Proceedings of the Linux Symposium

Volume Two

July 20nd–23th, 2005
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Stephanie Donovan, *Linux Symposium*

Review Committee

Gerrit Huizenga, *IBM*
Matthew Wilcox, *HP*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Matt Domsch, *Dell*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

LINUX® Virtualization on Virtual Iron™ VFe

Alex Vasilevsky, David Lively, Steve Ofsthun
Virtual Iron Software, Inc.
{alex,dflively,sofsthun}@virtualiron.com

Abstract

After years of research, the goal of seamlessly migrating applications from shared memory multi-processors to a cluster-based computing environment continues to be a challenge. The main barrier to adoption of cluster-based computing has been the need to make applications cluster-aware. In trying to solve this problem two approaches have emerged. One consists of the use of middleware tools such as MPI, Globus and others. These are used to rework applications to run on a cluster. Another approach is to form a pseudo single system image environment by clustering multiple operating system kernels[Pfister-98]. Examples of these are Locus, Tandem NonStop Kernel, OpenSSI, and openMosix.

However, both approaches fall far short of their mark. Middleware level clustering tools require applications to be reworked to run a cluster. Due to this, only a handful of highly specialized applications sometimes referred to as `embarrassingly parallel`—have been made cluster-aware. Of the very few commercial cluster-aware applications, the best known is Oracle® Database Real Application Clustering. OS kernel clustering approaches present other difficulties. These arise from the sheer complexity of supporting a consistent, single system image to be seen on every system call made by every program running on the system: applications, tools, etc; to making existing applications that use SystemV shared-memory constructs to run transparently, without any modifications, on this pseudo single system.

In 2003, Virtual Iron Software began to investigate the potential of applying virtual machine monitors (VMM) to overcome difficulties in programming and using tightly-coupled clusters of servers. The VMM, pioneered by IBM in the 1960s, is a software-abstraction layer that partitions hardware into one or more virtual machines[Goldberg-74], and shares the underlying physical resource among multiple applications and operating systems.

The result of our efforts is Virtual Iron VFe, a purpose-built clustered virtual machine monitor technology, which makes it possible to transparently run any application, without modification, on a tightly-coupled cluster of computers. The Virtual Iron VFe software elegantly abstracts the underlying cluster of computers with a set of Clustered Virtual Machine Monitors (CVMM). Like other virtual machine monitors, the CVMM layer takes complete control of the underlying hardware and creates virtual machines, which behave like independent physical machines running their own operating systems in isolation. In contrast to other virtual machine monitors, the VFe software transparently creates a shared memory multi-processor out of a collection of tightly-coupled servers.

Within this system, each operating system has

the illusion of running on a single multi-processor machine with `N` CPUs on top of `M` physical servers interconnected by high throughput, low latency networks.

Using a cluster of VMMs as the abstraction layer greatly simplifies the utilization and programmability of distributed resources. We found that the VFe software can run any application without modification. Moreover, the software supports demanding workloads that require dynamic scaling, accomplishing this in a manner that is completely transparent to OSs and their applications.

In this paper we'll describe Linux virtualization on Virtual Iron VFe, the virtualization capabilities of the Virtual Iron Clustered VMM technology, as well as the changes made to the LINUX kernel to take advantage of this new virtualization technology.

1 Introduction

The CVMM creates virtual shared memory multi-processor servers (Virtual Servers) from networks of tightly-coupled independent physical servers (Nodes). Each of these virtual servers presents an architecture (the Virtual Iron Machine Architecture, or ViMA) that shares the user mode instruction set with the underlying hardware architecture, but replaces various kernel mode mechanisms with calls to the CVMM, necessitating a port of the guest operating system (aka guest OS) kernel intended to run on the virtual multi-processor.

The Virtual Iron Machine Architecture extends existing hardware architectures, virtualizing access to various low-level processor, memory and I/O resources. The software incorporates a type of Hybrid Virtual Machine Monitor [Robin-00], executing non-privileged instructions (a subset of the hardware platform's Instruction Set Architecture) natively in hardware, but replacing the ISA's privileged instructions with a set of (sys)calls that provide the missing functionality on the virtual server. Because the virtual server does not support the full hardware ISA, it's not a virtual instance of the underlying hardware architecture, but rather a virtual instance of the Virtual Iron Machine Architecture (aka Virtual Hardware), having the following crucial properties:

- The virtual hardware acts like a multi-processor with shared memory.
- Applications can run natively "as is," transparently using resources (memory, CPU and I/O) from all physical servers comprising the virtual multi-processor as needed.
- Virtual servers are isolated from one another, even when sharing underlying hardware. At a minimum, this means a software failure in one virtual server does *not* affect[1] the operation of other virtual servers. We also prevent one virtual server from seeing the internal state (including deallocated memory contents) of another. This property is preserved even in the presence of a maliciously exploitive (or randomly corrupted) OS kernel.

Guaranteeing the last two properties simultaneously requires a hardware platform with the following key architectural features[Goldberg-72]:

- At least two modes of operation (aka privilege levels, or rings) (but three is better for performance reasons)

[1] Unreasonably, that is. Some performance degradation can be expected for virtual servers sharing a CPU, for example. But there should be no way for a misbehaving virtual server to starve other virtual servers of a shared resource.

- A method for non-privileged programs to call privileged system routines
- A memory relocation or protection mechanism
- Asynchronous interrupts to allow the I/O system to communicate with the CPU

Like most modern processor architectures, the Intel IA-32 architecture has all of these features. Only the Virtual Iron CVMM is allowed to run in kernel mode (privilege level 0) on the real hardware. Virtual server isolation implies the guest OS cannot have uncontrolled access to any hardware features (such as the CPU control registers) nor to certain low-level data structures (such as the paging directories/tables and interrupt vectors).

Since the IA-32 has four privilege levels, the guest OS kernel can run at a level more highly privileged than user mode (privilege level 3), though it may not run in kernel mode (privilege level 0, reserved for the CVMM). So the LINUX kernel runs in supervisor mode (privilege level 1) in order to take advantage of the IA-32's memory protection hardware to keep applications from accessing pages meant only for the kernel.

2 System Design

In the next few sections we describe the basic design of our system. First, we mention the features of the virtualization that our CVMM provides. Next, we introduce the architecture of our system and how virtual resources are mapped to physical resources. And lastly we describe the LINUX port to this new virtual machine architecture.

2.1 Virtual Machine Features

The CVMM creates an illusion of a shared memory virtual multi-processor. Key features of our virtualization are summarized below:

- The CVMM supports an Intel® ISA architecture of modern Intel processors (such as Intel XEON™).
- Individual VMMs within the CVMM are not implemented as a traditional virtual machine monitor, where a complete processor ISA is exposed to the guest operating system; instead a set of data structures and APIs abstract the underlying physical resources and expose a "virtual processor" architecture with a conceptual ISA to the guest operating system. The instruction set used by a guest OS is similar, but not identical to that of the underlying hardware. This results in a greatly improved performance, however it does require modifications to the guest operating system. This approach to processor virtualization is known in the industry as hybrid virtualization or as paravirtualization[Whitaker-00].
- The CVMM supports multiple virtual machines running concurrently in complete isolation. In the Virtual Iron architecture, the CVMM provides a distributed hardware sharing layer via the virtual multiprocessor machine. This virtual multiprocessor machine provides access to the basic I/O, memory and processor abstractions. A request to access or manipulate these items is handled via the ViMA APIs presented by the CVMM.
- Being a clustered system Virtual Iron VFe provides *dynamic resource management*, such as node eviction or addition visible

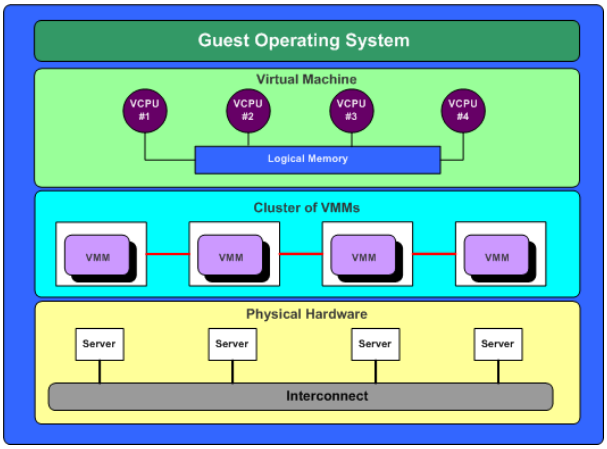

Figure 1: A Cluster of VMMs supporting a four processor VM.

to the Virtual Machine as hot-plug processor(s), memory and device removal or addition respectively.

We currently support LINUX as the guest OS; however the underlying architecture of the Virtual Iron VFe is applicable to other operating systems.

2.2 Architecture

This section outlines the architecture of Virtual Iron VFe systems. We start with differing views of the system to introduce and reinforce the basic concepts and building blocks. The Virtual Iron VFe system is an aggregation of component systems that provide scalable capabilities as well as unified system management and reconfiguration. Virtual Iron VFe software creates a shared memory multi-processor system out of component systems which then runs a single OS image.

2.2.1 Building Blocks

Each system is comprised of elementary building blocks: processors, memory, interconnect, high-speed I/O and software. The basic hardware components providing processors and memory are called nodes. Nodes are likely to be packages of several components such as processors, memory, and I/O controllers. All I/O in the system is performed over interconnect fabric, fibre channel, or networking controllers. All elements of the system present a shared memory multiprocessor to the end applications. This means a unified view of memory, processors and I/O. This level of abstraction is provided by the CVMM managing the processors, memory and the interconnect fabric.

2.2.2 Server View

Starting from the top, there is a virtual server running guest operating system, such as RHAS, SUSE, etc. The guest operating system presents the expected multi-threaded, POSIX server instance running multiple processes and threads. Each of these threads utilizes resources such as processor time, memory and I/O. The virtual server is configured as a shared memory multi-processor. This results in a number of processors on which the guest operating system may schedule processes and threads. There is a unified view of devices, memory, file systems, buffer caches and other operating system items and abstractions.

2.2.3 System View

The Building Blocks View differs from the previously discussed Server View in significant ways. One is a collection of unshared components. The other is a more typical, unified, shared memory multi-processor system. This needs to be reconciled. The approach that we use is to have the CVMM that presents Virtual Processors (VPs) with unified logical memory to the guest OS, and maps these VPs onto the

physical processors and logical memory onto distributed physical memory. A large portion of the instruction set is executed by the machine's physical processor without CVMM intervention, the resource control and management is done via ViMA API calls into the CVMM. This is sufficient to create a new machine model/architecture upon which we run the virtual server. A virtual server is a collection of virtual processors, memory and virtual I/O devices. The guest OS runs on the virtual server and the CVMM manages the mapping of VPs onto the physical processor set, which can change as the CVMM modifies the available resources.

Nodes are bound into sets known as Virtual Computers (VC). Each virtual computer must contain at least one node. The virtual computers are dynamic in that resources may join and leave a virtual computer without any system interruption. Over a longer time frame, virtual computers may be created, destroyed and reconfigured as needed. Each virtual computer may support multiple virtual servers, each running a single instance of an operating system. There are several restrictions on virtual servers, virtual computers, and nodes. Each virtual server runs on a single virtual computer, and may not cross virtual computers. An individual node is mapped into only a single virtual computer.

The virtual server guest operating system, LINUX for instance, is ported to run on a new virtual hardware architecture (more details on this further in the document). This new virtual hardware architecture is presented by the CVMM. From the operating system point of view, it is running on a shared memory multi-processor system. The virtual hardware still performs the computational jobs that it always has, including context switching between threads.

In summary, the guest operating system runs on a shared memory multi-processor system of new design. The hardware is managed by the CVMM that maps physical resources to virtual resources.

3 Implementation of the CVMM

In this section we describe how the CVMM virtualizes processors, memory and I/O devices.

3.1 Cluster of VMMs (CVMM)

The CVMM is the software that handles all of the mapping of resources from the physical to virtual. Each node within a CVMM cluster runs an instance of the VMM, and these instances form a shared-resource cluster that provides the services and architecture to support the virtual computers and appear as a single shared memory multi-processor system. The resources managed by the CVMM include:

- Nodes
- Processors
- Memory, local and remote
- I/O (devices, buses, interconnects, etc)
- Interrupts, Exceptions and Traps
- Inter-node communication

Each collection of communicating and cooperating VMMs forms a virtual computer. There is a one-to-one mapping of virtual computer to the cluster of VMMs. The CVMM is re-entrant and responsible for the scheduling and management of all physical resources. It is as thin a layer as possible, with a small budget for the overhead as compared to a bare LINUX system.

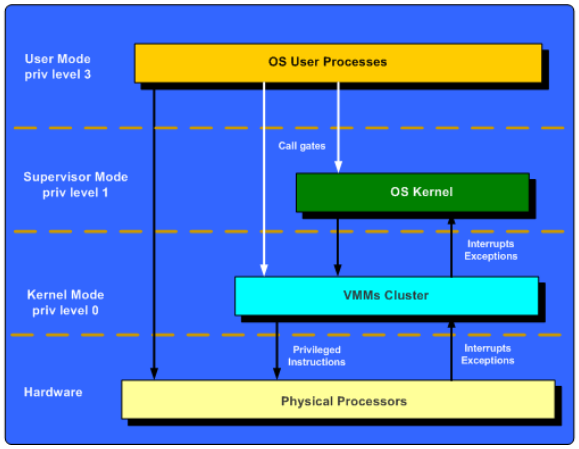

Figure 2: Virtual Iron paravirtualization (Intel IA-32).

3.2 Virtualizing Processors

Each of the physical processors is directly managed by the CVMM and only by the CVMM. A physical processor is assigned to a single virtual computer, and may be used for any of the virtual servers that run on that virtual computer. As we stated before, the method of virtualizing the processors that is used in the Virtual Iron VFe is *paravirtualization*. The diagram in *Figure 2* illustrates our implementation of the paravirtualization concept on the IA-32 platform:

In this scheme, the vast majority of the virtual processor's instructions are executed by the real processor without any intervention from the CVMM, and certain privileged instructions used by the guest OS are rewritten to use the ViMA APIs. As with other VMMs, we take advantage of underlying memory protection in the Intel architecture. The CVMM runs in the privilege ring-0, the guest OS runs in ring-1 and the user applications run in ring-3. The CVMM is the only entity that runs in ring-0 of the processor and it is responsible for managing all operations of the processor, such as booting, initialization, memory, exceptions, and so forth.

Virtual processors are mapped to physical processors by the CVMM. There are a number of rules that are followed in performing this mapping:

- Virtual processors are scheduled concurrently, and there are never more virtual processors than physical processors within a single virtual server.
- The CVMM maintains the mapping of virtual processors to physical processors.
- Physical processors may belong to a virtual computer, but are not required to be used or be active for any particular virtual server.

These rules lead to a number of conclusions. First, any number of physical processors may be assigned to a virtual computer. However, at any given moment, the number of active physical processors is the same as the number of virtual processors in the current virtual server. Moreover, the number of active virtual processors in a virtual server is less than or equal to the number of physical processors available. For instance, if a node is removed from a virtual computer, it may be necessary for a virtual server on that virtual computer to reduce the number of active virtual processors.

3.3 Interrupts, Traps and Exceptions

The CVMM is set-up to handle all interrupts, traps and exceptions. Synchronous traps and exceptions are mapped directly back into the running virtual processor. Asynchronous events, such as interrupts, have additional logic such that they can be mapped back to the appropriate virtual server and virtual processor.

3.4 Virtualizing Memory

Memory, like processors, is a resource that is shared across a virtual computer and used by the virtual servers. Shared memory implemented within a distributed system naturally results in non-uniform memory access times. The CVMM is responsible for memory management, initialization, allocation and sharing. Virtual servers are not allowed direct access to the page tables. However, these tables need to be managed to accommodate a number of goals:

- First, they need to be able to specifically locate an individual page which may reside in any one of the physical nodes
- They must be able to allow several levels of cost. That is, the virtual server should be able to manipulate page tables at the lowest possible cost in most instances to avoid round-trips through the CVMM.
- Isolation is a requirement. No virtual server should be able to affect any other virtual server. If several virtual servers are running on the same virtual computer, then any failures, either deliberate or accidental, should not impact the other virtual servers.

The illusion of a shared memory multiprocessor system is maintained in the Virtual Iron architecture by the sharing of the memory resident in all of the nodes in a virtual computer. As various processors need access to pages of memory, that memory needs to be resident and available for local access. As processes, or the kernel, require access to pages, the CVMM is responsible for insuring the relevant pages are accessible. This may involve moving pages or differences between the modified pages.

3.5 Virtualizing I/O Devices

Just as with the other resources, the CVMM manages all I/O devices. No direct access by a virtual server is allowed to any I/O device, control register or interrupt. The ViMA provides APIs to access the I/O devices. The virtual server uses these APIs to access and control all I/O devices.

All I/O for the virtual computer is done via interfaces and mechanisms that can be shared across all the nodes. This requires a set of drivers within the CVMM that accommodate this, as well as a proper abstraction at the level of a virtual server to access the Fibre Channel and Ethernet.

With the previous comments in mind, the job of the CVMM is to present a virtualized I/O interface between the virtual computer physical I/O devices and the virtual servers. This interface provides for both sharing and isolation between virtual servers. It follows the same style and paradigm of the other resources managed by the CVMM.

4 LINUX Kernel Port

This section describes our port of the LINUX 2.6 kernel to the IA-32 implementation of the Virtual Iron Machine Architecture. It doesn't specify the architecture in detail, but rather describes our general approach and important pitfalls and optimizations, many of which can apply to other architectures real and virtual. First we'll look at the essential porting work required to make the kernel run correctly, then at the more substantial work to make it run well, and finally at the (substantial) work required to support dynamic reconfiguration changes.

4.1 Basic Port

We started with an early 2.6 LINUX kernel, deriving our architecture port from the i386 code base. The current release as of this writing is based on the 2.6.9 LINUX kernel. As the burden of maintaining a derived architecture is substantial, we are naturally interested in cooperating with various recent efforts to refactor and generalize support for derived (e.g., x86_64) and virtualized (e.g., Xen) architectures.

The ViMA interface is mostly implemented via soft interrupts (like syscalls), though memory-mapped interfaces are used in some special cases where performance is crucial. The data structures used to communicate with the CVMM (e.g., descriptor tables, page tables) are close, if not identical, to their IA-32 equivalents.

The basic port required only a single modification to common code, to allow architectures to override *alloc_pgd()* and *free_pgd()*. Though as we'll see later, optimizing performance and adding more dynamic reconfiguration support required more common code modifications.

The virtual server configuration is always available to the LINUX kernel. As mentioned earlier, it exposes the topology of the underlying hardware: a cluster of *nodes*, each providing memory and (optionally) CPUs. The configuration also describes the virtual devices available to the server. Reading virtual server configuration replaces the usual boot-time BIOS and ACPI table parsing and PCI bus scanning.

4.2 Memory Management

The following terms are used throughout this section to describe interactions between the LINUX guest operating system and the Virtual Iron CVMM.

Page Type	Meaning
Physical page	A local memory instance (copy) of a ViMA logical page. The page contents are of interest to the owning virtual server.
Physical page frame	A local memory container, denoted by a specific physical address, managed by the CVMM.
Logical page	A virtual server page, the contents of which are managed by the guest operating system. *The physical location of a logical page is not fixed, nor even exposed to the guest operating system.*
Logical page frame	A logical memory container, denoted by a specific logical address, managed by the guest operating system.
Replicated page	A logical page may be replicated on multiple nodes as long as the contents are quaranteed to be identical. Writing to a replicated logical page will invalidate all other copies of the page.

Table 1: Linux Memory Management in Virtual Iron VFe

Isolating a virtual server from the CVMM and other virtual servers sharing the same hardware requires that memory management be carefully controlled by the CVMM. Virtual servers cannot see or modify each other's memory under any circumstances. Even the contents of freed or "borrowed" physical pages are never visible to any other virtual server.

Accomplishing this isolation requires explicit mechanisms within the CVMM. For example, CPU control register cr3 points to the top-level page directory used by the CPU's paging unit. A malicious guest OS kernel could try to point this to a `fake` page directory structure mapping pages belonging to other virtual servers into its own virtual address space. To prevent this, ***only*** *the CVMM can create and modify the page directories / tables used by the hardware, and it must ensure that cr3 is set only to a top-level page directory that it created for the appropriate virtual server.*

On the other hand, the performance of memory management is also of crucial importance. Taking a performance hit on every memory access is not acceptable; the *common case* (in which the desired logical page is in local memory, mapped, and accessible) *suffers no virtualization overhead.*

The MMU interface under ViMA 32-bit architecture is mostly the same as that of the IA-32 in PAE mode, with three-level page tables of 64-bit entries. A few differences exist, mostly that ours map 32-bit virtual addresses to 40-bit logical addresses, and that we use software dirty and access bits since these aren't set by the CVMM.

The page tables themselves live in logical memory, which can be distributed around the system. To reduce possibly-remote page table accesses during page faults, the CVMM implements fairly aggressive software TLB. Unlike the x86 TLB, the CVMM supports Address Space Number tags, used to differentiate and allow selective flushing of translations from different page tables. The CVMM TLB is naturally kept coherent within a node, so a lazy flushing scheme is particularly useful since (as we'll see later) we try to minimize cross-node process migration.

4.3 Virtual Address Space

The standard 32-bit LINUX kernel reserves the last quarter (gigabyte) of virtual addresses for its own purposes. The bottom three quarters of virtual addresses makes up the standard process (user-mode) address space.

Much efficiency is to be gained by having the CVMM share its virtual address space with the guest OS. So the LINUX kernel is mapped into the top of the CVMM's user-space, somewhat reducing the virtual address space available to LINUX users. The amount of virtual address space required by the CVMM depends on a variety of factors, including requirements of drivers for the real hardware underneath. This overhead becomes negligible on 64-bit architectures.

4.4 Booting

As discussed earlier, a guest OS kernel runs at privilege level 1 in the IA-32 ViMA. We first replaced the privileged instructions in the arch code by syscalls or other communication with the CVMM. Kernel execution starts when the CVMM is told to start a virtual server and pointed at the kernel. The boot virtual processor then starts executing the boot code. VPs are always running in protected mode with paging enabled, initially using a `null` page table signifying direct (logical = virtual) mapping. So the early boot code is fairly trivial, just establishing a stack pointer and setting up the initial kernel page tables before dispatching to C code. Boot code for secondary CPUs is even more trivial since there are no page tables to build.

4.5 Interrupts and Exceptions

The LINUX kernel registers handlers for interrupts with the CVMM via a virtualized Inter-

rupt Descriptor Table. Likewise, the CVMM provides a virtualized mechanism for masking and unmasking interrupts. Any information (e.g., cr2, etc.) necessary for processing an interrupt or exception that is normally readable only at privilege level 0 is made available to the handler running at level 1. Interrupts actually originating in hardware are delivered to the CVMM, which processes them and routes them when necessary to the appropriate virtual server interrupt handlers. Particular care is taken to provide a "fast path" for exceptions (like page faults) and interrupts generated and handled locally.

Particularly when sharing a physical processor among several virtual servers, interrupts can arrive when a virtual server is not currently running. In this case, the interrupt(s) are `pended`, possibly coalescing several for the same device into a single interrupt. Because the CVMMs handle all actual device communication, LINUX is not subject to the usual hardware constraints requiring immediate processing of device interrupts, so such coalescing is not dangerous, provided that the interrupt handlers realize the coalescing can happen and act accordingly.

4.6 I/O

The ViMA I/O interface is designed to be flexible and extensible enough to support new classes of devices as they come along. The interface is not trying to present something that looks like `real hardware`, but rather higher-level generic conduits between the guest OS and the CVMM. That is, the ViMA itself has no understanding of I/O operation semantics; it merely passes data and control signals between the guest operating system and the CVMM. It supports the following general capabilities:

- device discovery
- device configuration
- initiation of (typically asynchronous) I/O operations
- completion of asynchronous I/O operations

Because I/O performance is extremely important, data is presented in large chunks to mitigate the overhead of going through an extra layer. The only currently supported I/O devices are Console (VCON), Ethernet (VNIC), and Fibre Channel storage (VHBA). We have implemented the `bottom layer` of three new device drivers to talk to the ViMA, while the interface from above remains the same for drivers in the same class. Sometimes the interface from above is used directly by applications, and sometimes it is used by higher-level drivers. In either case, the upper levels work "as is."

In almost all cases, completion interrupts are delivered on the CPU that initiated the operation. But since CPUs (and whole nodes) may be dynamically removed, LINUX can steer outstanding completion interrupts elsewhere when necessary.

4.7 Process and Thread Scheduling

The CVMM runs one task per virtual processor, corresponding to its main thread of control. The LINUX kernel further divides these tasks to run LINUX processes and threads, starting with the vanilla SMP scheduler. This approach is more like the one taken by CoVirt[King-03] and VMWare Workstation[Sugerman-01], as opposed to having the underlying CVMM schedule individual LINUX processes as done in L4[Liedtke-95] . This is consistent with our

general approach of exposing as much information and control as possible (without compromising virtual server isolation) to the guest OS, which we assume can make better decisions because it knows the high-level context. So, other than porting the architecture-specific context-switching code, no modifications were necessary to use the LINUX scheduler.

4.8 Timekeeping

Timekeeping is somewhat tricky on such a loosely coupled system. Because the *jiffies* variable is used all over the place, updating the global value on every clock interrupt generates prohibitively expensive cross-node memory traffic. On the other hand, LINUX expects jiffies to progress uniformly. Normally *jiffies* is aliased to *jiffies_32*, the lower 32 bits of the full 64-bit *jiffies_64* counter. Through some linker magic, we make *jiffies_32* point into a special per-node page (a page whose logical address maps to a different physical page on each node), so each node maintains its own *jiffies_32*. The global *jiffies_64* is still updated every tick, which is no longer a problem since most readers are looking at *jiffies_32*. The local *jiffies_32* values are adjusted (incrementally, without going backwards) periodically to keep them in sync with the global value.

4.9 Crucial Optimizations

The work described in the previous sections is adequate to boot and run LINUX, but the resulting performance is hardly adequate for all but the most contrived benchmarks. The toughest challenges lie in minimizing remote memory access (and communication in general).

Because the design space of potentially useful optimizations is huge, we strive to focus our optimization efforts by guiding them with performance data. One of the advantages of a virtual machine is ease of instrumentation. To this end, our CVMM has a large amount of (optional) code devoted to gathering and reporting performance data, and in particular for gathering information about cross-node memory activity. Almost all of the optimizations described here were driven by observations from this performance data gathered while running our initial target applications.

4.9.1 Logical Frame Management and NUMA

When LINUX runs on non-virtualized hardware, page frames are identified by physical address, but when it runs on the ViMA, page frames are described by `logical` address. Though logical page frames are analogous to physical page frames, logical page frames have somewhat different properties:

- Logical page frames are dynamically mapped to physical page frames by the CVMM in response to page faults generated while the guest OS runs
- Logical page frames consume physical page frames only when mapped and referenced by the guest OS.
- The logical page frames reserved by the CVMM are independent of the physical page frames reserved for PC-compatible hardware and BIOS.

Suppose we have a system consisting of four dual-processor SMP nodes. Such a system can be viewed either as a “flat” eight-processor SMP machine or (via `CONFIG_NUMA`) as a two-level hierarchy of four two-processor nodes (i.e., the same as the underlying hardware). While the former view works correctly,

hiding the real topology has serious performance consequences. The NUMA kernel assumes each node manages its own range of physical pages. Though pages can be used anywhere in the system, the NUMA kernel tries to avoid frequent accesses to remote data.

In some sense, the ViMA can be treated is a virtual cache coherent NUMA (ccNUMA) machine, in that access to memory is certainly non-uniform.

By artificially associating contiguous logical page ranges with nodes, we can make our virtual server look like a ccNUMA machine. We realized much better performance by treating the virtual machine as a ccNUMA machine reflecting the underlying physical hardware. In particular the distribution of memory into more zones alleviates contention for the zone lock and lru_lock. Furthermore, the optimizations that benefit most ccNUMA machines benefit ours. And the converse is true as well. We're currently cooperating with other NUMA LINUX developers on some new optimizations that should benefit all large ccNUMA machines.

For various reasons, the most important being some limitations of memory removal support, we currently have a fictitious CPU-less `node 0` that manages all of low memory (the DMA and NORMAL zones). So HIGHMEM is divvied up between the actual nodes in proportion to their relative logical memory size.

4.9.2 Page Cache Replication

To avoid the sharing of page metadata by nodes using replicas of read-only page cache pages, we have implemented a NUMA optimization to replicate such pages on-demand from node-local memory. This improves benchmarks that do a lot of exec'ing substantially.

4.9.3 Node-Aware Batch Page Updates

Both *fork()* and *exit()* update page metadata for large numbers of pages. As currently coded, they update the metadata in the order the pages are walked. We see measurable improvements by splitting this into multiple passes, each updating the metadata only for pages on a specific node.

4.9.4 Spinlock Implementation

The i386 spinlock implementation also turned out to be problematic, as we expected. The atomic operation used to try to acquire a spinlock requires write access to the page. This works fine if the lock isn't under contention, particularly if the page is local. But if someone else is also vying for the lock, spinning as fast as possible trying to access remote data and causing poor use of resources. We continue to experiment with different spinlock implementations (which often change in response to changes in the memory access characteristics of the underlying CVMM). Currently we always try to get the lock in the usual way first. If that fails, we fall into our own `spinlock_wait()` that does a combination of "remote" reads and yielding to the CVMM before trying the atomic operation again. This avoids overloading the CVMM to the point of restricting useful work from being done.

4.9.5 Cross-Node Scheduling

The multi-level scheduling domains introduced in 2.6 LINUX kernel match very nicely with a hierarchical system like Virtual Iron VFe. However, we found that the cross-node scheduling decisions in an environment like this are based on much different factors than the schedulers for more tightly-coupled domains.

Moreover, because cross-node migration of a running program is relatively expensive, we want to keep such migrations to a minimum. So the cross-node scheduler can run *much* less often than the other domain schedulers, so it becomes permissible to take a little longer making the scheduling decision and take more factors into account. In particular, task and node memory usage are crucial—much more important than CPU load. So we have implemented a different algorithm for the cross-node scheduling domain.

The cross-node scheduling algorithm represents node load (and a task's contribution to it) with a 3-d vector whose components represent CPU, memory, and I/O usage. Loads are compared by taking the vector norm[Bubendorfer-96]. While we're still experimenting heavily with this scheduler, a few conclusions are clear. First, memory matters far more than CPU or I/O loads in a system like ours. Hence we weight the memory component of the load vector more heavily than the other two. It's also important to be smart about how tasks share memory.

Scheduling and logical memory management is tightly intertwined. Using the default NUMA memory allocation, processes try to get memory from the node on which they're running when they allocate. We'd prefer that processes use such local pages so they don't fight with other processes or the node's swap daemon when memory pressure rises. This implies that we would rather avoid moving a process after it allocates its memory. Redistributing a process to another node at *exec()* time makes a lot of sense, since the process will have its smallest footprint at that point. Processes often share data with other processes in the same process group. So we've modified *sched_exec()* to consider migrating an exec'ing process to another node only if it's a process group leader (and with even more incentive—via a lower imbalance threshold—for session group leaders). Furthermore, when *sched_exec()* does consider migrating to another node, it looks at the 3-d load vectors described earlier. This policy has been particularly good for distributing the memory load around the nodes.

4.10 Dynamic Reconfiguration Support (Hotplug Everything)

When resources (CPU or memory) are added or removed from a the cluster, the CVMM notifies the guest OS via a special "message" interrupt also used for a few other messages ("shutdown," "reboot," etc.). LINUX processes the interrupt by waking a message handler thread, which then reads the new virtual server configuration and starts taking the steps necessary to realize it. Configuration changes occur in two phases. During the first phase, all resources being removed are going away. LINUX acknowledges the change when it has reduced its resource usage accordingly. The resources are, of course, not removed until LINUX acknowledges. During the second phase, all resources being added are added (this time before LINUX acknowledges), so LINUX simply adds the new resources and acknowledges that phase. This implies certain constraints on configuration changes. For example, there must be at least one VP shared between the old and new configurations.

4.10.1 CPU and Device Hotplug

Adding and removing CPUs and devices required some porting to our methods of starting and stopping CPUs and devices, but for the most part this is much easier with idealized virtual hardware than with the real thing.

4.10.2 Node Hotplug

Adding and removing whole nodes was a little more problematic as most iterations over nodes in the system assumes online nodes are contiguous going from 0 to *numnodes-1*. Node removal can leave a "hole" which invalidates this assumption. The CPUs associated with a node are made "physically present" or absent as the node is added or removed.

4.10.3 Memory Hotplug

Memory also comes with a node (though nodes' logical memory can be increased or decreased without adding or removing nodes), and must be made hotpluggable. Unfortunately our efforts in this area proceeded independently for quite a while until we encountered the memory hotplug effort being pursued by other members of LINUX development community. We've decided to combine our efforts and plan on integrating with the new code once we move forward from 2.6.9 code base. Adding memory isn't terribly hard, though some more synchronization is needed. At the global level, a memory hotplug semaphore, analogous to the CPU hotplug semaphore, was introduced. Careful ordering of the updates to the zones allows most of the existing references to zone memory info to continue to work without locking.

Removing memory is much more difficult. Our existing approach removes only high memory, and does so by harnessing the swap daemon. A new page bit, *PG_capture*, is introduced (name borrowed from the other memory hotplug effort) to mark pages that are destined for removal. Such pages are swapped out more aggressively so that they may be reclaimed as soon as possible. Freed pages marked for capture are taken off the free lists (and out of the per-cpu pagesets), zeroed (so the CVMM can forget them), then counted as removed. During memory removal, the swap daemons on nodes losing memory are woken often to attempt to reclaim pages marked for capture. In addition, we try reclaiming targeted pages from the shrinking zones' active lists.

This approach works well on a mostly idle (or at least suspended) machine, but has a number of weaknesses, particularly when the memory in question is being actively used. Direct page migration (bypassing swap) would be an obvious performance improvement. There are pages that can't be removed for various reasons. For example, pages locked into memory via *mlock()* can't be written to disk for security reasons.

But because our logical pages aren't actually tied to nodes (but just artificially assigned to them for management purposes), we can tolerate a substantial number of "unremovable" pages. A node that has been removed, but still has some "unremovable" pages is known as a "zombie" node. No new pages are allocated from the node, but existing pages and zone data are still valid. We'll continue to try and reclaim the outstanding pages via the node's swap daemon (now running on a different node, of course). If another node is added in its place before all pages are removed, the new node can subsume the "unremovable" pages and it becomes a normal page again. In addition, it is also possible to exchange existing free pages for "unremovable" pages to reclaim space for replicas. While this scheme is currently far from perfect or universal, it works predictably in enough circumstances to be useful.

5 Conclusion

In this paper we have presented a Clustered Virtual Machine Monitor that virtualizes a set

of distributed resources into a shared memory multi-processor machine. We have ported LINUX Operating System onto this platform and it has shown to be an excellent platform for deploying a wide variety of general purpose applications.

6 Acknoweledgement

We would like to thank all the members of the Virtual Iron Software team without whom Virtual Iron VFe and this paper would not be possible. Their contribution is gratefully acknowledged.

References

[Pfister-98] Gregory F. Pfister. *In Search of Clusters, Second Edition, Prentice Hall PTR*, pp. 358–369, 1998.

[Goldberg-74] R.P. Goldberg. Survey of Virtual Machines Research. *Computer*, pp. 34–45, June 1974.

[Robin-00] J.S. Robin and C.E. Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *Proceedings of the 9th USENIX Security Symposium*, pp. 3–4, August 20, 2000.

[Goldberg-72] R.P. Goldberg. *Architectural Principles for Virtual Computer Systems.* Ph.D. Thesis, Harvard University, Cambridge, MA, 1972.

[Whitaker-00] A. Whitaker, M. Shaw, and S. Gribble. Scale and Performance in the Denali Isolation Kernel. In *ACM SIGOPS Operating System Rev.*, vol. 36, no SI, pp. 195–209, Winter 2000.

[King-03] S. King, G. Dunlap, and P. Chen Operating System Support for Virtual Machines. In *Proceedings of the 2003 USENIX Technical Conference*, 2003.

[Sugerman-01] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O Devices on VMWare Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the 2001 USENIX Technical Conference*, June, 2001.

[Liedtke-95] Dr. Jochen Liedtke. On Micro-Kernel Construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December, 1995.

[Bubendorfer-96] K.P. Bubendorfer. *Resource Based Policies for Load Distribution.* Masters Thesis, Victoria University of Wellington, Wellington, New Zealand, 1996.