4th Symposium on Operating Systems Design and Implementation (OSDI 2000)

OCTOBER 23–25, 2000

SAN DIEGO, CALIFORNIA, USA

KEYNOTE ADDRESS

SYSTEMS ISSUES IN GLOBAL INTERNET CONTENT DELIVERY

Daniel Lewin, Akamai Technologies, Inc.

Summarized by Darrell Anderson

Daniel Lewin started his talk with a brief introduction to the development of the Internet, emphasizing how it is really built, and how that influences what his company does. In the beginning, the Web was very simple: content providers on one side of the "Internet," users on the other. In between, network providers made sure there was a path from every user to every content provider, and vice versa. This scheme has four bottlenecks: the first mile, peering points, the network backbone, and the last mile.

The first mile includes the content provider's databases, application servers, Web servers, load balancers, switches, routers, and bandwidth. This centralized traffic creates an inherent bottleneck.

Around 7,000 networks account for up to 95% of the access traffic, and the number of those networks is growing. The edge of the Internet is becoming more diverse, with many small ISPs.

Peering is critical to how the Internet works. Peering is a bottleneck because there's no incentive to peer well — networks compete. Also, the technology for peering is inadequate: pipe size is limited, and it is difficult to peer two networks at more than five to 10 places simultaneously.

The third bottleneck is the backbone. Backbones are difficult to build and are very expensive. The business model of a backbone requires high utilization to drive down costs. Though capacity is increasing, it cannot keep up with demand. Peak utilization of the backbone is around 200Gbps (usage data from forward proxy logs), close to the "useful" capacity of the network. Demand can increase faster than capacity.

The last mile presents the final bottleneck. Once the last mile gets fast, one would expect that the whole Internet will get faster. However, if you gave everybody a cable modem, the current infrastructure of the Internet would collapse. The bottom line is, centralized delivery is slow and unreliable.

At a very high level, Akamai does "edge distribution," deploying servers inside many networks. The goal is to infuse all 7,000 networks that matter, distributing data from the content provider to these servers, serving content from as close to the end users as possible. Edge distribution bypasses most bottlenecks (first mile, peering, and backbone), improving performance, reliability, and capacity. Akamai provides network monitoring tools, and its servers are free.

Akamai estimates speeds on average are between two and 46 times faster, with an 86% reduction in download times. In addition, edge distribution improves consistency and availability.

Lewin then posed the following questions: How should a content distribution network organize servers? Storage? What data do we need to predict performance? How can we gather this data reliably and in real time?

This data enables resource management/server selection. Data gathering and server selection must be distributed and fault tolerant, and must work with imperfect information and in an unreliable setting. Also, system monitoring and management need to represent data visually to alert and allow humans to interact with the system. Object distribution and invalidation introduce problems in maintaining consistency in a massively distributed system. A content distribution system must provide access information the same way a centralized server would.

Later, Lewin described the "common point" metric, estimating the latency between end users and servers. Rather than measure actual latency between users and servers, measure latency of segments of routes, ending in a common point and enabling correlation. As a set coverage problem, common points reduce the set size from 200K to 6K, enabling feasible measurement. Pulling it together, the system computes the common point sets, gathers network data, distributes data, and performs server selection.

Akamai uses the DNS hierarchy to break up resource management and server selection into usable chunks.

Lewin concluded that the Internet is an evil place, subject to the Heisenberg uncertainty principle. It is difficult to predict popularity, distributions, and capacity. Server selection wants information quickly and accurately, and even in the face of inaccuracies, should not overload available resources. This is a hard problem.

Akamai's solution makes a few simple (and sometimes wrong) assumptions: that utilizations converge, popularities are poison, and the number of active users behind any particular DNS server is never too large. When these assumptions fail, servers or links may be overcommitted. The problem is easily parallelized because groups of users may be split.

For more information, see <*http://www.akamai.com*>.

SESSION: APPLYING LANGUAGE TECHNOLOGY TO SYSTEMS

Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions

Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem, Computer Systems Laboratory, Stanford University

Summarized by David Oppenheimer

Dawson Engler described a compiler extension system that allows compile-time checking of application-specific rules. The extensions are written in a state-machine language called *metal*, which somewhat resembles the yacc specification language, and are dynamically linked into a modified version of g++ called xg++. The metal specification describes patterns in the source language that, at compile time, cause state transitions in the state machine described by the metal specification. At the time xg++ translates a function into the compiler's internal representation, it applies the metal extensions down every code path in the function. States corresponding to rule violations signal a potential rule violation in the source program. This system is one instantiation of a general concept Engler calls *Meta-Level Compilation (MC)*, which raises compilation from the level of the programming language to the "meta level" of the code itself (e.g., interfaces, components, and system-level properties), allowing the checking, optimization, and transformation of systems at a high level much as is done for low-level code by compilers for code written in traditional programming languages.

Engler argued that MC offers benefits over traditional techniques for detecting violations of system rules. Specifically, he suggested that MC rules are significantly easier to write than formal specifications (verification), scale much better with code size (testing), and don't require difficult reasoning about the code (manual inspection). By informing the compiler of system-specific rules about the code, MC allows automated checking of many system-level properties at compile time.

Engler applied an MC system consisting of metal specifications that extend the xg++ compiler, to Linux, OpenBSD, the Stanford FLASH machine's protocol code, and the Xok exokernel. By checking rules, Engler's system found over 600 bugs in these systems. Most extensions were written in fewer than 100 lines of code and by individuals unfamiliar with the MC system itself.

During Q&A, several conference participants asked about opportunities to improve the system in areas such as reducing the number of false-positive errors reported, handling function pointers, and automatically deriving rules from code by looking at what the code "usually" does (from a static code path perspective). In response to a question about the feasibility of using the tool throughout the software development process, Engler indicated that this sounded like a potentially good idea, and he pointed out that the system's usefulness is improved when programmers structure their code to be as simple as possible, so that it is more amenable to compiler analysis.

For more information, see *<http://www.stanford.edu/~engler/>*.

Devil: An IDL for Hardware Programming

Fabrice Mérillon, Laurent Réveillère, Charles Consel, Renaud Marlet and Gilles Muller, Compose Group, IRISA/INRIA

Summarized by David Oppenheimer

Gilles Muller described Devil, an Interface Definition Language (IDL) for hardware programming. Devil attempts to ease the driver development process by defining an IDL in which a driver writer composes a strongly typed, high-level, easy-to-write description of a hardware device's software interface.

The Devil IDL is based on a few key abstractions: *ports*, which serve as a communication point and correspond to an address range; *registers*, which serve as a repository of data and are the grain of data exchange between a device and the CPU; and *variables*, which serve as the programmer interface and correspond to collections of register fragments that are given semantic values, such as bounded integers or enumerated types.

The IDL compiler checks the consistency of a Devil specification with respect to properties such as conformance to type rules, use of all declared entities, absence of multiple definition of entities, and absence of overlap of port and register descriptions. It then generates the necessary low-level driver code, which includes code to check for the proper use of the generated interface.

Muller described the Devil description of a device driver for the Logitech Busmouse and several other devices, including an IDE disk controller. Muller has found that device drivers generated using Devil are five times less prone to errors than is C code, and that Devil offers negligible performance overhead while improving programmer productivity. Muller's vision is that hardware vendors will supply device specifications as a Devil description, which can then be used to generate documentation and the device driver itself.

During Q&A, one conference attendee pointed out that embedding the hardware specification inside the compiler limits the possibility of simultaneously generating drivers for multiple platforms.

For more information, see *<http://www.irisa.fr/compose/devil/>*.

TAMING THE MEMORY HOGS: USING COMPILER-INSERTED RELEASES TO MANAGE PHYSICAL MEMORY INTELLIGENTLY

Angela Demke Brown and Todd C. Mowry, Carnegie Mellon University

Summarized by Tep Narula

Angela Demke Brown began her talk by pointing out the rapid memory consumption behavior of computational problems with large data sets, termed "out-of-core" applications. Such applications often suffer from high page-fault rates caused by the operating system's default virtual memory management policy. An earlier work by the same team had shown that, by using the compiler to analyze and insert page prefetches into the code, out-of-core applications could achieve good performance improvement on a dedicated machine. However, out-of-core tasks with aggressive prefetching tend to have a severe negative impact on other applications in a multiprogrammed environment. Brown then described the extensions they made in order to turn an out-of-core task into a "good neighbor" without placing any additional burden on the programmer.

The system consists of three parts: OS support, compiler analysis, and runtime layer. The OS support includes primitives for page prefetch and releases, as well as information on page location, usage, and availability. This was implemented as a memory management policy module and a kernel daemon called *releaser* on SGI IRIX 6.5. The compiler algorithm performs reuse analysis, locality analysis, loop splitting, and software pipelining in order to decide where to insert prefetch and release hints. The implementation is a pass in the Stanford University Intermediate Format (SUIF) compiler. The runtime layer dynamically analyzes both the static, compiler-supplied hints and the dynamic, OS-supplied system status and decides when to issue a request for either prefetch or release of a page to the OS. The runtime layer is implemented as a library that spawns a pool of pthreads to handle the prefetch and release requests within the application space. There are two release policies implemented in the library: aggressive release and priority-based buffered release. Experiments were performed using the out-of-core version of five applications taken from the NAS Parallel benchmark suite plus the MATVEC kernel. An SGI Origin 200 with four processors was used for the experiments. Overall, the results showed significant performance improvements both in dedicated and multiprogrammed scenarios.

SESSION: SCHEDULING

SURPLUS FAIR SCHEDULING: A PROPORTIONAL-SHARE CPU SCHEDULING ALGORITHM FOR SYMMETRIC MULTIPROCESSORS

Abhishek Chandra, Prashant Shenoy, and Micah Adler, University of Massachusetts, Amherst; Pawan Goyal, Ensim Corporation

Summarized by Darrell Anderson

Scheduling is important for diverse Web and multimedia applications, such as HTTP, streaming, e-commerce, and games. Applications are often hosted on large, multiprocessor servers. A key challenge is to design OS mechanisms to provide fair and proportional resource allocation. Other requirements include isolating misbehaving or overloaded applications and achieving low overheads for efficient implementation in real systems.

Proportional-share scheduling is one class of scheduling algorithms that addresses these requirements. It associates a weight with each application and allocates CPU bandwidth proportional to weight. There are a number of algorithms in use, but do they work well on multiprocessor systems?

Abhishek Chandra illustrated how one such algorithm, *Start-Time Fair Queuing* (*SFQ*), can lead to starvation when scheduling three threads on two CPUs. This starvation is a result of "infeasible weight assignment," where the accounting is different from actual allocation. A simple correction, or "weight readjustment," can be made limiting any one thread's weight to a single CPU's bandwidth, preserving overall weight ratios. Weight readjustment is efficient, running in time proportional to the number of CPUs in the system, and can be incorporated easily into existing scheduling algorithms.

A second problem arises under frequent arrivals and departures of short jobs, which Chandra calls the "short jobs problem." Again, SFQ performs correctly on uniprocessor systems but breaks down when scheduling across multiple processors.

Surplus Fair Scheduling (*SFS*) addresses this problem. With this algorithm, the scheduler measures observed processor bandwidth share, which it compares with the ideal share, computing the scheduling surplus for each thread. By scheduling threads in order of increasing surplus, the short jobs problem sees fair scheduling on single and multiprocessor systems.

Chandra presented proportional allocation tests where SFS yields near optimal allocation. Second, SFS was compared with time sharing for isolation and scheduling overhead. SFS provides superior isolation at modest additional scheduling overhead.

Q: Don't many scheduling algorithms, such as lottery scheduling, have nearly trivial extensions for multiprocessor systems?

A: Lottery scheduling has problems with proportional share when applied directly on a multiprocessor. On a multiprocessor, tickets do not translate to scheduling weight.

Q: You claim a proportional-share scheduling algorithm will not work well on multiprocessors. Does your algorithm

scale? What happens to the overhead with a very large number of processors?

A: We have developed some algorithms independent of the number of processors.

Q: How important is the virtual time in your algorithm?

A: The idea is basically heuristic.

Q: It doesn't seem that the problem is inherent in SFQ. It seems you need to have a notion of a global virtual time, rather than weight readjustment.

A: We looked at a few algorithms we could apply, but did not come up with a simple answer.

For more information, see <*http://lass.cs.umass.edu/software/gms*>.

Performance-Driven Processor Allocation

Julita Corbalán, Xavier Martorell, and Jesús Labarta, Universitat Politècnica de Catalunya

Summarized by Darrell Anderson

Performance-driven processor allocation uses runtime information to make sure that processors are always in use. The scheduling problem is how to allocate processors to applications, both for space sharing and time sharing. Things work best when the number of processes is equal to the number of processors.

Processor allocation should be proportional to application performance. A drawback of this metric is that application performance is not known before execution. One solution involves a priori calculation by measuring multiple executions, using previous results to predict later performance. Julita Corbalán proposed an alternative approach where processors are allocated to those applications that can take advantage of them. This runtime dynamic performance analysis approach, called *Performance-Driven Processor Allocation (PDPA)* requires coordination between medium- and long-term schedulers.

Corbalán used the NANOS execution environment on a shared memory multiprocessor. NANOS uses a queuing system and CPU Manager to schedule OpenMP parallel applications. The dynamic performance analysis is done by the SelfAnalyzer, a tool that estimates execution time for processes.

The SelfAnalyzer remembers baseline performance results for two and four processors, which are then used to predict performance. Performance-driven processor allocation is space sharing, allocating for acceptable efficiency. Processes run to completion with minimum allocation of one processor. Dynamic partitioning and reallocation is driven by the runtime system.

Corbalán compared PDPA against three other multiprocessor schedulers, using the OpenMP application suite on an SGI Origin 2000 with 64 processors running IRIX 6.5.8 and multiprogramming level set to 4, PDPA performs as well as, and frequently better than, the competing schedulers. Corbalán showed different applications that perform well for one scheduler, with matching PDPA performance. In some cases, PDPA performs significantly better than all three alternatives. Corbalán observed that it is very important to provide accurate performance information to the scheduler.

For more information, see <*http://www.ac.upc.es/NANOS*>.

Policies for Dynamic Clock Scheduling

Dirk Grunwald and Philip Levis, University of Colorado; Keith Farkas, Compaq Western Research Laboratory; Charles Morrey III and Michael Neufeld, University of Colorado

Summarized by Darrell Anderson

"Saving energy or power is important, both for battery life and at the microarchitecture level for clock speeds," said Michael Neufeld, as he indicated that this work is a study of proposed algorithms and is a negative result.

Instantaneous power consumption of CMOS components is proportional to the square of voltage, times frequency. Batteries will perform better if drained at a lower rate. Secondly, frequency depends on voltage. Greater voltage permits higher frequency, to a point. Lower frequencies provide more than linear reduction in power needs. It is better to run a system slowly and steadily than to run it as fast as possible and then shut the processor off. Clock scaling algorithms require load prediction and speed adjustment.

Adding to prior work, Neufeld's contribution includes a real implementation instead of simulation, focusing on practical aspects. The authors use an exponential weighting algorithm proposed in earlier work, predicting utilization from earlier intervals. Their implementation uses a 10 millisecond interval time, looking back three intervals with 50% and 70% busy as thresholds.

They modified the Compaq Itsy to measure current draw and power consumption with 5,000 samples per second. They ran the Compaq-modified Linux kernel v2.0.30, adjusted to perform clock/voltage scaling. They also used the Kaffe Java VM, instrumenting it to record and replay applications. They ran four applications, measuring energy and smoothness. Smoothness implies infrequent changes in clock speed and voltage. They ran an MPEG player, hoping that its characteristics would translate well to clock scaling. In practice, the algorithm performed only marginally better.

The negative result: weighted averaging methods do not appear to work well. What went wrong? Averaging attenuates oscillations, but does not remove them. Larger interval history might help, but would reduce responsiveness. Even if tuning were possible, it would be fragile. Also, a linear change in frequency doesn't

always mean a linear change in idle time. The authors don't have a conclusive explanation.

Neufeld points out that existing hardware has only limited voltage-scaling capabilities. A wider range of hardware would be very useful for experimentation.

SESSION: STORAGE MANAGEMENT

Towards Higher Disk-Head Utilization: Extracting "Free" Bandwidth from Busy Disk Drives

Christopher R. Lumb, Jiri Schindler, Gregory R. Ganger, David F. Nagle, Carnegie Mellon University; Erik Riedel, Hewlett-Packard Labs

Summarized by Mac Newbold

"Disk drives increasingly limit performance in many computer systems," Greg Ganger pointed out as he explained the need for better utilization of disk bandwidth. His presentation outlined a method of scheduling disk accesses that can extract an additional 20–50% of a disk's potential bandwidth without affecting the service times of the original requests. Their method describes two pools of requests: foreground requests, which include the normal and high-priority workload of the disk, and background requests, low-priority tasks that must get done but whose time of completion is less important. This "free" bandwidth is extracted by scheduling the low-priority requests between high-priority ones so that time normally spent on rotational latencies gets used for background reads and writes.

Typical disk use generally requires the disk heads to spend a relatively large amount of time seeking the desired track and waiting for the desired sectors to rotate to the disk head. These are the seek time and the rotational latency, respectively. The seek time is inevitable. However, the rotational latency can be used for other reads without affecting the time the original request would take. The authors call this process *freeblock scheduling*.

The effectiveness of freeblock scheduling relies on the ability to find background requests that fit well between the foreground requests. Tasks that are most appropriate for this are processes that are low priority, have large sets of desired blocks, require no particular order of access, and have small working memory footprints. Applications that fit these requirements include those that perform scanning, internal storage optimization, or prefetching and prewriting.

The actual results published in the paper are very promising. They demonstrate that for a process that wants to read the entire disk in the background, the full potential free bandwidth (35–40% of total potential) can be utilized until over 90% of the disk has already been scanned, and even until the entire disk has been scanned, over half of the potential free bandwidth can actually be utilized. They also show increases of disk bandwidth utilization on the order of 10 times that of the original utilization.

It remains to be seen how much of this scheduling can be done outside of modern disk drives, given the complexity of their internal algorithms and the lack of low-level interfaces. If freeblock scheduling indeed proves to be compatible with modern disk-drive technology, it could be very beneficial and could significantly increase disk bandwidth.

Latency Management in Storage Systems

Rodney Van Meter, Quantum Corporation; Minxi Gao, University of California, Berkeley

Summarized by Vijay Gupta

Rodney Van Meter indicated that the primary motivation for this work was the 11-orders-of-magnitude difference between latency of access to a tape and that of access to RAM.

To address the above problem, the authors proposed the concept of *Storage Latency Estimation Descriptors (SLEDs)*. An API, SLEDs allow applications to understand and take advantage of the dynamic state of the storage system. SLEDs are complementary to the notion of hints (which are used in transparent informed prefetching). Whereas hints are given by applications to the OS, SLEDs are given by the OS to the applications.

Van Meter motivated the use of SLEDs by giving the example of an application which makes two sequential passes over a file. Suppose there are three pages in the buffer, and the file size is five pages. If the page replacement strategy is LRU, then the second pass will have five hard faults. On the other hand, SLED reorders reads in the second pass so that there are only two hard faults. Thus reordering I/Os yields large gains.

The SLEDs were added to v2.2.12 of the Linux kernel. The authors added new ioctl options which return SLEDs data. In addition, they modified several UNIX utilities such as `find`, `wc`, `grep` and `GMC` (which is a GUI file manager) to assess the benefit of SLEDs. `wc` reorders I/O; `grep` reorders and prunes directory search trees; `find` uses their `-latency` predicate. The feature of the `-latency` predicate is that if the latency to access some portion of the file system is going to be beyond the specified latency, then that portion of the file system would be skipped.

Then Van Meter presented a large, complex example of an astronomy application (LHEASOFT) which is made up of 840,000 lines of C and Fortran. It has a 100,000-line I/O library. The application was modified to reorder the I/O operations. This reordering achieved 11–25% reduction in execution time in spite of the fact that the program was already optimized with the I/O library.

Overall, the paper raised some very important issues for heterogeneous systems, which are becoming increasingly common.

A Low-Overhead, High-Performance Unified Buffer Management Scheme that Exploits Sequential and Looping References

Jong Min Kim, Jongmoo Choi, Jesung Kim, Sang Lyul Min, Yookun Cho, and Chong Sang Kim, Seoul National University; Sam H. Noh, Hong-Ik University;

Summarized by Tamara Balac

This talk focused on a new solution to an old problem. The problem is determining which pages the OS should cache in the main memory. The motivation for their solution approach was that LRU has problems for sequential and looping references. If the access is purely sequential, then caching recently used pages is wasteful. If the access is looping, then LRU cannot figure it out. To overcome this difficulty, they proposed a new concept called *Inter-Reference Gap (IRG)*. IRG for a block is the difference between the points when the block is successively accessed.

The types of references which the authors considered are: *sequential*, *looping*, and *other*. They developed a new scheme, called *unified buffer management (UBM)*, which includes *automatic detection*, *replacement*, and *allocation*, to exploit these references.

In automatic detection, they take into account "fileID, start, end, period" for references. Initially, the period is infinity. As more references are encountered, the period is adjusted. For sequential access, the period always stays at infinity. But in the case of looping references, there is a definite period. For replacement, they adopt different schemes depending on the type of reference. For sequential access, they use MRU; for looping references, they use a period-based replacement scheme; and for other references, they use LRU. The allocation scheme is a bit too mathematical to explain here, although the intuition involves using marginal gains and IRG.

To wrap up, Noh showed the results for trace-driven simulations and showed graphs for how UBM takes into account the looping references.

SESSION: SECURITY

How to Build a Trusted Database on Untrusted Storage

Umesh Maheshwari, Radek Vingralek, and William Shapiro, STAR Lab, InterTrust Technologies Corporation

Summarized by Tamara Balac

Digital rights management protects rights of the provider (i.e., database balances, contracts). Existing systems are missing trusted storage in bulk. Umesh Maheshwari presented a Trusted DataBase system (TDB) that resists accidental and malicious corruption by using Crypto Basis, which leverages persistent storage in a trusted environment.

The TDB architecture consists of three layers: a Collection Store, an Object Store, and a Chunk Store. The Chunk Store provides trusted storage for variable-sized sequences of bytes which are the unit of encryption and validations (100B–10KB). The Object Store manages a set of named objects. The Collection Store manages a set of named collections of objects.

Performance evaluation demonstrated that Crypto overhead is small compared to I/O and that TDB performs well compared to commercial packages (e.g., XDB).

End-to-End Authorization

Jon Howell, Consystant Design Technologies; David Kotz, Dartmouth College

Summarized by Mac Newbold

In a very entertaining presentation, Jon Howell explained the need for an end-to-end authorization scheme. Currently, when a local user needs to grant access to a resource to a non-local user, it creates an authentication problem, because the server doesn't know about the non-local user. Typically this is solved by creating an account locally for the non-local user by sharing passwords, or by installing a gateway that is implicitly trusted by the server, all of which we know have many weaknesses. The solution proposed by Howell is a system for delegating authority in a way that the server has a complete proof of correctness and an audit trail for the access the remote user was given.

The system is based on delegations. For instance, local user Alice wants to give remote user Bob access to some of her files. So she delegates her authority over those files to Bob. Then when Bob asks for file X in that set of files, the server is presented with a collection of statements: first the request, "*Bob wants to access file X*," then the delegation, "*Bob speaks for Alice concerning file X*," and finally the basis for delegation, "*Alice owns file X*." The server then can make a decision about allowing Bob to access X, instead of relying on one or more gateways to decide. In this scheme, gateways are required only for translation and relay of requests and proofs. Of course, at this point, the client trusts the gateway not to abuse its authority. To deal with this, a gateway authentication scheme can be used.

One advantage of a system like this is that the gateway is very simple. It only needs to carefully quote each request and only use Alice's authority for Alice's requests. It need not make access decisions. This also allows for multiple gateways to bridge the gap between client and server, and ultimately the server is the one who grants or denies access.

The implementation of the authorization scheme uses Simple Public Key Infrastructure (SPKI) and is part of the Snowflake project. The paper includes performance evaluations that reflect some additional overhead for the end-to-end authorization but points out that a large part of the added overhead is directly attributed to slow and inefficient

SPKI libraries. It is estimated that optimized SPKI libraries would perform almost as well as a Java and Java SSL implementation of HTTP authorization. Because Snowflake performs steps that very closely correspond to those performed by SSL, most of which are computationally expensive cryptographic operations, it is expected that an optimized Snowflake would perform as well as SSL plus a small overhead for the proving steps that are not performed by SSL. These results show that their technique is potentially a very valuable resource for end-to-end authentication.

Self-Securing Storage: Protecting Data in Compromised Systems

John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, Gregory R. Ganger, Carnegie Mellon University

Summarized by Tamara Balac

Intrusions are a fact of modern computing. What are system administrators to do? Diagnosis and recovery. Nevertheless, the major problem with diagnosis and recovery is that intruders can manipulate ALL stored information.

John Strunk presented an implementation of a *Self-Securing Storage System,* named S4, that prevents undetectable modifications by providing a complete history of all modifications. S4 is a separate piece of hardware that runs a separate operating system. The next step is to add internal reasoning and auditing, by creating a new version (called collections) on every write, and to store these for a guaranteed amount of time (called detection window).

The benefits of S4 include providing an opportunity to analyze security compromises, enabling speedy recovery, and allowing recovery from accidents (like accidental file modifications).

Diagnostic comparison of S4 and conventional systems showed that conventional systems use guessing, while S4's audit log shows the sequence of storage events. Additionally, an S4 administrator can recreate the state of the storage at any point of time.

Feasibility evaluation showed that large detection windows, even multi-week detection windows, are possible. More importantly, the performance overhead was less than 15%.

Fast and Secure Distributed Read-Only File System

Kevin Fu, M. Frans Kaashoek, and David Mazières, MIT Laboratory for Computer Science

Summarized by Vijay Gupta

The motivation for this talk was that Internet users increasingly rely on publicly available data for everything from software installation to investment decisions. Unfortunately, the vast majority of public content on the Internet comes with no integrity or authenticity guarantees. This work uses a secure file system (SFS) that was built by the authors and presented at SOSP 1999.

David Mazières started off by providing an example of installing an OS over the network. He also gave reasons why people avoid security: performance, scalability, reliability, and convenience. Another issue in a distributed system is that the more replicas you have, the greater the chance of a break-in. To mitigate this problem, people resort to ad hoc solutions. As an example, lots of software packages contain PGP signatures. The problem with PGP is that it is not general purpose; most users ignore signatures, and it requires the continued attention of the user.

So, they propose a *self-certifying read-only file system* , which has been designed to be widely replicated on untrusted servers. This acts as a content distribution system providing secure, scalable access to public, read-only data. With this, one can publish any data. It is convenient because you can access it from any application. It is scalable because publishing has been separated from the distribution infrastructure.

In their approach, an administrator creates a database of a file system's contents and digitally signs it offline using the file system's private key. The administrator then widely replicates the database on untrusted machines. Client machines pick their data from these machines and before data is returned to the applications, SFS checks the authenticity of data. The good thing about their approach is that no cryptographic operations are performed on servers, and the overhead of cryptography on the clients is low. This is accomplished using collision-resistant cryptographic hashes. For details about the scheme, the interested reader is referred to the paper.

The read-only file system is implemented as two daemons (sfsrocd and sfsrosd in the SFS system). A performance evaluation shows that sfsrosd can support 1,012 short-lived connections per second on a PC, which is 92 times better than a secure Web server. Finally, Mazières also mentioned the necessity of periodically re-signing data and putting them on to the server to prevent break-ins.

For more information, see <*http://www.fs.net*>.

SESSION: NETWORKING

Overcast: Reliable Multicasting with an Overlay Network

John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, James W. O'Toole Jr., Cisco Systems

Summarized by Mac Newbold

Overcast addresses many of the problems with current multicasting solutions in the Internet. One option is IP multicast, but it has several weaknesses. It can only be used for live transmission, has no delivery guarantees, requires support in routers, servers, and clients, and makes accurate billing and good security nearly impossible. Another alternative is a con-

tent distribution system, which provides on-demand content. But it doesn't perform so well with live content, often is not scalable, and doesn't allow for nodes to be added or deleted on the fly.

John Jannotti indicated that Overcast provides a unified reliable multicast solution. It is self-organized, handles live, time-delayed, and on-demand content, and any HTTP client can join without modification. As an overlay network, it is also incrementally deployable and requires no special support from the underlying network or its routers. It uses HTTP over TCP port 80 and has other features that make it compatible with NATs, firewalls, and HTTP proxies.

One key to the Overcast design is the process of building an efficient multicast distribution tree. The source of the multicast is designated as the root of the tree, and any nodes that want to join contact that root node and attach to the tree as its child. Periodically, each node considers its "sibling" nodes and "grandparent" nodes as possible new parents. If it finds that its connection to a sibling node is better than its connection to its current parent (for instance, in terms of latency, bandwidth, or hop counts), it makes that sibling node its new parent. It could also find that a connection directly to its grandparent would be a better connection than through its current parent, and it would, in effect, become a sibling to its parent node. This protocol makes the topology flexible when faced with changing network conditions.

Overcast uses only "upstream" connections when contacting other nodes; that is, the child always must establish a connection with the parent. This ensures that HTTP proxies and NATs do not interfere with Overcast functionality. For this reason, an up/down protocol was created for maintaining information about node status. It requires each node to check in with its parent periodically, and if it misses a check-in, it is considered dead. Through the use of "death certificates" and "birth certificates," it notifies the hierarchy of changes in topology. A key to scalability here is a system for quenching messages that aren't necessary for nodes further up the hierarchy. Overcast network usage for these messages scales sublinearly, and space usage on the root node is linear, but even in a group of millions of nodes, total RAM cost for the root would be under $1,000.

Jannotti referred to the paper which also outlines a system for replicating the root node to increase reliability of the root server itself. The system uses techniques used by normal redundant server setups, such as DNS redirection or round-robin load balancing and, in the case of a failed server, IP address takeover. The Overcast-specific solution is to replicate root state by setting up the servers linearly, with each replicated root being the only child of the root node above it, so that the rest of the hierarchy is descended from each of the root nodes. This way when one fails, another can immediately take over without any interruption of service or loss of state.

Jannotti concluded that the reliable multicast solution proposed in Overcast is a very feasible solution in terms of deployability, scalability, efficiency, flexibility, and usability in real-world situations.

System Support for Bandwidth Management and Content Adaptation in Internet Applications

David Andersen, Deepak Bansal, Dorothy Curtis, and Hari Balakrishnan, MIT Laboratory for Computer Science; Srinivasan Seshan, Carnegie Mellon University

Summarized by Vijay Gupta

David Andersen opened his talk by saying that the primary motivation of the work was to facilitate end-to-end congestion control. TCP uses an additive increase, multiplicative decrease (AIMD) scheme for congestion control. That's wonderful for FTP and email, which use just one TCP connection. But HTTP uses four connections in parallel between the same two endpoints in the Internet. Furthermore, not all applications necessarily need the reliability of TCP, so such applications use UDP. To make them TCP-friendly, they end up using some home-grown congestion control. The goal of this work is to have some kernel-level mechanisms to facilitate TCP friendliness.

Andersen showed where the congestion controller occurs in the Linux kernel, where they implemented their scheme. They use callbacks for orchestrating transmissions and application notification. The clients for these callbacks are in-kernel TCP clients.

The implementation handles flow control and congestion control. It also has a user-level rate callback API, called libcm, which tells if bandwidth goes up by some factor (e.g., a factor of 2).

Andersen addressed evaluation issues such as the impact on the network and on the connections. Their approach has a positive effect on the TCP friendliness of host-to-host flows, although the throughput of the connections may be slightly worse.

For testing the flow integration, they used a series of Web-like requests on the Utah testbed (see <*http://www.cs.utah.edu/flux/testbed/*>). Andersen also presented graphs to show that congestion manager (CM) is efficient, and he then showed some results for layered MPEG-4 (which is not in the paper). He also said that implementing an adaptive visual audio toolkit (vat) was very trivial using the CM toolkit.

For more information, see <*http://nms.lcs.mit.edu/projects/cm/*>.

SESSION: STORAGE DEVICES

OPERATING SYSTEM MANAGEMENT OF MEMS-BASED STORAGE DEVICES

John Linwood Griffin, Steven W. Schlosser, Gregory R. Ganger, and David F. Nagle, Carnegie Mellon University

Summarized by Vijay Gupta

John Griffin began the talk with an overview of *microelectromechanicals* (*MEMS*) and MEMS-based storage devices. MEMS is a new storage technology currently under development in industry and academia. Real-life systems such as car airbags already use them. The results in this paper are based on collaboration with the MEMS lab at CMU.

The real advantage of MEMS-based storage devices is that the seek time is an order of magnitude less than with disks. The purpose of the talk was to show that work on disk-scheduling algorithms is also applicable to MEMS-based storage devices. This was shown through a variety of graphs which had the same shape for disk-scheduling and MEMS-based storage.

An interesting aspect of MEMS-based storage is that the access is faster for data in the center, and slower for data at the borders. So the authors suggest that the center be used for metadata and small objects, and that the border be used for large-streaming media objects.

During Q&A, one person asked about price/GB, and Griffin said that it might be cheaper than hard disk. Someone else asked when we could see them in real systems, and he replied that it could be expected in a few years. For power requirements, he referred to their upcoming ASPLOS paper, while for MTTF, Griffin responded that he did not have an answer.

For more information, see <*http://lcs.Web.cmu.edu/research/MEMS*>.

TRADING CAPACITY FOR PERFORMANCE IN A DISK ARRAY

Xiang Yu, Benjamin Gum, Yuqun Chen, Randolph Y. Wang, Kai Li, Princeton University; Arvind Krishnamurthy, Yale University, Thomas E. Anderson, University of Washington

Summarized by Vijay Gupta

Large disks are now available for a fraction of what they used to cost, and hence, one can be more creative about how to use them. In this paper, Randolph Wang proposes a way to reduce the usage level of disks.

Since the access times between memory and disks keeps on increasing, RAID systems were made to improve the read/write throughput of the systems and to improve reliability. However, this work goes beyond RAID by contributing an SR-array, which flexibly combines striping with rotational replication to reduce both seek and rotational delay. The average seek distance becomes less than one-third of the ratio of the maximum to average rotational delay, and the rotational delay for reads is reduced to half.

Wang went on to show some of the equations which they derived as part of their theoretical model. They have a prototype MimdRAID implementation (which is a simulator) that puts the theory to test. The bottom line was that the MimdRAID prototype can deliver latency and throughput results unmatched by conventional approaches.

For more information, see

<*http://www.cs.princeton.edu/~rywang/mimdraid*>.

INTERPOSED REQUEST ROUTING FOR SCALABLE NETWORK STORAGE

Darrell C. Anderson, Jeffrey S. Chase, Amin M. Vahdat, Duke University

Summarized by Mac Newbold

Jeff Chase described a new storage system architecture called Slice. It takes advantage of high-speed networks to interpose a request switching filter, a microproxy or μproxy, and presents an NFS interface to clients that has a back end which scales well in both bandwidth and capacity.

Because of recent advances in LAN performance, a specialized Storage Area Network (SAN) with faster network connections (like Fibre Channel) is no longer required, and similar approaches can be used in a LAN environment to provide scalable network storage. The Slice file service is a group of servers that cooperate to provide an arbitrarily large "virtual volume" to a client, who sees it as a single file server. Client requests are separated into three classes: high-volume I/O to large files, I/O on small files, and operations on namespace or file attributes. This diverts high-volume data flow around manager nodes and allows specialization of the servers for each type of data.

The μproxy handles all bulk I/O requests and was designed to be small, simple, and fast. It has been implemented as a loadable packet filter module for FreeBSD. It may rewrite source or destination addresses or other fields in request and response packets. It maintains a bounded amount of soft state that is not shared across clients, so it can easily be placed on the client itself, in a network interface, or in a network element close to the storage servers. All of its functions can be replicated freely to provide scalability, with the constraint that requests for a given client all pass through the same μproxy. The μproxy routes requests directly to the storage array without any further intervention by management nodes.

The storage nodes themselves use an object-based method as opposed to sector-based. This feature lets the μproxy be located outside of the server's trust boundary and use encryption to protect object identifiers, limiting damage from a compromised μproxy to those files and directories that its clients have permission to access. Slice is also compatible

with sector-based storage if every μproxy is trusted. Redundancy can also be provided at two levels, either internally to each storage node or across nodes through mirroring and striping. It can be configured on a per-file basis, and a Slice configuration could even use redundancy at both levels for stronger protection.

The two types of management nodes, the directory servers and the small file servers, take load off of the storage nodes and allow for further specialization for these types of operations. These managers are data-less, and all their state comes from the storage arrays, so they provide only memory and CPU resources to cache and manipulate the structures. The directory server handles all lookups, creation, renaming, and deletion of directories and files and their attributes. The small file servers provide more efficient space allocation and file growth, as well as batching of multiple small requests into larger ones for efficient disk writes. The managers also help provide atomicity and recovery features through a write-ahead log and two-phase commits.

Performance data indicates that Slice does indeed scale very well. Name-intensive benchmarks showed directory service can be improved simply by adding more directory server sites, and for any given configuration, the performance scales linearly with the number of clients. Performance was evaluated with the industry-standard SPECsfs97 benchmark, and Slice kept up perfectly with the offered load up to its saturation point, which can be easily raised by adding more storage nodes.

Slice definitely provides a network storage solution that is scalable, reliable, practical, easy to upgrade, and comparable to similar commercial solutions in terms of performance. With benefits like these, we are sure to hear more about Slice in the near future.

For more information, see <*http://www.cs.duke.edu/ari/slice/*>.

SESSION: RELIABILITY

Proactive Recovery in a Byzantine-Fault-Tolerant System

Miguel Castro and Barbara Liskov, MIT Laboratory for Computer Science

Summarized by Tamara Balac

Today's computer systems provide crucial information and services which make them more vulnerable to malicious attacks and make the consequences of these attacks, as well as software bugs, more serious. As an alternative to the usual technique of rebooting the system, this paper proposes a new means of system recovery that does not use public key cryptography.

Miguel Castro presented an asynchronous state-machine replication system that offers both integrity and high availability and is able to tolerate Byzantine faults which can be caused by malicious attacks or software errors. The paper presents a number of new techniques, like proactive recovery of replicas, fresh messages, and efficient state transfer, needed to provide good recovery service.

The task of recovery from Byzantine faults is made harder by the fact that the recovery protocol itself needs to tolerate other Byzantine-faulty replicas. Attackers must be prevented from impersonating recovered replicas. The advantage of this system over previous state-machine replication algorithms is the use of symmetric cryptography for authentication of all protocol messages, which bypasses the major public key cryptography bottleneck.

This algorithm has been implemented as a simple interface, generic program library that can be used to provide Byzantine-fault-tolerant versions of different services.

For more information, see <*http://www.pmg.lcs.mit.edu*>.

EXPLORING FAILURE TRANSPARENCY AND THE LIMITS OF GENERIC RECOVERY

David E. Lowell, Western Research Laboratory, Compaq Computer Corporation; Subhachandra Chandra and Peter M. Chen, University of Michigan

Summarized by David Oppenheimer

David Lowell described the abstraction of "failure transparency," in which an operating system provides the illusion of failure-free operation by automatically recovering applications after hardware, operating system, or application failures without explicit programmer assistance. His work proposes two invariants that must be upheld to achieve failure transparency. The "save-work invariant" specifies what application state must be preserved to mask the failures. The "lose-work invariant" specifies what application state must be discarded to allow recovery from failures that affect the application state.

Lowell's theory defines a space of recovery protocols. Each point in the space represents a different technique for upholding save-work. One axis of this space is the effort made to commit only visible events, while the other is the effort made to identify and convert non-deterministic events. Lowell mapped a number of protocols onto this space and discussed how performance, simplicity, reliability, recovery time, and other design variables vary over the space.

Lowell presented performance results, using "save-work invariant," for four applications: nvi, magic, xpilot, and TreadMarks. He ran each application on a number of protocols from the protocol space. For these applications he found an overhead of 0–12% when his recovery system used reliable memory as stable storage. He found overhead of 13–40% for the interactive workloads when using disk as stable storage.

In addition to nvi, Lowell used postgres for measuring performance with "lose-work invariant." By injecting faults into nvi and postgres, Lowell also measured the fraction of application faults that violate lose-work by committing after the fault is activated. He found that upholding save-work for these applications caused them to violate lose-work in at least 35% of application crashes from non-deterministic faults. Merging this result with published fault distributions, he estimated that perhaps greater than 90% of application crashes in the field violate lose-work, making application generic recovery impossible in those cases.

Because operating system faults often do not manifest as propagation failures, OS faults cause violation of lose-work less frequently than application faults. Lowell presented fault-injection study results in which nvi and postgres violated lose-work in 15% and 3% of OS crashes, respectively.

Lowell concluded that for stop failures, which do not require upholding lose-work since by definition they crash the application before corrupting any application state, low-overhead failure transparency is possible for many real applications. Recovering from propagation failures is much more difficult because upholding save-work often forces violation of lose-work. From this study, Lowell concluded that providing failure transparency for stop failures is feasible, but that recovery from propagation failures cannot be accomplished transparently and must involve help from the application.

During Q&A, conference attendees asked about the feasibility of rebooting applications as a mechanism for stopping a program before an error has propagated, and the difference in the chance of violating lose-work for hardware failures as opposed to software failures.

For more information, see *<http://www.eecs.umich.edu/Rio>*.

DESIGN AND EVALUATION OF A CONTINUOUS CONSISTENCY MODEL FOR REPLICATED SERVICES

Haifeng Yu and Amin Vahdat, Duke University

Summarized by David Oppenheimer

Amin Vahdat described the design and evaluation of TACT, a continuous consistency model for replicated services. This model proposes a continuous range of consistency options for distributed applications, rather than simply the traditional strong and optimistic concurrency policies. By proposing a set of metrics to quantify the consistency spectrum — numerical error, order error, and staleness — TACT allows the investigation of tradeoffs among consistency, availability, and performance as consistency policies are varied in the space between strong and optimistic concurrency.

Applications that use TACT specify their desired consistency semantics using *conits*. A conit is a three-dimensional vector associated with each application-specific physical or logical unit of consistency (e.g., a block of seats on a flight in a distributed airline reservation system). The three elements of a conit are numerical error, which bounds the discrepancy between the local value of a piece of data and the value in the "final image" of the data; order error, which bounds the difference in the order in which updates are applied to a local replica and the ordering of those updates in the "final image"; and staleness, which specifies a maximum amount of time before a non-local write is accepted to be applied locally. Bayou-style anti-entropy is used as the mechanism for maintaining consistency among replicas.

TACT is implemented as a middleware layer that enforces the consistency bounds specified by the application's conits. It allows applications to dynamically trade consistency for performance based on service, network, and request characteristics. Three systems have been

built using the TACT platform: a bulletin board, an airline reservation system, and a system for enforcing quality of service (QoS) guarantees among distributed Web servers. For these systems Vahdat evaluated such issues as latency for posting a bulletin board message as a function of the numerical error bound; reservation conflict rate, throughput, and reservation latency as a function of inconsistency in the airline reservation system; and number of consistency messages as a function of relative error for the distributed Web server QoS system.

During Q&A, Vahdat indicated that his group is currently investigating issues such as how responsive the system is to rapid variation in desired consistency levels and how much effort is required by a programmer to incorporate conits into an application.

For more information, see <*http://www.cs.duke.edu/ari/issg/TACT/*>.

SESSION: SYSTEM ARCHITECTURE

Scalable Distributed Data Structures for Internet Service Construction

Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler, University of California, Berkeley

Summarized by Mac Newbold

Building and running a cluster-based Internet service is hard. Steven Gribble explained the implementation of a Distributed Data Structure (DDS) that is designed to be a reusable storage layer for Internet services. The goals of the DDS are to be scalable, highly available and reliable in the face of failures, to maintain strict consistency of distributed data, and to provide operational manageability. In particular, they have designed and implemented a distributed hash table that meets these goals.

The basic design starts with a cluster. This provides low latency, redundancy, and makes a two-phase or multiple round-trip system feasible. All instances of the service see the same data structure, so any client can work with any server for any transaction, which simplifies load balancing and request routing.

This design provides incremental scalability as nodes are added to the cluster and was tested up to terabytes of storage space. Because individual nodes and disks might fail, each partition of data is replicated on multiple nodes, forming a replica group, which are all kept strictly coherent. Any replica can service a data lookup, but any state changes must happen in all replicas.

A simple algorithm is also in place to provide fault tolerance. If a replica crashes, it is simply removed from the replica group and operation continues. When a node joins or rejoins a replica group, the partitions that it will duplicate are copied from existing replicas, and the node is added to the replica group. Individual partitions are kept small (approximately 100MB), so that an entire data partition can be copied in 1 to 10 seconds (given a 100Mbps to 1Gbps network). The partition to be copied is locked by the joining node, copied, and the replica group maps are updated, and the locks are released. Any write operations on that partition will have failed during that time, but after the lock is released, retries will succeed.

Performance data shows that the maximum throughput of the DDS scales linearly with the number of bricks. Read operations also scale linearly up to the saturation point, where read throughput plateaus, but again, by adding bricks, the throughput is increased. Write operations run into problems, however, because garbage collection ends up causing an imbalance between the nodes in a replica group, and because the writes have to happen on all replicas, the slow one becomes the bottleneck. Even performance in recovery was very promising; an N-brick DDS during a single failure and recovery appeared to yield performance near to that of an (N-1)-brick DDS, except that the partitions on the failed brick were available only for reading.

An example of the usefulness of DDS for rapid construction of Internet Services is Sanctio, an instant messaging gateway. It translates between ICQ, AOL's AIM protocol, email, and voice messaging over cellular phones. It also uses AltaVista's BabelFish to do language translation. During his presentation, Gribble told of using Sanctio to translate his English ICQ connection to an Italian AIM connection to communicate with a friend's grandmother in Italy. Using DDS for its storage, Sanctio was completed in less than one person month, and the code that interacts with the DDS took less than a day to develop.

This work shows that a Distributed Data Structure can provide a scalable, highly available, reliable, consistent, and easy-to-use interface to network storage, and shows its usefulness for constructing Internet services.

Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java

Godmar Back, Wilson H. Hsieh, and Jay Lepreau, University of Utah

Summarized by Tamara Balac

Godmar Back described the design and implementation of KaffeOS, a Java virtual machine that supports the operating system abstraction of a process and provides the ability to isolate applications from each other or to limit their resource consumption and still share objects directly. Processes enable several important features. First, the resource demands for Java processes can be accounted for separately, including memory consumption and GC time. Second, Java processes can be terminated, if their resource demands are too high, without damaging the system. Third, termination reclaims the resources of the terminated Java process.