Virtual-future-computer essay

Each of these virtual machines was sufficiently similar to the underlying physical machine to run existing software unmodified. At the time, general-purpose computing was the domain of large, expensive mainframe hardware, and users found that Vim's provided a compelling way to multiplex such a scarce resource among multiple applications. Thus, for a brief period, this technology flourished both in industry and in academic research. The sass and sass, however, brought modern multitasking operating systems and a simultaneous drop in hardware cost, which eroded the value of Vim's.

As mainframes gave way to minicomputers and then PC's, Vim's disappeared to the extent that computer architectures no longer provided the necessary hardware to implement them efficiently. By the late 1 sass, neither academics nor Industry practitioners viewed Vim's as much more than a historical curiosity. Fast forwarding to 2005, Vim's are again a hot topic in academia and industry: Venture capital firms are competing to fund startup companies touting their virtual-machine-based technologies. Intel. MAD, Sun Microsystems, and MM are developing fertilization strategies that target markets with revenues in the billions and growing.

In research labs and universities, searchers are developing approaches based on virtual machines to solve mobility, security, and manageability problems. What happened between the Vim's essential retirement and its current resurgence? In the sass, Stanford university researchers began to look at the potential of virtual machines to overcome difficulties that hardware and operating system limitations imposed: This time the problems stemmed from massively parallel processing (MAP) machines that were difficult to program and could not run existing operating systems.

With virtual machines, researchers found they could make these unwieldy architectures look sufficiently reject came the people and ideas that underpinned Ovenware Inc. (www. Ovenware. Com), the original supplier of Vim’s for commodity computing hardware. The implications of having a VIM for commodity platforms intrigued both researchers and entrepreneurs. WHY THE REVIVAL? Ironically, the capabilities of modern operating systems and the drop in hardware cost-? the very 0018-9162/05/$20. 0 2005 IEEE Computer Published by the IEEE Computer Society combination that had obviated the use of Vim’s during the sass-? began to cause problems that researchers thought Vim’s might solve. Less expensive hardware had De to a proliferation of machines, but these machines were often underused and incurred significant space and management overhead. And the increased functionality that had made operating systems more capable had also made them fragile and vulnerable. To reduce the effects of system crashes and breaking, system administrators again resorted to a computing model with one application running per machine.

This in turn increased hardware requirements, imposing significant cost and management overhead. Moving applications that once ran on many physical machines into virtual machines and consolidating those virtual machines onto Just a ewe physical platforms increased use efficiency and reduced space and management costs. Thus, the Vim’s ability to serve as a means of multiplexing hardware-? this time in the name of server consolidation and utility computing-? again led it to prominence.

Moving forward, a VIM will be less a vehicle for multitasking, as it was originally, and more a solution for security and reliability. In many ways Vim’s give operating systems developers another opportunity to develop functionality no longer practical in today’s complex and ossified operating systems, where innovation moves at a geologic pace. Functions like migration and security that have proved difficult to achieve in modern operating systems seem much better suited to implementation at the VIM layer.

In this context, Vim’s provide a backward-capability path for deploying innovative operating system solutions, while providing the ability to safely pull along the existing software base. App App Operating system Operating system Virtual machine monitor Hardware DECOUPLING HARDWARE AND SOFTWARE As Figure 1 shows, the VIM decouples the software from the hardware by forming a level of indirection between the software running in the virtual machine (layer above he VIM) and the hardware.

This level of indirection lets the VIM exert tremendous control over how guest operating systems (Suggests)-? operating systems running inside a virtual machine-? use hardware resources. A VIM provides a uniform view of underlying hardware, making machines from different vendors with different 1/0 subsystems look the same, which means that virtual machines can run on any available computer. Thus, instead of worrying about individual machines with tightly coupled hardware and software dependencies, administrators can view hardware simply as a pool of resources that can run arbitrary services on demand.

Because the VIM also offers complete encapsulation of a virtual machine's software state, the VIM layer can map and remap virtual machines to available hardware resources at will and even migrate virtual machines across machines. Load balancing among a collection of machines thus becomes trivial, and there is a robust model for dealing with hardware failures or for scaling systems. When a computer fails and must go offline or when a new machine comes online, the VIM layer can simply remap virtual machines accordingly.

Virtual machines are also easy to replicate, which lets administrators bring new services online as needed. Encapsulation also means that administrators can suspend virtual machines and resume them at arbitrary times or checkpoint them and roll them back to a previous execution state. With this general-purpose undo capability, systems can easily recover from crashes or configuration errors. Encapsulation also supports a very general mobility model, since users can copy a suspended virtual machine over a network or store and transport it on removable media.

The VIM can also provide total mediation of all interactions between the virtual machine and underlying hardware, thus allowing strong isolation between ritual machines and supporting the multiplexing of many virtual machines on a single hardware platform. The VIM can then consolidate a collection of virtual machines with low resources onto a single computer, thereby lowering hardware costs and space requirements. Strong isolation is also valuable for reliability and security. Applications that previously ran together on one machine can now separate into different virtual machines.

If one application crashes the operating system because of a bug, the other applications are isolated from this fault and can continue Figure 1 . Classic VIM. The VIM is a thin software layer that exports a virtual machine abstraction. The abstraction looks enough like the hardware that any software written for that hardware will run in the virtual machine. May 2005 35 The central design goals for Vim’s are compatibility, performance, and simplicity. Compromise a single application, the attack is contained to Just the compromised virtual machine.

Thus, Vim’s are a tool for restructuring systems to enhance robustness and security-? without imposing the space or management overhead that would be required if applications executed on separate physical machines. VIM IMPLEMENTATION ISSUES The VIM must be able to export a hardware interface to the software in a virtual machine that is roughly equivalent to raw hardware and simultaneously maintain control of the machine and retain the ability to interpose on hardware access. Various techniques can help achieve this, each offering different design tradeoffs.

When evaluating these tradeoffs, the central design goals for Vim’s are compatibility, performance, and simplicity. Compatibility is clearly important, since the Vim’s chief benefit is its ability to run legacy software. The goal of performance, a measure of retaliation overhead, is to run the virtual machine at the same speed as the software would run on the real machine. Simplicity is particularly important because a VIM failure is likely to cause all the virtual machines running on the computer to fail. In particular, providing secure isolation requires that the VIM be free of bugs that attackers could use to subvert the system.

CPU fertilization A CPU architecture is Brazzaville if it supports the basic VIM technique of direct execution-? executing the virtual machine on the real machine, while letting the VIM retain ultimate control of the CPA]. Implementing basic direct execution requires running the virtual machine’s privileged (operating-system kernel) and unprivileged code in the Cups unprivileged mode, while the VIM runs in privileged mode. Thus, when the virtual machine attempts to perform a privileged operation, the CPU traps into the VIM, which emulates the privileged operation on the virtual machine state that the VIM manages.

The VIM handling of an instruction that disables interrupts provides a good example. Letting a guest operating system disable interrupts would not be safe since the VIM could not regain control of the CPA]. Instead, the VIM old trap the operation to disable interrupts and then record that interrupts were disabled for that virtual machine. The VIM would then postpone delivering subsets the key to providing Brazzaville architecture is to provide trap semantics that let a VIM safely, transparently, and directly use the CPU to execute the virtual machine.

With these semantics, the VIM can use direct execution to create the illusion of a normal physical machine for the software running inside the virtual machine. Challenges. Unfortunately, most modern CPU architectures were not designed to be Brazzaville, including the popular ex. architecture. For example, ex. operating systems use the ex. POP instruction (pop CPU flags from stack) to set and clear the interrupt-disable flag. When it runs in unprivileged mode, POP does not trap.

Instead, it simply ignores the changes to the interrupt flag, so direct execution techniques will not work for privileged-mode code that uses this instruction. Another challenge of the ex. architecture is that unprivileged instructions let the CPU access privileged state. Software running in the virtual machine can read the code segment register to determine the processor’s current privilege level. A Brazzaville processor old trap this instruction, and the VIM could then patch what the software running in the virtual machine sees to reflect the virtual machine’s privilege level.

The ex., however, doesn’t trap the instruction, so with direct execution, the software would see the wrong privilege level in the code segment register. Techniques. Several techniques address how to implement Vim’s on Cups that can’t be brutalized, the most prevalent being parameterizations and direct execution combined with fast binary translation. With personalization, the VIM builder defines the virtual machine interface by replacing invaluableness portions of the original instruction set with easily brutalized and more efficient equivalents.

Although operating systems must be ported to run in a virtual machine, most normal applications run unmodified. Disco, 3 a VIM for the invaluableness MIPS architecture, used personalization. Disco designers changed the MIPS interrupt flag to be simply a special memory location in the virtual machine rather than a privileged register in the processor. They replaced the MIPS equivalent of the ex. POP instruction and the read access to the code segment register with accesses to this special memory location.

This replacement also eliminated fertilization overhead such as traps on privileged instructions, which resulted in increased performance. The designers then modified a version of the Iris operating system to take advantage of this parallelized version of the MIPS architecture. The biggest drawback to personalization is incompatibility. Any operating system run in a parallelized VIM must be ported to that architecture. Operating system vendors must cooperate, legacy operating systems cannot run, and existing machines cannot easily migrate into virtual machines.

With years of excellent backward-compatible ex. hardware, huge amounts of legacy software are still in use, which means that giving up backward compatibility is not trivial. In spite of these drawbacks, academic research projects have favored personalization because building a VIM that offers full compatibility and high performance is a significant engineering challenge. To provide fast, compatible fertilization of the ex. architecture, Ovenware developed a new fertilization technique that combines operating systems, the processor modes that run normal application programs are Brazzaville and hence can run using direct execution.

A binary translator can run privileged modes that are invaluableness, patching the invaluableness ex. instructions. The result is a high-performance virtual machine that matches the hardware and thus maintains total software compatibility. Others have developed binary translators that translate code between Cups with different instruction sets. Ovenware’s binary translation is much simpler because the source and target instruction sets are nearly identical. The Vim’s basic technique is to run privileged mode code (kernel code) under control of the binary translator.

The translator translates the privileged code into a similar block, replacing the problematic instructions, which lets the translated block run directly on the CPA]. The binary translation system caches the translated block in a trace cache so that translation does not occur on subsequent executions. The translated code looks much like the results from the parallelized approach: Normal instructions execute unchanged, while the translator replaces instructions that need special treatment, like POP and reads from the code segment registers with an instruction sequence similar to what a parallelized virtual machine would need to run.

There is one important difference, however: Rather than applying the changes to the source code of the operating system or applications, the binary translator applies the changes when the code first executes. While binary translation does incur some over- head, it is negligible on most workloads. The translator runs only a fraction of the code, Building a VIM and execution speeds are nearly indistinct offers full guessable from direct execution once the trace cache has warmed up. Compatibility and Binary translation is also a way to optimize high performance direct execution.

For example, privileged code is a significant that frequently traps can incur significant administering action overhead when using direct execution since each trap ranchers control from the overcharging. Dual machine to the monitor and back. Binary translation can eliminate many of these traps, which results in a lower overall fertilization overhead. This is particularly true on Cups with deep instruction pipelines, such as the modern ex. JPL’s, where traps incur high overhead. Future support.

In the near term, both Intel with its Ponderosa technology and MAD with its Pacific technology have announced hardware support for ex. CPU Vim’s. Rather than making existing execution modes Brazzaville, both the Intel and MAD technologies add a new execution mode to the processor that lets a VIM safely and rampantly use direct execution for running virtual machines. To improve performance, the mode attempts to reduce both the traps needed to implement virtual machines and the time it takes to perform the traps.

When these technologies become available, directionality-only Vim’s could be possible on ex. processors, at least for operating system environments that do not use these new execution modes. If this hardware support works as well as the IBM mainframe fertilization support of the early days, it should be possible to decrease performance overhead even more, s well as simplifying the implementation of fertilization techniques. Lessons from the past indicate that adequate hardware support can decrease overhead, even virtual machine abstraction overrides any performance benefits from breaking compatibility.

Memory fertilization The traditional implementation technique for brutalizing memory is to have the VIM maintain a shadow of the virtual machine’s memory-management data structure. This data structure, the shadow page table, lets the VIM precisely control which pages of the machine’s memory are available to a virtual machine. When the operating system running in a virtual machine establishes a mapping in its page table, the May 2005 37 VIM detects the changes and establishes a mapping in the corresponding shadow page Resource table entry that points to the actual page electromagnets Zion in the hardware memory.

When the oversold great Dual machine is executing, the hardware uses promise as an the shadow page table for memory translation so that the VIM can always control area for future what memory each virtual machine is using. Research. Like a traditional operating system's virtual memory subsystems, the VIM an page the virtual machine to a disk so that the memory allocated to virtual machines can exceed the hardware's physical memory size. Because this effectively lets the VIM overcoming the machine memory, the virtual machine workload requires less hardware.

The VIM can dynamically control how much memory each virtual machine gets according to what it needs. Challenges. The Vim's virtual memory subsystem constantly controls how much memory goes to a virtual machine, and it must periodically reclaim some of that memory by paging a portion of the virtual machine out to disk. The operating system running in the virtual machine (the Guests), however, is likely to have much better information than a Vim's virtual memory system about which pages are good candidates for paging out.

For example, a Guests might note that the process that created a page has exited, which means nothing will access the page again. The VIM operating at the hardware level does not see this and might wastefully page out that page. To address this problem, Ovenware's SEX Servers adopted a personalization-like approach, in which a balloon process running inside the Guests can communicate with the VIM. When the VIM ants to take memory away from a virtual machine, it asks the balloon process to allocate more memory, essentially “ inflating” the process.

The Guests then uses its superior knowledge about page replacement to select the pages to give to the balloon process, which the process then passes to the VIM for reallocation. The increased memory pressure caused by inflating the balloon process causes the Guests to intelligently page memory to the virtual disk. A second challenge for memory fertilization is the size of modern operating systems and applications. Running multiple virtual machines can waste considerable memory by storing attendant copies of code and data that are identical across virtual machines.

To address this challenge, Ovenware designers developed content-based page sharing for server products. In this scheme, the VIM tracks the contents of physical pages, noting if they are identical. If so, the VIM modifies the virtual machine’s shadow page tables to point to only a single copy. The VIM can then delectate the redundant copy, thereby freeing the memory for other uses. As with a normal copy- on-write page-sharing scheme, the VIM gives each virtual machine its own copy of the page if the contents later diverge.

To give an idea of potential savings, an ex. computer might have 30 virtual machines running Microsoft Windows 2000 but only one copy of the Windows kernel in the computer’s memory-? a significant reduction in physical memory use. Future support. Operating systems make frequent changes to their page tables, so keeping shadow copies up to date in software can incur undesirable overhead. Hardware-managed shadow page tables have long been present in mainframe fertilization architectures and would prove a fruitful direction for accelerating ex. CPU fertilization.

Resource management holds great promise as an area for future research. Much work remains in investigating ways for Vim’s and guest operating systems to make cooperative resource management decisions. In addition, research must look at resource management at the entire data center level, and we expect significant strides will be made in this area in the coming decade. 1/0 fertilization Thirty years ago, the 1/0 subsystems of IBM mainframes used a channel-based architecture, in which access to the 1/0 devices was through communication with a separate channel processor.

By using a channel processor, the VIM could safely export 1/0 device access directly to the virtual machine. The result was a very low fertilization overhead for 1/0. Rather than communicating with the device using traps into the VIM, the software in the virtual machine could directly read and write the device. This approach worked well for the 1/0 devices of that time, such as text terminals, disks, card readers, and card punches. Challenges. Current computing environments, with their richer and more diverse collection of 1/0 devices, make brutalizing 1/0 much more difficult.

The ex.-based computing environments support a huge collection of 1/0 devices from different vendors with different programming interfaces. Consequently, the Job of writing a VIM layer that talks to these various devices becomes a huge effort. In addition, some devices such as a modern PC’s graphics subsystem or a modern server’s network interface have extremely high performance requirements. This makes low-overhead fertilization an even more critical prerequisite for widespread acceptance. Exporting a standard device interface means that the fertilization layer must be able to communicate with the computer’s 1/0 devices.

To provide this capability, Ovenware Workstation, a product targeting desktop computers, developed the hosted architectures shown in Figure 2. In this architecture, the fertilization layer uses the device drivers of a host operating system (Hosts) such as Windows or Linux to access devices. Because most 1/0 devices have drivers for these operating systems, command to read or write blocks from the virtual disk, the virtual layer translates the command into a system call that reads or writes a file in the Hostess's file system.

Similarly, the 1/0 VIM renders the virtual machine's virtual display card in a window on the Hosts, which lets the Hosts control, drive, and manage the virtual machine's 1/0 display devices regardless of what devices the Guests thinks are resent. The hosted architecture has three important advantages. First, the VIM is simple to install because users can install it like an application on the Hosts rather than on the raw hardware, as with traditional Vim's. Second, the hosted architecture fully accommodates the rich diversity of 1/0 devices in the ex. PC marketplace.

Third, the VIM can use the scheduling, resource management, and other services the Hosts environment offers. The disadvantages of the hosted architecture became material when Ovenware started to develop products for the ex. server marketplace. The hosted architecture greatly increases the performance overhead for 1/0 device fertilization. Each 1/0 request must transfer control to the Hosts environment and then transition through the Hostess's software layers to talk to the 1/0 devices. For server environments with high-performance network and disk subsystems, the resulting overhead was unacceptably high.

Another problem is that modern operating systems such as Windows and Linux do not have the resource- management support to provide performance isolation and service guarantees to the virtual machines-? a feature that many server environments require. SEX Servers adopts a more traditional VIM approach, running directly on the hardware without a host operating system. In addition to sophisticated scheduling and resource management, SEX App App App 1/0 VIM Guests VIM Hosts Standard ex. PC hardware Figure 2. Ovenware's hosted architecture.

Rather than running as a layer below all other software, the hosted architecture shares the hardware with an existing operating system (Hosts). Server has a highly optimized 1/0 subsystem for network and storage devices. The SEX Server kernel can use device drivers from the Linux kernel to talk directly to the vice, resulting in significantly lower fertilization overhead for 1/0 devices. Ovenware could use this approach because relatively few network and storage 1/0 devices have passed certification to run in major ex. vendor server machines. Limiting support to these 1/0 devices makes directly managing the 1/0 devices feasible for servers.

Yet another performance optimization in Ovenware's products is the ability to export special highly optimized virtual 1/0 devices that don't correspond to any existing 1/0 requires that Guests environments use a special device driver to access the 1/0 devices. The result is a more fertilization-friendly 1/0 device interface with lower overhead for communicating the 1/0 commands from the Guests and thus higher performance. Future support. Like CPU trends, industry trends in 1/0 subsystems point toward hardware support for high-performance 1/0 device fertilization.

Discrete 1/0 devices, such as the standard ex. PC keyboard controller and DID disk controllers that date back to the original IBM PC, are giving way to channel-like 1/0 devices, such as USB and CICS. Like the IBM mainframe 1/0 channels, these 1/0 interfaces greatly ease implementation complexity and reduce fertilization overhead. With adequate hardware support, safely passing these channel 1/0 devices directly to the software in the virtual machine should be possible, effectively eliminating all 1/0 fertilization overhead.

For this to work, 1/0 devices will need to know about virtual machines and be able to support multiple virtual interfaces so that the VIM can safely map the interface into the virtual machine. In this way, the virtual machine's device drivers will be able to comma 2005 39 enunciate directly with the 1/0 device without the overhead of trapping into the VIM. Virtual machines 1/0 devices that perform direct memory provide a powerful access ill require address remapping.

The unifying paradigm remapping ensures that the memory for restructuring addresses that the device driver running in the virtual machine specifies will get mapped to desktop the locations in the computer's memory that management. The shadow page tables specify. For the isolation property to hold, the device should be able to access only memory belonging to the virtual machine regardless of how the driver in the virtual machine programs the device. In a system with multiple virtual machines using the same 1/0 device, the VIM will need an efficient mechanism for routing device completion interrupts to the correct virtual machine.

Finally, Brazzaville 1/0 devices will need to interface to the VIM to maintain isolation between hardware and software and ensure that the VIM can continue to migrate and take a checkpoint of the virtual machines. 1/0 devices that provide this kind of support could minimize fertilization overhead, allowing the use of virtual machines for even the most 1/0-intensive workloads. Besides performance, a significant benefit is the improved security and reliability gained from removing complex device driver code from the VIM. Teens physical machines according to the data center’s needs.

The VIM can handle traditional hardware-management problems, such as hardware failure, simply by placing the virtual machines running on the failed computer onto other correctly functioning hardware. The ability to move running virtual machines also eases some hardware challenges, such as scheduling preventive maintenance, dealing with equipment lease ends, and deploying hardware upgrades. Administrators can use hot migration to perform these tasks without service interruptions.