and optimization suggestion support into

modern IDEs. They proposed a system for doing this called PEEK, a proactive energy-aware development kit. The PEEK framework is split into three parts: the front-end UI (e.g., Eclipse plugin, or CLI), the middleware that handles data storage and tracking, and the back-end energy analysis system and optimization hint engine. PEEK leverages the Git version control system to snapshot copies of code to be analyzed as well as various build-parameters. This allows developers to separate potential enhancements into separate Git branches and then use PEEK to analyze and compare the respective energy performance of each branch. Completed PEEK analysis results and potential energy optimization tips are also saved via Git. The front-end UI extracts these data and can automatically generate a source code patch that the developer may then choose to apply.

In addition to building the PEEK framework, the authors created a novel dedicated energy measurement device. Existing energy measurement devices tend to lack programmable control interfaces and the necessary measurement capabilities to produce accurate results. The authors' solution leverages a current-mirror to produce accurate energy measurements even at limited sampling rates. Their system uses an ARM Cortex-M4 MCU and contains a USB-based programmable control interface for fully automated energy measurement. Using the PEEK framework and their custom hardware, the authors were able to reduce developer energy optimization time by a factor of 8 while also reducing code energy consumption by about 25%.

Naila Farooqui (Georgia Institute of Technology) referenced some of Hönig's result graphs and asked why performance did not always map to power. Why do slower programs not always use less power? Hönig answered that they would have to look more closely at specific situations to know for sure. Kornilios Kourtis (ETH Zürich) commented that the optimality search base for power-use tweaks must be huge, including compiler flags, scheduling options, etc. Kourtis then asked whether there is additional support PEEK can provide to make it simpler for the developer to select the ideal set of optimizations. Hönig answered that future work will aim to tackle this problem and provide better developer hints. Jay York asked how PEEK synchronizes hardware measurements with code execution. Hönig answered that the system uses GPIO signals and relies on minimal overhead/cycles in the GPIO loop to keep measurements synchronized with code. Liuba Shrira asked whether PEEK can take advantage of common sections of the code across multiple snapshots to avoid extra analysis. Hönig answered that although their previous work explores that, PEEK's snapshot batching capabilities are primarily aimed at allowing developers to logically group potential changes, not at minimizing back-end analysis effort.

System Structuring

Summarized by Timo Hönig (thoenig@cs.fau.de)

From Feast to Famine: Managing Mobile Network Resources Across Environments and Preferences

Robert Kiefer, Erik Nordstrom, and Michael J. Freedman, Princeton University

Robert Kiefer presented Tango, a platform that manages network resource usage through a programmatic model.

Kiefer motivated his talk by demonstrating that network usage of today's mobile devices impact other system resources (i.e., data cap, battery, performance). Network resources should be allocated in different ways depending on dynamic conditions (e.g., if a user changes location) that cause different network technology (i.e., WiFi, 3G/LTE) to become available. Users' interests may also change over time (foreground vs. background applications), and network usage may depend on usage characteristics (e.g., interactive vs. streaming). Divergent goals between user and application trigger resource conflicts that have to be moderated by the user.

Using the example of a streaming music app, Kiefer further illustrated various conflicts between applications and between applications and users. He showed that one of today's mobile systems (Android) only exposes "all or nothing" controls that are unsuitable for efficiently managing resources. The available configuration options are application-specific, and the naming of the configuration options differs between applications. Users usually cannot control resource usage on their mobile phones as they want to.

In Tango, user and application configurations are encoded as policies. With these policies, user configurations have top priority, while application configurations have some flexibility. Kiefer further introduced the architecture of their framework (measure and control primitives, controller, user and application policies). Policies are actually programs that turn states into actions. A state of the system is used as input for a policy program to transform the system into a new state. Actions may impact network interfaces or network flows, where user policies may affect actions at interface level and flow level, but application policies may only affect actions at (their own) flow level.

With constraints and hints, Tango detects conflicts regarding resource usage at the user and application level. Hints for "future rounds" (e.g., higher network bandwidth) are matched with existing constraints.

The evaluation of Tango was demonstrated by a streaming music application used across a campus WiFi with unreliable data connection. They used this scenario to analyze when to switch between WiFi and the mobile network (3G/LTE). The evaluation scenario revealed certain switching problems and demonstrated which user policies to apply to optimize the 3G usage.

Gilles Muller (INRIA) asked how Kiefer would optimize applications with two competing policies. Kiefer replied that because of hierarchies (different classes of applications), his framework

does not have to know each single application. If an application's policy does not fit into a user policy, the user may either change the constraints or find a different application that fits the user's policy. Someone commented on the campus use case that showed areas of "good" and "bad" WiFi, and asked why a "good" WiFi actually is good and a "bad" WiFi actually is bad. Kiefer replied that "good" WiFi meant the user was close to the WiFi access point. Bad WiFi was usually when noise and transmission errors increased as the user moved out of reach. Ken Birman asked whether there was a policy that should say, "Don't use my cell connection if my buffer is under x percent." Kiefer explained that they have implemented this with a policy that restricts the amount of data that may be consumed during a specific amount of time. With this, Kiefer et al. were able to reduce cell usage by about 30%.

On Sockets and System Calls: Minimizing Context Switches for the Socket API

Tomas Hruby, Teodor Crivat, Herbert Bos, and Andrew S. Tanenbaum, Vrije Universität Amsterdam

Tomas Hruby motivated his talk by giving a quick overview on the problems of the POSIX Sockets API. While the POSIX Sockets API is well understood and portable across different operating systems, Hruby raised the concern that (1) each API call requires its own system call, (2) system calls are disruptive, (3) OS execution interleaves with applications, and (4) non-blocking system calls are expensive. Hruby further outlined how current solutions address issues of POSIX sockets. In contrast, Hruby et al. believe that the socket API itself is not broken. Instead, it is the underlying implementations that are broken. The system call itself is the key problem.

The approach presented by Hruby tackles the challenge of eliminating system calls from sockets while keeping the API intact. Their proposed solution is so-called exposed socket buffers, which allow applications to inspect socket buffer data directly without going through system calls. This way, most system calls could be handled in user space. An exposed socket buffer consists of a piece of shared memory between the OS and the application, two queues, and two memory areas for allocating data. With this data structure, an application can test in user space whether a socket is empty or full. Further, Hruby gave details on how signaling is implemented in their system. As a result of the system design, the network stack cannot easily poll without disturbing the applications. This is why the authors decided to move the network stack to a dedicated core.

The implementation is based on NewtOS, a multiserver system based on MINIX 3. Hruby gave numbers on the amount of messages required to complete a recvfrom() call that returns from EAGAIN on the modified NewtOS (137 messages) and compared it to Linux (478 messages). Hruby emphasized the improvement over the original NewtOS implementation (>19,000 messages).

For the evaluation, the authors used lighttpd (single process) serving static files cached in memory on a 12-core AMD 1.9 GHz system with a 10 Gigabit/sec network interface. During the presentation, Hruby showed numbers on the instruction cache miss rate of lighttpd where NewtOS (1.5 %) was performing better than Linux (8.5 %) and the unmodified NewtOS (4.5 %). Before concluding his talk, Hruby discussed the limitations of the presented approach (e.g., fork() is not supported) and presented related work.

Jie Liao (Rice University) asked how much more effort it takes to adopt applications to the programming model of the presented approach. Hruby replied that the application remained unchanged since the programming model is not changed at all. Jon A. Solworth (University of Illinois at Chicago) asked what would happen when you have large reads (i.e., megabyte reads) since caching and shared memory would be affected. Hruby replied that it really depends on the application and how the application processes the data. Kishore Pusukuri (Oracle Inc.) wanted to know how the system handles multithreaded applications such as memcached. Hruby referred to previous work and said that their system can run multithreaded applications just fine. Kishore inquired about the underutilization of available resources. Hruby answered that this is not an issue. Xinyang Ge (Pennsylvania State University) asked whether the presented design impacts the performance of applications which do not use the network. Hruby answered that this is not a problem because they use hyper-threading for the network stack and so the core is not lost for other operations.

Lightning in the Cloud: A Study of Transient Bottlenecks on n-Tier Web Application Performance

Qingyang Wang, Georgia Institute of Technology; Yasuhiko Kanemasa, Fujitsu Laboratories Ltd.; Jack Li, Chien-An Lai, and Chien-An Cho, Georgia Institute of Technology; Yuji Nomura, Fujitsu Laboratories Ltd.; Calton Pu, Georgia Institute of Technology

Qingyang Wang presented their study analyzing very short bottlenecks that are also bottlenecks with a very short life span (~ tens of milliseconds) and their impact on the overall system. According to Wang, very short bottlenecks are causing large response-time fluctuations for Web applications (10 milliseconds to 10 seconds).

Their study analyzed the reasons for very short bottlenecks in different system layers (system software, application, architecture) and investigated how such short bottlenecks can lead to delayed processing, dropped requests, and TCP retransmissions. Common for the analysis of all three system layers is a four-tier Web server (Apache/Tomcat/OJDBC/MySQL) running the RUBBoS benchmark, which emulates the workload of 24 users. Wang presented results that the Java Garbage Collector of the evaluation system caused very short bottlenecks in Tomcat. These short bottlenecks eventually lead to a push-back wave of queued clients of the Apache Web server. (Wang acknowledged that the Java Garbage Collector was fixed in JDK 1.7 and no longer suffers from the very short bottlenecks of Tomcat running with JDK 1.6.) Wang further demonstrated results of very short bottlenecks caused by bursty workloads in virtual machines, which eventually led to dropped requests and TCP retransmissions.

Someone asked whether there are generic solutions to avoid push-back waves caused by very short bottlenecks. Wang answered that there are two different solutions to address very short bottlenecks for the presented evaluation. First, very short bottlenecks in the first evaluation scenario can be avoided by upgrading the Java version. Second, very short bottlenecks in the second evaluation scenario can be avoided by migrating the affected virtual machine to a different machine. However, the authors are still working on a generic solution that addresses the problem by breaking up the push-back wave. Landon Cox (Duke University) asked how to generally diagnose the cause of very short bottlenecks. Wang replied that it is not easy to diagnose the cause and that it helps to apply fine-grained monitoring tools that collect as much data as possible. However, Wang admitted that there is no generic way to diagnose very short bottlenecks.

Security

Summarized by Ennan Zhai (ennan.zhai@yale.edu)

Custos: Increasing Security with Secret Storage as a Service

Andy Sayler and Dirk Grunwald, University of Colorado, Boulder

Andy presented Custos, an SSaaS prototype that can preserve encryption keys if customers store any encrypted data on the remote cloud side. Andy first described the Dropbox and Google Drive storage background: For current cloud providers, customers either trust the providers or store encrypted data on the cloud side but keep the key themselves or on other cloud storage providers. Both cases introduce privacy risks. Custos can provide the ability to securely store keys, and its goals are: (1) centralized secret storage, (2) flexible access control, and (3) auditing and revocation.

For each of the three goals, Andy showed a corresponding architecture graphic that was very illustrative. For centralized storage, Custos manages the key between different devices and sends the key to a centralized server. In addition, the authors leveraged a scalable SSL processor and multiple providers to maintain key shares. About this part, Andy said they applied an *n*-threshold cryptographic primitive to combine key shares, finally generating the desired key. For the flexible access control property, Custos allows the customers to write the access control specifications and ensure the security of the stored keys. Using this mechanism, customers can control the access time of the keys and have time-limited access capability. For the final property of Custos, i.e., auditing and revocation, the system can audit logs and keep track of key access, thus offering auditing and revocation capability.

Ennan Zhai (Yale) asked where the maintainers hold key shares in practice, and who produces the access control specification. Andy said in practice there are many individual companies that offer services maintaining such shares, so Custos can distribute the shares to them. For the second question, Andy thought the customers can use existing tools or experts to achieve their goals; in practice it is not so hard to do. Someone noted that since Dropbox can share data with some mobile devices, it is harder for random third-party mobile applications to handle that. Andy said in principle it is not a concern for Custos, since the Custos system can flexibly handle such things.

Managing NymBoxes for Identity and Tracking Protection

David Wolinsky, Daniel Jackowitz, and Bryan Ford, Yale University

David Wolinsky began by noting that current anonymity tools (Tor) still cannot provide perfect privacy protection, and there have been many examples of an adversary focusing on breaking the user environment and not the tool. From this observation, the authors proposed Nymix, an operating system that can offer isolation for each browser task or session. Each browser or session is running in a virtual machine called Nym.

David used three interesting examples to describe the three target problems, including application-level attacks, correlation attacks, and confiscation attacks, which can expose users' privacy even if they use Tor. David then presented Nymix architecture design: In general, each Nym has two components: AnonVM and CommVM. AnonVM is mainly responsible for running the actual applications the user wants to run, while CommVM communicates with the outside environment. Since Nymix offers virtual machine isolations, the user environment cannot be compromised by the three types of attacks described above.

In the evaluation, David showed how the prototype can be set up as well as covering CPU overhead, memory usage, and networking overhead. Finally, David talked a lot about future improvements on Nymix, including fingerprintable CPU, VMM timing channels, accessing local hardware, and storing data retrieved from the Internet.

Someone asked about a fingerprintable CPUm and whether the authors had tried any experiments on this level. David thought in principle the results were not hard to anticipate, but it was still an interesting direction. The session chair asked about malicious virtual machines that can communicate with each other, compromising privacy defenses. David said that basically the paper does not need to consider such a case, the assumption being that the virtual machine manager is trusted.

10th Workshop on Hot Topics in System Dependability

October 5, 2014, Broomfield, CO

Summarized by Martin Küttler, Florian Pester, and Tobias Stumpf

Paper Session 1

Summarized by Florian Pester (florian.pester@tu-dresden.de) and Tobias Stumpf (tobias.stumpf@tu-dresden.de)

Compute Globally, Act Locally: Protecting Federated Systems from Systemic Threats

Arjun Narayan, Antonis Papadimitriou, and Andreas Haeberlen, University of Pennsylvania

Cascading failures can bring down whole systems, and in a world of critical systems, this should be avoided. However, privacy concerns often prevent the global view necessary to detect impeding catastrophic cascades. The authors studied this problem using the example of the financial crisis of 2008.

Banks usually have financial risks that are greater than their capital, therefore the surplus risk—the difference between their risk and their capital—is offloaded to other banks. This creates a network of dependencies between banks that results in a very large dependency graph. However, each bank only has its local view of the situation.

System-wide stress tests could be a solution to the problem; unfortunately, a lot of the input needed for such a test produces privacy issues for the banks. Therefore, economists do not know how to compute a system-wide stress test—although they do know what to compute. A regulator is not an option, because a regulating entity would require too much access. Secure multi-party computation, on the other hand, does not scale well enough.

A workable solution must deal with two main challenges: privacy and scalability. In order to provide scalability, a graph-based computation is used instead of matrix multiplication. Each bank is assigned a node and performs what the authors call limited multi-party computation.

In order to solve the privacy problem, the authors use differential privacy, which provides strong, provable privacy guarantees. This works by adding a small amount of imprecision to the output of the computations, which can be tolerated because the aim of the computation is detection of early warnings of large problems. Limited multi-party computation is essentially multiple multi-party computations with k parties. Each party gets another party's output as input for their own computation; in this way, no party has access to the other party's secrets.

Implementation was left as future work.

Someone commented that this work could also be applied for the power grid and other dependable systems. Additionally, the question was raised whether it is a realistic assumption that the banks will follow the protocol. The answer was that the regulators get localized data and can enforce the protocol.

Running ZooKeeper Coordination Services in Untrusted Clouds

Stefan Brenner, Colin Wulf, and Rüdiger Kapitza, Technische Universität Braunschweig

Privacy issues and insufficient trust in cloud providers slow down the adoption of cloud technologies. However, computation of data is tricky if that data is encrypted. Trusted Execution Environments (TEE) can help to mitigate the problem.

Apache ZooKeeper provides coordination for cloud applications. The authors added an encryption layer to ZooKeeper in order to gain trusted coordination.

This encryption layer is provided in the form of the ZooKeeper privacy proxy, which runs inside a TEE. Communication from the client to the ZooKeeper privacy proxy is encrypted by SSL, while communication between ZooKeeper and the ZooKeeper privacy proxy is encrypted using a shared key between these two. Name-clashing problems are solved by a dictionary node.

Someone asked the presenter to clarify whose signing key needs to be trusted ultimately. The answer was the hardware signing key. A second questioner wanted to know more about the applications that can be used with the system. The presenter answered that the system needs specific applications. The final questioner wondered why the system was implemented as a proxy. The reason is so that this solution is completely transparent to the server and the client.

Who Writes What Checkers?—Learning from Bug Repositories

Takeshi Yoshimura and Kenji Kono, Keio University

Takeshi and Kenji presented a tool that learns from bug repositories to help developers eliminate bugs. The tool uses static code analysis to find typical bugs—e.g., not freed memory or infinite polling—and is based on the Clang analyzer. During analysis Takeshi figured out that bugs coming from humans are mainly domain specific. Their tool uses machine-learning techniques to extract bug patterns from documentations written in English and the corresponding patches. The tool works in two steps: First, it uses natural-language processing to extract keywords from the documentation. Second, their tool extracts bug patterns based on topic. For the paper, Takeshi analyzed 370,000 patch documentations and sorted them into 66 groups called "clusters." They also found subclasses by using keywords; for instance, around 300 patches contain the keyword "free." They use the known free-semantic to check patches and see whether a free statement is missing. The presented tool is useful for detecting typical bugs. In his conclusion, Takeshi mentioned that finding unknown bugs is more complicated but possible.

Someone asked about the documentation used. Takeshi clarified that they used only the commit messages and not information from the bug repository. Further questions were related to language processing. Takeshi explained that they used techniques to reduce language noise from developers' commit messages.

The last questioner wanted to know which kind of bug patterns they could find and which not. The presenters clarified that their tool is limited to known bugs which are already documented.

Leveraging Trusted Computing and Model Checking to Build Dependable Virtual Machines

Nuno Santos, INESC-ID and Instituto Superior Técnico, Universidade de Lisboa; Nuno P. Lopes, INESC-ID and Instituto Superior Técnico, Universidade de Lisboa and Microsoft Research

Nuno Santos and Nuno Lopes developed an approach based on trusted booting and model checking to ensure that a virtual machine (VM) in the cloud runs the software that a customer expects. The basic approach to launch a VM is quite easy. Someone creates a VM image and sends it into the cloud, and someone (maybe the image creator) launches the VM. Previous studies showed that this approach includes risks for the creator as well as for the user. For instance, a misconfigured VM image can contain creator-specific data (e.g., passwords, IDs) or include obsolete or unlicensed software. To overcome these problems, Nuno proposed model checking to ensure that a VM image matches the specifications before it is launched. The specification includes all necessary information (e.g., configurations, applications) to ensure a specification behavior of the VM. After a VM is checked, a trusted computing environment is used to ensure that the checked image is launched.

Someone wanted to know how to identify passwords. Nuno answered that it is done by annotations. A second questioner asked who he has to trust (machine, cloud provider) to launch a VM. The software integration can be done outside the cloud, and therefore it is not necessary to trust any cloud provider. Another question related to software specification in general. Nuno answered that their work does not focus on software verification but on checking the properties of a given instance. Finally, to the question of how much work it takes to write a specification, Nuno said the workload depends on the level of fine tuning. It is simple to write a running specification, but the tuning part can be quite expensive.

Paper Session 2

Summarized by Martin Küttler (martin.kuettler@os.inf.tu-dresden.de)

Erasure Code with Shingled Local Parity Groups for Efficient Recovery from Multiple Disk Failures

Takeshi Miyamae, Takanori Nakao, and Kensuke Shiozawa, Fujitsu Laboratories Ltd.

Takeshi Miyamae began by noting the need for erasure codes with high durability and an efficient recovery. He presented a Shingled Erasure Code (SHEC) that targets fast recovery and correctness in the presence of multiple disk failures.

He discussed the three-way tradeoff between space efficiency, durability, and recovery efficiency. SHEC targets recovery efficiency foremost by minimizing the read sizes necessary for recovery. Compared to Reed Solomon, MS-LRC, and Xorbas, SHEC has substantially better theoretical recovery speed for multiple disk failures.

Next, Takeshi showed that SHEC is comparable to MS-LRC in terms of durability for double disk failures, but is more customizable and offers more fine-grained tradeoffs in efficiency. He then briefly explained the implementation of SHEC, which is a plugin for the free storage platform Ceph.

Takeshi presented an experiment comparing SHEC to Reed Solomon for double disk failures. He found that SHEC's recovery was 18.6% faster and read 26% less data from disk. Then he showed that there was 35% more room for recovery time improvement because the disks were the bottleneck only 65% of the time.

Someone asked why the disks were not always the bottleneck during the experiment. Takeshi explained that other resources can be the bottleneck too, but in their experiment only the disk was a bottleneck. He was not sure what acted as bottleneck during the remaining 35% of the experiment, but it was not the CPU or the network.

The second questioner wondered how common it is to have multiple disk failures. Takeshi answered that the probability for that is higher in practice than in theory.

Providing High Availability in Cloud Storage by Decreasing Virtual Machine Reboot Time

Shehbaz Jaffer, Mangesh Chitnis, and Ameya Usgaonkar, NetApp Inc.

Shehbaz Jaffer presented work on providing high availability in cloud storage. Virtual storage architectures (VSA)—storage servers that run as virtual machines on actual hardware servers—are considerably cheaper and more flexible than hardware servers. But they suffer from lower availability, because in case of failures they need to be rebooted, which introduces long wait times. The typical solution for hardware servers is to deploy high availability (HA) pairs, i.e., two servers that both serve requests, so that one can keep working when the other reboots.

To achieve the goal of this work, reducing the VSA reboot time in order to increase availability, Shehbaz presented multiple steps. The first was to cache VM metadata in the host RAM and provide access to that data on reboot. This improved boot time by 5%, which was less than was expected.

Next the authors tried to improve the SEDA architecture to improve performance. In particular, synchronously returning cached file-system information, instead of performing an asynchronous call, reduced boot time by 15%.

Finally, they improved block-fetching time during a reboot. Turning read-ahead during reboot decreased the boot time by another 18%.

The time breakdown showed that consistent checkpointing takes a lot of time. The authors left replacing consistent checkpointing with a faster technique as an open problem, which they want to look at in follow-up work.

Somebody asked how often VSA failures, and subsequent reboots, happen in practice. Shehbaz answered that about 85–90% of the failures are software failures, such as OS bugs or mishandled client requests. But he had no absolute numbers. Another questioner asked why not use multiple replicated VMs, which can take over when one fails. Shehbaz answered that in deploying a VSA at some datacenter, there is no way to ensure that the VMs are going to be near each other, and maintaining consistency is much harder when they aren't.

Understanding Reliability Implication of Hardware Error in Virtualization Infrastructure

Xin Xu and H. Howie Huang, George Washington University

As motivation for their work, Xin Xu pointed out that hardware failures are a common problem in the cloud: If a server has one failure in three years, a datacenter with 10,000 servers has an average of nine failures per day. Still, the datacenter provider needs to handle these incidents to provide a reliable service.

In his work, Xin focused on soft errors such as transient bit flips, which are not reproducible and are hard to detect. With smaller process technologies, the probability for transient failures is expected to increase greatly, which is a major reliability concern in datacenters.

He also pointed out that hypervisors are a single point of failure and do not isolate hardware errors. They studied this problem by doing fault injections into virtualized systems. For these experiments, the authors analyzed the hypervisor to find the most used functions and found that by injecting errors in them, they could find more crashes than by doing random injections. Using this analysis, they also found more errors that propagated into the VMs.

In addition, they categorized the injected faults based on the crash latency, which is the number of instructions between fault injection and detection of the crash. Most crashes have a short latency of fewer than 100 instructions, but some are significantly longer and are thus likely to propagate into a VM. To study this they also analyzed failure location. In many cases, crashes happen in the C-function where a fault was injected. A small percentage (up to 5%) leave the hypervisor, meaning there is no fault-isolation.

Comparing his work to previous approaches, Xin highlighted two new contributions: a simulation-based fault injection framework, and analysis of error propagation.

Somebody asked whether they considered bit flips in opcodes as well as in program data. Xin answered that they only injected bit flips into registers, because injecting errors into memory is more difficult to track. He was then asked whether he considered storage more reliable. He responded that he did not, but that memory is protected by ECC, and that errors in the CPU are more difficult to detect. Somebody asked how Xin would go about making the hypervisor more robust, and what implications their framework had on performance. For the first question, Xin referred the questioner to another paper on ICPP about detecting failures early, before they propagate to dom0. Regarding the performance, he said that the overhead was generally less than 1% because they used a lot of hardware support such as perf counters.

Towards General-Purpose Resource Management in Shared Cloud Services

Jonathan Mace, Brown University; Peter Bodik, Microsoft Research; Rodrigo Fonseca, Brown University; Madanlal Musuvathi, Microsoft Research

Jonathan Mace presented the motivation of the work, describing how performance problems can arise in shared-tenant cloud services. Requests can drain resources needed by others. The system typically does not know who generates requests, nor which requests might interfere, and therefore it can give no performance guarantees to users. Ideally, one would like quality-of-service guarantees similar to those provided by hypervisors.

To address these issues, Jonathan proposed design principles for resource policies in shared-tenant systems. He then presented Retro, a prototype framework for resource instrumentation and tracking designed according to these principles.

He started by discussing interferences of various file-system operations on an HDFS installation. He presented measured throughput rates of random 8k writes with different background loads. Even tasks like listing or creating directories interfered significantly with the writer task. From that the authors inferred the first principle: Consider all request types and all resources in the system.

Jonathan motivated the second principle by discussing latency requirements of specific tasks. He provided an example where a high priority tenant required a low latency, and two tenants with lower priority were running. Throttling one of them could significantly decrease the latency of the high priority tenant, while throttling the other did not change the latency at all. Therefore only the correct tenant should be throttled, which led to the second principle: Distinguish between tenants.

Jonathan discussed long requests and the problems they pose for giving guarantees to other tenants, which generated the third principle: Schedule early and often. This avoids having long-running requests taking up resources that should be available for high priority tenants.

Finally, Jonathan presented Retro, a framework for shared-tenant systems that can monitor resource accesses by each request. The framework tracks which tenant issued each request and resource access, aggregates statistics (thus providing a global view of all tenants), and allows a controller to make smart scheduling decisions. He revisited the example of the high priority tenant interfering with one of two low priority tenants. Retro figures out which low priority tenant it has to throttle to improve the latency of the higher priority tenant. Jonathan also presented measurements showing that Retro introduced less than 2% overhead.

Somebody asked about the level of isolation that Retro provides and whether crashes were considered. Jonathan answered that Retro can provide performance isolation but no strong guarantees, as only averages can be guaranteed. He also said that they did not consider crashes.

Scalable BFT for Multi-Cores: Actor-Based Decomposition and Consensus-Oriented Parallelization

Johannes Behl, Technische Universität Braunschweig; Tobias Distler, Friedrich-Alexander-Universität Erlangen-Nürnberg; Rüdiger Kapitza, Technische Universität Braunschweig

Johannes Behl presented a scalable Byzantine fault tolerance implementation for multicore systems. He motivated this work by saying that many systems targeting dependability only consider crashes, which is not enough. To be able to handle more subtle faults, Byzantine fault tolerance (BFT) is needed. But it is not common, because current BFT systems do not scale well with multiple cores, which is required to run efficiently on modern hardware.

Johannes talked about state-of-the-art BFT systems and their parallelization. The BFT protocol consists of an agreement stage and an execution stage, which need to be parallelized. The parallelization of the second stage depends on the service characteristics. For the agreement stage, Johannes briefly presented the current method of parallelization. The agreement is split into multiple tasks, each running in a different thread. Thus, finding a consensus involves all threads, which has a number of disadvantages. The number of threads is determined by the BFT-implementation and is therefore not well aligned with the number of cores. In addition the slowest task inside the agreement phase dictates the performance. And the need for frequent synchronization between threads further increases the time to find consensus.

Johannes presented consensus-oriented parallelization, which involves vertical parallelization, i.e., having each thread serve a complete agreement request. That way throughput rates can scale with the number of CPU cores, and synchronization is much less frequent.

In the evaluation, Jonathan compared his implementation to a state-of-the-art implementation of BFT. His implementation scales much better, and it is already about three times faster for a single core, because it does not have any synchronization overhead there.

Someone asked how the logging inside the agreement stage was parallelized. Johannes answered that they did not log to disk but only to memory. For disk logging he proposed using multiple disks or SSDs. The next question was how dependencies across requests were handled. Johannes answered that they made no assumptions about dependencies. Somebody asked whether the evaluation was done in the presence of faults so that non-trivial agreements had to be found. Jonathan answered that there were no faults or disagreements in the evaluation. The last question was whether batching was used in the evaluation. Jonathan said no and noted that batching can improve performance.