VEE '05: First ACM/USENIX Conference on Virtual Execution Environments

June 11–12, 2005

KEYNOTE ADDRESS

Summarized by Shuo Yang, Long Fei, and Xing Fang

- *A Unified View of Virtualization*

  James E. Smith, University of Wisconsin

  James Smith started his keynote by discussing why virtualization techniques are interesting: they enable transcending interfaces, flexible innovation, adaptation to other software/hardware, networked computing, and enhanced security. He views virtualization as the fourth pillar in computer systems besides hardware, system software and application software.

  He reviewed the domains and examples of virtual machines, and the origins of virtual machine concepts. He said that different interest groups, such as OS developers, compiler developers, and application programmers have different perspectives on virtual machines. System virtual machines provide a system environment that is constructed at the ISA level. Process virtual machines are constructed at the ABI level. High-level-language virtual machines (HLL VMs) provide APIs, i.e., they raise the level of abstraction.

  He pointed out that many of the existing VM techniques were invented on an *ad hoc* basis, and he asked whether there might exist something more fundamental than a collection of techniques. He then discussed his idea of how to establish uniform concepts. He also reviewed the solutions and challenges of VM techniques. From there, he put forward the question of how to enhance virtual machine performance. For example, the increase of system layer complexity leads to inefficiency. The Intel VT-x solution is to add another set of layers. He then presented some performance primitives, such as code cache support, visibility of micro-ops, and so forth.

  Smith reminded the audience that killer applications motivate virtualization techniques. One of the killer apps for virtualization is security. HLL VMs, with security features built in; process VMs, whose code can be inspected before being executed; and system VMs, which provide isolation with simple VMMs, can all be used to support secure networked computing.

  Smith also discussed the education curriculum about VMs. He proposed an outline of what to teach in a VM course, for example, putting together virtually all levels of computer system hardware and software. He believes it is a challenging and necessary task.

  Smith concluded the talk with the following points: the common framework and terms should be resolved; virtualization needs higher concepts, rather than a bag of terms and techniques; and a unified conference to exchange ideas from different communities is needed, and VEE serves that goal perfectly.

SCALABILITY, PERFORMANCE, AND REAL-TIME

Summarized by Shuo Yang

- *Friendly Virtual Machines: Leveraging a Feedback-Control Model for Application Adaptation*

  Yuting Zhang, Azer Bestavros, Mina Guirguis, Ibrahim Matta, and Richard West

  Azer Bestavros presented a Friendly Virtual Machines (FVM) framework for efficient and fair resource allocation when sharing an underlying host system. Bestavros first discussed the background of virtual machine adaptation: the trend of VM techniques being increasingly adapted to support applications running on third-party hosts, the need to isolate independent constituents, and the emergence of VM abstractions. He then said that the motivation of the research is to ensure fairness and efficiency in the underlying host resource allocation. He advocated the use of self-adaptation in the guest VMs themselves, based on feedback about resource usage and availability.

  Bestavros defined a virtual machine that fairly and efficiently adjusts its demand for system resources as a Friendly VM (FVM) and proposed a resource-sharing technique that is applicable to any application whose execution is "friendly" to other applications sharing the same underlying resources. The friendliness feature of FVM applies the classical end-to-end argument to the problem of multi-resource allocation across a set of applications sharing the same infrastructure.

  Bestavros said that the host in FVM needs to provide unbiased on-demand resource allocation and VMs, and he mentioned the pricing issues enabled by the FVM framework. Bestavros showed the experimental result of resource utilization using the FVM system with "made-up" benchmarks and real Web server benchmarks. The results showed that FVM successfully achieves fairness and efficiency in sharing common hosting resources.

- *Diagnosing Performance Overheads in the Xen Virtual Machine Environment*

  Aravind Menon, Jose Santos, Yoshio Turner, G. (John) Janakiraman, and Willy Zwaenepoel

  Yoshio Turner presented Xenoprof, a systemwide statistical profiling toolkit for the Xen virtual machine environment. Turner first discussed how the increased adaptation of virtualization techniques can affect application performance in unexpected ways. He then presented an example of Web server performance degradation under the Xen system, which motivated their Xenoprof project. An application's performance in a virtual machine environment can differ markedly

from its performance in a non-virtualized environment, because of interactions with the underlying VMM. Xenoprof is designed to enable coordinated profiling of multiple VMs in a system to obtain the distribution of hardware events. He introduced the Xenoprof design: a paravirtualized interface to support domain-level profilers. OProfile has been ported to Xen environment as a domain-specific profiler.

Turner presented an example of Xenoprof use, showing how they found a TCP-receive performance anomaly under XenoLinux. After that, he presented the work done with Xenoprof to analyze several performance problems observed under different Xen configurations for receiver, sender, and Web server applications. Turner concluded that Xenoprof is a useful tool to identify major overhead in Xen. Xenoprof will be included in official Xen and OProfile releases.

- ***Supporting Per-Processor Local-Allocation Buffers Using Lightweight User-Level Preemption Notification***

  Alex Garthwaite, Dave Dice, and Derek White

  Alex Garthwaite presented a local-allocation buffers (LAB) management technique that supports local buffer allocation with regard to processors instead of threads.

  Garthwaite first gave a performance comparison under different LAB-size policies for the VlanoMark benchmark. He made the point that garbage collection is a hard problem when there are more threads than processors and high preemption rates.

  He then presented a processor-local allocation buffers (PLABs) strategy that associates local allocation buffers (LABs) with processors and with buffers for each thread allocated from its processor's LAB. Multi-processor restartable critical sections (MP-RCS) techniques implement such a buffer allocation strategy. He then introduced the challenges of PLABs, such as the cost of dynamic checks and the need for expression translation in C. There is an overhead for using PLABs over thread-local allocation buffers (TLABs) when the number of threads is less than the number of processors, or when threads are entirely compute-bound. PLABs are much better than TLABs when the number of threads is larger than the number of processors. He showed that their mechanism can combine the PLAB and TLAB adaptively. Garthwaite said that their mechanism can easily be implemented in x86 architecture. He then presented the experimental results of their techniques and concluded that the simple MP-RCS mechanism is independent of the threading/scheduling model and is applicable to many platforms.

- ***A Programmable Microkernel for Real-Time Systems***

  Christoph Kirsch, Marco Sanvido, and Thomas Henzinger

  Marco Sanvido presented the microkernel system architecture for hard real-time applications. He first presented the reactive (response to environment) and the proactive (task scheduling in platforms) requirement in embedded systems, and introduced the concept and architecture of their solution's design. The E (embedded) machine is a virtual machine that triggers the execution of software tasks with respect to events. The S (scheduling) machine is a virtual machine that orders the execution of software tasks, and together their E+S machines equal the microkernel model. Their model represents the abstraction of the interaction between the hardware platform, reactively constrained by the E machine and proactively constrained by the S machine.

  Sanvido talked about the time-safety requirement of hard real-time applications and presented a proof that the time-safety requirements were fulfilled with schedule-carry code in their system. He introduced the implementation issues of the E+S machine, which has been implemented using the StrongARM processor and integrated into HelyOS. In conclusion, Sanvido said that their work has adopted microkernel architecture for the real-time application domain.

OBJECTS AND THEIR COLLECTION

Summarized by Long Fei

- ***The Pauseless GC Algorithm***

  Cliff Click, Gil Tene, and Michael Wolf

  Cliff Click began by pointing out that garbage collection response time has become an important problem for applications that contain response-time-sensitive components. He described a system, including CPU, chip, board, and OS, built by Azul Systems to run garbage-collected virtual machines. The hardware supports fast user-mode trap handlers. The hardware TLB supports an additional privilege level, GC mode, which lies between the usual user and kernel modes. TLB violations on GC-protected pages generate fast user-level traps instead of OS-level exceptions. The CPU supports a read barrier instruction, which resembles a standard load instruction except that if it refers to a GC-protected page a fast user-mode trap (GC-trap) handler is invoked.

  The GC algorithm is highly concurrent, parallel, and compact. The algorithm is divided into three phases, Mark, Relocate, and Remap. The Mark phase is responsible for periodically marking the live and dead objects. The Relocate phase finds pages with little live data, to GC-protect, relocate, and compact them and to free the backing physical memory. The Remap phase updates every relocated pointer in the heap. During the relocating phase, if a mutator's read-barrier GC-traps, the GC-trap handler looks up the forwarding pointer and places the correct value both in the register and in memory.

The authors backed up their design with experiments using a modified version of SpecJBB benchmark. The authors also state that the read barrier behavior can be emulated on standard hardware at some cost.

- ***Use Page Residency to Balance Tradeoffs in Tracing Garbage Collection***

  Daniel Spoonhower, Guy Blelloch, and Robert Harper

  Daniel Spoonhower presented this paper. The key innovation of the paper is a mechanism that allows the collector to dynamically balance the tradeoffs of copying and non-copying collection for each page based on page residency, a measure of the density of reachable objects on a page. If the residency of a page is sufficiently high, the page should be promoted, otherwise it should be copied.

  Measuring the residency of even a single page requires a traversal of the entire heap. To avoid this overhead, the authors devised several residency-prediction heuristics and recovery mechanisms to handle poor predictions. The authors also identified a continuous range of tracing collectors and showed that classic GC algorithms can be considered special cases of the proposed GC algorithm with extreme residency assumptions.

  Their experiments revealed the impact of heap size and configuration thresholds on the performance of GC algorithms. Mark-sweep performs better when the heap size is small, whereas semi-space performs better when the heap size is large. In both cases, their algorithm yields a performance close to the better of these. Experiments show that this new algorithm has only a small variation in performance under six different configurations.

- ***Exploiting Frequent Field Values in Java Objects for Reducing Heap Memory Requirements***

  Guangyu Chen, Mahmut Kandemir, and Mary J. Irwin

  Guangyu Chen said that the capabilities of applications executing on embedded and mobile devices are strongly influenced by memory size limitations. The authors use object compression to improve memory space utilization in an embedded Java environment. The compression is based on the observation that a small set of values appears frequently in heap-allocated objects.

  Their approach uses profile information to categorize the object fields into three levels: level-0 (the field does not have a dominant frequent value), level-1 (the field has a non-zero or non-null frequent value), level-2 (the field has a frequent value that is zero or null). They propose two compression schemes. The first one divides an object into primary part and secondary part (containing level-2 fields). The secondary part is eliminated if all the level-2 fields are zero or null. The second scheme shares level-1 fields among multiple objects. Experimental results showed that these compression schemes can reduce the heap size significantly with little performance impact.

GOING NATIVE

Summarized by Long Fei

- ***An Efficient and Generic Reversible Debugger using the Virtual Machine based Approach***

  Toshihiko Koju, Shingo Takada, and Norihisa Doi

  Toshihiko Koju began with the statement that reverse execution is very useful for locating the cause of software failures. He described a novel reversible debugger that uses a virtual machine based approach. This debugger provides compatibility and efficiency. In addition, it provides two execution modes: native mode, where the debugger is directly executed on a real CPU, and the virtual machine mode, where the debugger is executed on a virtual machine.

  In order to provide compatibility and efficiency, the debugger uses native machine code as its target. The virtual machine translates the native machine code of the target program by inserting code to save states that are changed during execution. The debugger is capable of switching between the native and the VM mode. In the native mode, users cannot reverse-execute the target program; in the VM mode, the users can use reverse execution. Some basic debugging functionality (e.g., breakpoint, step) is supported in both modes. The debugger allows four types of settings, to allow trade-offs between granularity, accuracy, overhead, and memory requirements of reverse execution. The user can choose the appropriate setting by designating the proper reverse execution unit (line or procedure) and optimization flag (enable or disable).

- ***Module-aware Translation for Real-life Desktop Applications***

  Jianhui Li, Peng Zhang, and Orna Etzion

  Jianhui Li explained that a dynamic binary translator is a just-in-time compiler that translates source architecture binaries into target architecture binaries on the fly. When hot modules are loaded and unloaded repeatedly, traditional dynamic translators spend a significant amount of time on repeatedly translating these modules. He proposed a translation reuse engine that uses a novel verification method and a module-aware memory management mechanism.

  There are three stages to accomplish translation reuse: translation reservation, source binary verification, and translation revivification. In this framework, when a translated code block is invalidated, it is

preserved by the reuse engine and saved by the execution engine. In order to verify that the saved translation is exactly the expected translation for a code block, the translation engine uses a save-and-comparison scheme. If the reuse engine decides to reuse the translation for a piece of the source binary, it saves a minimum set of source binaries that determine the semantics of translation. Before the translation engine translates a piece of the source binary, it requests the reuse engine to compare the saved partial source binaries with their counterparts in the current binary image. The saved translation is used if they are the same.

The authors propose a module-aware memory management mechanism, which organizes the translation code blocks of different modules into different pools (module-private page pool and general page pool). When a hot module is unloaded, its private code pages are reserved for future reuse (unless it's identified as not reusable). Experiments with real-life desktop applications show that this new translation-reuse technique can significantly improve the performance of four real-life desktop applications.

- ***Planning for Code Buffer Management in Distributed Virtual Execution Environments***

  Shukang Zhou, Bruce R. Childers, and Mary Lou Soffa

  Many devices in a distributed computing environment have tight memory constraints. One approach is to download code partitions on demand from a server and to cache the partitions in the client. Shukang Zhou and his colleagues addressed the problem of intelligently managing the code buffer to minimize the overhead of code buffer misses. They propose to move the code buffer management to the server, where sophisticated schemes can be employed.

  A program is first divided into code partitions, which are then stored in a code server connected to the client. Profiling is used to capture the hotness of code partitions. The client's code buffer is partitioned into multiple sub-buffers. The sub-buffers are ordered by the hotness of partitions assigned to them. One sub-buffer holds very hot code, while another may hold infrequently executed code. This approach is based on the fact that most programs spend a large part of their execution in a small portion of code.

  The authors discuss the overall strategy of CB memory planning and then describe two particular schemes. In the fixed scheme, code partitions are always housed in the same sub-buffer during execution. In the adaptive scheme, partitions are cached in sub-buffers based on a program's run-time behavior. The authors also introduce a heuristic called density, which is defined as a partition's execution frequency divided by its size, to measure the priority of code partitions to reside in CB. Experiments show that these schemes have fewer CB misses, which translates to a significant speedup.

KEYNOTE ADDRESS

Summarized by Shuo Yang, Xing Fang, and Long Fei

- ***Application Servers as Virtualization Environments***

  Martin Nally, CTO, IBM Rational

  Martin Nally first gave a broad overview of virtualization services. Virtualization techniques serve as tools of software development synergy between software and the execution environment. According to IDC data, the worldwide application server software platform revenue is increasing dramatically. He then introduced an example of writing Web server applications without knowing the specific target application servers, and showed the necessity for the Web server application to be compatible with different OS and hardware platforms.

  He introduced various approaches to building Web server applications, such as JVM (J2EE), CGI-BIN, etc., and discussed the pros and cons of each approach. Then he discussed different levels of virtualization—OS, JVM, and Web server—with respect to their generality. OS virtualization provides important and very general concepts to support application execution. JVM virtualization provides platform independence. Application server virtualization targets certain classes of applications.

  He said that application servers are usually thought of as containers and pointed out the important role of virtualization. First, the virtual environment serves as a logic wrapper. He gave a Web server application as an example of this view. He then discussed Web application containers as a virtual secure environment and talked about the scalability and performance challenges of application servers.

  He presented cluster and workload management and hardware virtualization. One of the key features of application servers is scaling. There are a wide variety of configurations to virtualize hardware. He introduced how Web cluster failover is handled and recovered. Web clusters allow transparent application updates and enable continuous availability of service. WebSphere XD is the next level of sophistication of virtualizing hardware. He discussed the challenges of an on-demand operating environment for a large financial company: for example, the underutilization of servers and the inability to share. He believes virtualization, such as WebSphere XD's automatic management, provides a better solution than the conventional approaches in this case.

  Nally concluded his talk with the following points. Application servers provide a virtual environ-

ment for executing Internet and intranet applications. Application servers present a simple virtual environment to application programmers. Application servers virtualize across many physically individual computers that may be running different OS application servers, even using some virtualization techniques that are not seen at the OS level.

He raised some questions at the end of his talk. J2EE applications run on JVMs which run on OSes, and each of these layers is performing virtualization—is this working? Could some of the redundancy be removed? Could these layers work better together or even be coalesced?

DYNAMIC COMPILATION TECHNIQUES

Summarized by Xing Fang

- ***Escape Analysis in the Context of Dynamic Compilation and Deoptimization***

  Thomas Kotzmann and Hanspeter Mössenböck

  Thomas Kotzmann presented an intra- and interprocedural escape analysis for a dynamic compiler. Escape analysis determines, for each object, whether it is accessible from within a single method, or one thread, or multiple threads. Method-local objects are eliminated and replaced with scalar variables. Thread-local objects are stack-allocated, and synchronization on them is removed.

  The analyses and optimizations are implemented in Sun's Java HotSpot Client VM, which has a front end that operates on an SSA-based High-level Intermediate Representation (HIR). Escape analysis and scalar replacement are performed in parallel with the construction of the HIR. A state object containing a locals array is maintained by the compiler, to track the values most recently assigned to local variables. The state object also has a fields array which stores the current values of all fields.

  An object is represented by its allocation instruction. The intraprocedural analysis parses instructions that might cause an object to escape, and updates the escape state of the instruction representing the object. Effects of various instructions on escape states were discussed. With SSA form, control flow is captured in phi-functions at the control merge points.

  Test results about compilation time and machine code quality were performed and analyzed. The benefit is very evident for some benchmarks, most notably mtrt in SPECjvm98 and Monte Carlo in SciMark.

- ***Inlining Java Native Calls at Runtime***

  Levon Stepanian, Angela Demke Brown, Allan Kielstra, Gita Koblents, and Kevin Stoodley

  Levon Stepanian started out by observing that Java native calls are pervasive because they allow legacy, high-performance, or architecture-dependent native code to be integrated with Java applications. However, cross-language calls usually incur large time and space overheads, and this is true with JNI.

  To reduce these overheads, the authors propose inlining JNI calls into Java applications with a JIT compiler. Both callouts (Java calls to functions implemented in external languages) and callbacks (external code accessing and modifying data and services from a running JVM) can be inlined. Work was done on the IBM TR JIT compiler, and the native functions exist in the form of W-code, the mature stack-based bytecode-like representation generated by IBM compiler front-ends.

  Entire native functions are inlined at their call sites. For small native functions, removing the overhead of callouts can be a significant benefit. The benefit of transforming callbacks is even higher, with a minimum achieved speedup of nearly 12X in the micro-benchmark test cases. Inlining also appears to reduce the need for conservative assumptions about the behavior of native code in the JIT optimizer.

- ***Optimized Interval Splitting in a Linear Scan Register Allocator***

  Christian Wimmer and Hanspeter Mössenböck

  Linear scan register allocation is very suitable for JIT compilers because it is much faster than graph coloring and is nearly as effective. For each virtual register, lifetime intervals store the range of instructions where a value is active. Two intersecting intervals must not have the same register assigned. The algorithm assigns registers to values in a single linear pass over all intervals.

  Three optimizations are proposed and implemented. Split positions are optimized to reduce the number of spill loads and stores at runtime. They are moved out of loops and into block boundaries. When two intervals are connected only by a move instruction, the interval of the move target stores the source of the move as its register hint. When possible, the target gets the same register allocated as the source, eliminating the register-to-register move. In two common cases that cover most of the intervals, moves inserted for spill stores can be removed.

  Christian Wimmer presented the flow of the algorithm and used detailed examples to illustrate it. The algorithm is implemented for Sun's Java HotSpot Client VM. Results prove the efficiency of the optimized algorithm: the compilation time of the algorithm is nearly linear, and it is even faster than the original register allocation algorithm in the client compiler.

LANGUAGE REPRESENTATIONS

Summarized by Xing Fang

- ***An Execution Layer for Aspect-Oriented Programming Languages***

  Michael Haupt, Mira Mezini, Christoph Bockisch, Tom Dinkelaker, Michael Eichberg, and Michael Krebs

  Michael Haupt introduced the key concepts of the pointcut-and-advice (PA) flavor of Aspect Oriented Programming (AOP): join points, pointcuts, and advice. A join point is a point in the execution of a program, a pointcut is a query that quantifies over join points, defining related sets of join points, and an advice is a piece of functionality that can be attached to pointcuts, taking semantic effect when the respective pointcuts match.

  According to Haupt, AOP language mechanisms, like OOP mechanisms, deserve implementation effort. AOP features have not gained sufficient support.

  Currently, dispatching logic is inserted into application logic at compile or load time. This gap in semantics confuses debug efforts and incurs performance drawbacks. The contribution of the work is the integration of both the JPRM and the WM into the VM for supporting AspectJ's dynamic point model. The authors developed Steamloom, an extension to IBM's Jikes RVM that provides AOP functionality at the VM level, assessable through a Java API.

  Evaluations show that the overhead incurred by the modification to implement Steamloom is practically zero, for hot runs. Class loading and method compilation overhead is about 7.8%. Other results show that using an AOP-enabled infrastructure does not in itself mean that execution is slowed down. Applied modifications of the original VM do not critically interfere with other subsystems. AOP-related functionality is more efficiently realizable at VM level.

- ***Virtual Machine Showdown: Stack Versus Registers***

  Yunhe Shi, David Gregg, Andrew Beatty, and M. Anton Ertl,

  A long-running question in the design of Virtual Machines is whether stack architecture or register architecture can be implemented more efficiently with an interpreter. David Gregg started off by noting that stack machines were more popular, because of the small code size and the ease of building stack machines. The JVMs and PERL 5 interpreter took this approach. But PERL 6 used a register machine instead, because register code was perceived as faster to interpret.

  Execution of a VM instruction could be broken down into instruction dispatch, operand access, and actual computation times. Register code incurs less dispatch time than stack code, because of its fewer number of instructions. However, register code needs to access its operands explicitly so the code size is usually larger, resulting in more memory fetches for the code. Actual computation time is about the same for register and stack codes. Instruction dispatch is more costly than code fetching, so register code has the potential to be faster.

  The authors built a much more sophisticated translator from the stack code to the register code. Results showed that the register code has 47% fewer instructions at a cost of a 25% increase in code size. Optimizations on the register code were very effective. 43.47% of static VM instructions were eliminated, as well as 47.21% of the dynamic VM code. On a Pentium 4, the register machine requires 32.3% less time than stack machine, with a less than perfect dispatch scheme. With a better dispatch, the reduction in execution time was still 26.5%. It is a very strong indication that the register architecture is superior to the stack architecture for implementing interpreter-based VMs.

- ***Instrumenting Annotated Programs***

  Marina Biberstein, Vugranam C. Sreedhar, Bilha Mendelson, Daniel Citron, and Alberto Giammaria

  Instrumentation is commonly used to collect program profile information. It is a spectative (i.e., one that observes the program behavior) program transformation and must maintain the program structure and functionality. Program annotation enables developers and tools to pass extra information to later stages of software development and execution. It is widely used in the CLR platform and has been adopted into the Java 1.5 standard.

  Marina Biberstein gave a motivating example of two annotations that were both interfered with, although differently, by instrumentation. To solve the problem, instrumentation must handle different annotations in different ways. There must be active cooperation between the two.

  The proposed solution takes an instrumentation-driven approach. Annotations are classified according to their behavior, and annotation writers would, for each annotation type, provide its description, in the form of meta-annotations. The descriptions provide information about the stage and lifetime of the annotation, its scope, sensitivity to instrumentation, and whether the annotation can be removed or healed. This information is passed on to instrumentation, which then bases its decisions on the information provided.

  The authors proposed a taxonomy of annotations based on their study of more than two hundred live examples, which they used to classify annotations. They demonstrated their solution on a set of sample annotations.

DISTRIBUTED VEES

Summarized by Shuo Yang

- ***PDS: A Virtual Execution Environment for Software Deployment***

  Bowen Alpern, Joshua Auerbach, Vasanth Bala, Thomas Frauenhofer, Todd Mummert, and Michael Pigott

  Joshua Auerbach presented a virtual machine solution, the Progressive Deployment System (PDS), to manage complex software deployment. The idea of PDS is to have software packaged, to provision, deploy, and execute software on customer machines, and to share software updates with many others (customers).

  Auerbach gave an overview of PDS by introducing a working prototype under Windows XP: assets to be delivered include Eclipse, JBoss, and WebSphere. Using this example, he presented the architecture components of PDS and showed that PDS's virtual environment makes software look locally installed and resolves environment conflicts.

  He talked about PDS's virtualizer. A process VM virtualizes the application binary interface (ABI), not hardware. Processes derived from the same asset are in the same VM. PDS uses a selective virtualization and only virtualizes OS calls that access the assets' virtually installed image. Auerbach concluded his talk by comparing the related work and presenting the differences with PDS and saying that PDS provides a feasible and convenient approach to deploying software packages.

- ***The Entropia Virtual Machine for Desktop Grids***

  Brad Calder, Andrew Chien, Ju Wang, and Don Yang

  Brad Calder introduced the background of desktop distributed computing. The customers in this venue require the following features: desktop security, a clean execution environment, unobtrusiveness, application security, ease of application integration, lightweight VM installation, and low performance overhead.

  He gave a system-architecture overview of the Entropia desktop computing system, which includes job management, resource management, and physical node management. Entropia virtual machines (EVMs) consist of: (1) a desktop controller to guarantee unobtrusiveness; and (2) a sandbox execution layer to provide security features. The sandbox layer takes two approaches to guarantee security: device driver mediation (with relatively high overhead) and binary interception (with low overhead). The sandboxing execution layer provides file virtualization (all files are accessed through a confined virtual file system located in an Entropia directory); file I/O throttling and automated file encryption; registry virtualization; GUI virtualization; network virtualization; and network I/O throttling.

  Finally, he discussed the performance of jobs running on EVM. The talk concluded with an interesting discussion of market and customer demands.

- ***HyperSpector: Virtual Distributed Monitoring Environments for Secure Intrusion Detection***

  Kenichi Kourai and Shigeru Chiba

  Kenichi Kourai presented a virtual distributed monitoring environment called HyperSpector, whose goal is to achieve secure intrusion detection in distributed computer systems.

  He introduced the distributed intrusion detection systems (DIDs) and threats against DIDs by describing the behaviors and actions of active attacks and passive attacks. He then talked about the traditional approach to solving these challenges to DIDs—isolated monitoring—which is secure but needs additional hardware and supports only network-based IDSes (NIDSes).

  He discussed the design of HyperSpector: it runs IDSes and server applications on separate VMs, and it builds a virtual network across the IDS VMs. HyperSpector provides three mechanisms: software port mirroring (packet capturing), inter-VM mounting (filesystem checking), and inter-VM mapping (process checking).

  HyperSpector has been implemented in their Persona operating system, which is based on the FreeBSD 4.9 kernel. Their experimental results showed the effectiveness of their HyperSpector system design in terms of both security and overhead.