Disco: Running Commodity Operating Systems on Scalable Multiprocessors

Edouard Bugnion
and
Scott Devine
and
Kinshuk Govil
and
Mendel Rosenblum
Computer Systems Laboratory, Stanford University

In this paper we examine the problem of extending modern operating systems to run efficiently on large-scale shared memory multiprocessors without a large implementation effort. Our approach brings back an idea popular in the 1970s, virtual machine monitors. We use virtual machines to run multiple commodity operating systems on a scalable multiprocessor. This solution addresses many of the challenges facing the system software for these machines. We demonstrate our approach with a prototype called Disco that runs multiple copies of Silicon Graphics' IRIX operating system on a multiprocessor. Our experience shows that the overheads of the monitor are small and that the approach provides scalability as well as the ability to deal with the non-uniform memory access time of these systems. To reduce the memory overheads associated with running multiple operating systems, virtual machines transparently share major data structures such as the program code and the file system buffer cache. We use the distributed system support of modern operating systems to export a partial single system image to the users. The overall solution achieves most of the benefits of operating systems customized for scalable multiprocessors yet it can be achieved with a significantly smaller implementation effort.

Categories and Subject Descriptors: D.4.7 [**Software**]: Operating Systems—*Organization and Design*; C.1.2 [**Computer Systems Organization**]: Multiple Data Stream Architectures—*Parallel Processors*

General Terms: Operating systems, Virtual Machines

Additional Key Words and Phrases: Scalable Multiprocessors

Disco was developed as part of the Stanford FLASH project, funded by ARPA grant DABT63-94-C-0054. Ed Bugnion is supported in part by an NSF Graduate Fellowships Award. Mendel Rosenblum is partially supported by an NSF Young Investigator Award.

Address: Computer Systems Laboratory, Stanford University, Stanford, CA, 94305, USA.
http://www-flash.stanford.edu/Disco
E-mail: {bugnion,devine,kinshuk,mendel}@cs.stanford.edu

1. INTRODUCTION

Scalable computers have moved from the research lab to the marketplace. Multiple vendors are now shipping scalable systems with configurations in the tens or even hundreds of processors. Unfortunately, the system software for these machines has often trailed hardware in reaching the functionality and reliability expected by modern computer users. Operating systems developers shoulder much of the blame for the inability to deliver on the promises of these machines. Extensive modifications to the operating system are required to efficiently support scalable machines. The size and complexity of modern operating systems have made these modifications a resource-intensive undertaking.

In this paper, we present an alternative approach for constructing the system software for these large computers. Rather than making extensive changes to existing operating systems, we insert an additional layer of software between the hardware and operating system. This layer acts like a virtual machine monitor in that multiple copies of "commodity" operating systems can be run on a single scalable computer. The monitor also allows these commodity operating systems to efficiently cooperate and share resources with each other. The resulting system contains most of the features of custom scalable operating systems developed specifically for these machines at only a fraction of their complexity and implementation cost. The use of commodity operating systems leads to systems that are both reliable and compatible with the existing computing base.

To demonstrate the approach, we have constructed a prototype system targeting the Stanford FLASH shared memory multiprocessor [Kuskin et al. 1994], an experimental cache coherent non-uniform memory architecture (CC-NUMA) machine. The prototype, called Disco, combines commodity operating systems, not originally designed for large-scale multiprocessors, to form a high performance system software base.

Disco contains many features that reduce or eliminate the problems associated with traditional virtual machine monitors. Specifically, it minimizes the overhead of virtual machines and enhances the resource sharing between virtual machines running on the same system. Disco allows operating systems running on different virtual machines to be coupled using standard distributed systems protocols such as TCP/IP and NFS. It also allows for efficient sharing of memory and disk resources between virtual machines. The sharing support allows Disco to maintain a global buffer cache which is transparently shared by all the virtual machines, even when the virtual machines communicate through standard distributed protocols.

Our experiments with realistic workloads on a detailed simulator of the FLASH machine show that Disco achieves its goals. With a few simple modifications to an existing commercial operating system, the basic overhead of virtualization ranges from 3% to 16% for all our uniprocessor workloads. We show that a system with eight virtual machines can run some workloads 1.7 faster than on a commercial symmetric multiprocessor operating system by increasing the scalability of the system software, without substantially increasing the system's memory footprint. Finally, we show that page placement and dynamic page migration and replication allow Disco to hide the NUMA-ness of the memory system, reducing the execution time by up to 37%. Early experiments on a uniprocessor SGI machine onfirm the simulation-based results.

In Section 2, we provide a more detailed presentation of the problem being addressed. Section 3 describes an overview of the approach and the challenges of using virtual machines to construct the system software for large-scale shared-memory multiprocessors. Section 4 presents the design and implementation of Disco. Section 5 shows experimental results based on machine simulation and Section 6 shows experimental results on the SGI uniprocessor. We end the paper with a discussion of related work in Section 7 and conclude in Section 8.

2. PROBLEM DESCRIPTION

This paper addresses the problems seen by computer vendors attempting to provide system software for innovative hardware. For the purposes of this paper, the innovative hardware is scalable shared memory multiprocessors, but the issues are similar for any hardware innovation that requires significant changes in the system software. For shared memory multiprocessors, research groups have demonstrated prototype operating systems such as Hive [Rosenblum et al. 1996] and Hurricane [Unrau et al. 1995] that address the challenges of fault containment and scalability. Silicon Graphics has announced the Cellular IRIX operating system to support its shared memory machine, the Origin2000 [Laudon and Lenoski 1997]. These designs require significant OS changes, including partitioning the system into scalable units, building a single system image across the units, as well as other features such as fault containment and CC-NUMA management [Verghese et al. 1996].

With the size of the system software for modern computers in the millions of lines of code, the changes for CC-NUMA machines represent a significant development cost. These changes have an impact on many of the standard modules that make up a modern system, such as virtual memory management and the scheduler. As a result, the system software for these machines is generally delivered significantly later than the hardware. Even when the changes are functionally complete, they are likely to introduce instabilities for a certain period of time.

Late, incompatible, and possibly even buggy system software can significantly impact the success of such machines, regardless of the innovations in the hardware. As the computer industry matures, users expect to carry forward their large base of existing application programs. Furthermore, with the increasing role that computers play in today's society, users are demanding highly reliable and available computing systems. The cost of achieving reliability in computers may even dwarf the benefits of the innovation in hardware for many application areas.

Computer hardware vendors that use "commodity" operating systems such as Microsoft's Windows NT [Custer 1993] face an even greater problem in obtaining operating system support for their CC-NUMA multiprocessors. These vendors need to persuade an independent company to make changes to the operating system to support the new hardware. Not only must these vendors deliver on the promises of the innovative hardware, they must also convince powerful software companies that running on their hardware is worth the effort of the port [Perez 1995]. Given this situation, it is not surprising that computer architects frequently complain about

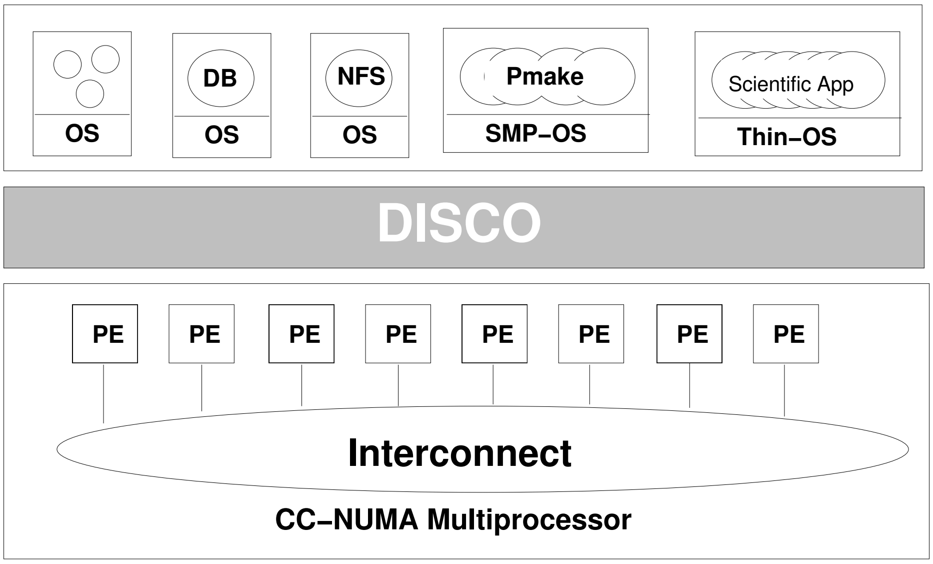

Fig. 1. **Architecture of Disco:** Disco is a virtual machine monitor, a software layer between the hardware and multiple virtual machines that run independent operating systems.

the constraints and inflexibility of system software. From their perspective, these software constraints are an impediment to innovation. To reduce the gap between hardware innovations and the adaptation of system software, system developers must find new ways to develop their software more quickly and with fewer risks of incompatibilities and instabilities.

3. A RETURN TO VIRTUAL MACHINE MONITORS

To address the problem of providing system software for scalable multiprocessors, we have developed a new twist on the relatively old idea of virtual machine monitors [Goldberg 1974]. Rather than attempting to modify existing operating systems to run on scalable shared-memory multiprocessors, we insert an additional layer of software between the hardware and the operating system. This layer of software, called a virtual machine monitor, virtualizes all the resources of the machine, exporting a more conventional hardware interface to the operating system. The monitor manages all the resources so that multiple virtual machines can coexist on the same multiprocessor.

Figure 1 shows how the virtual machine monitor allows multiple copies of potentially different operating systems to coexist. In this figure, five virtual machines coexist on the multiprocessor. Some virtual machines run commodity uniprocessor or multiprocessor operating systems, and others run specialized operating systems fine-tuned for specific workloads. The virtual machine monitor schedules the virtual resources (processor and memory) of the virtual machines on the physical resources of the scalable multiprocessor.

Virtual machine monitors, in combination with commodity and specialized operating systems, form a flexible system software solution for these machines. A large CC-NUMA multiprocessor can be configured with multiple virtual machines each running a commodity operating system such as Microsoft's Windows NT or some variant of UNIX. Each virtual machine is configured with the processor and memory resources that the operating system can effectively handle. The virtual machines communicate using standard distributed protocols to export the image of a cluster of machines.

Although the system looks like a cluster of loosely-coupled machines, the virtual machine monitor uses global policies to manage all the resources of the machine, allowing workloads to exploit the fine-grain resource sharing potential of the hardware. For example, the monitor can move memory between virtual machines to keep applications from paging to disk when free memory is available in the machine. Similarly, the monitor dynamically schedules virtual processors on the physical processors to balance the load across the machine. The use of commodity software leverages the significant engineering effort invested in these operating systems and allows CC-NUMA machines to support their large application base. Since the monitor is a relatively simple piece of code, this can be done with a small implementation effort as well as with a low risk of introducing software bugs and incompatibilities.

The approach offers two different possible solutions to handle applications whose resource needs exceed the scalability of commodity operating systems. First, a relatively simple change to the commodity operating system can allow applications to explicitly share memory regions across virtual machine boundaries. The monitor contains a simple interface to setup these shared regions. The operating system is extended with a special virtual memory segment driver to allow processes running on multiple virtual machines to share memory. For example, a parallel database server could put its buffer cache in such a shared memory region and have query engines running on multiple virtual machines.

Second, the flexibility of the approach supports specialized operating systems for resource-intensive applications that do not need the full functionality of the commodity operating systems. These simpler, specialized operating systems better support the needs of the applications and can easily scale to the size of the machine. For example, a virtual machine running a highly-scalable lightweight operating system such as Puma [Shuler et al. 1995] allows large scientific applications to scale to the size of the machine. Since the specialized operating system runs in a virtual machine, it can run alongside commodity operating systems running standard application programs. Similarly, other important applications such as database and web servers could be run in highly-customized operating systems such as database accelerators.

Besides the flexibility to support a wide variety of workloads efficiently, this approach has a number of additional advantages over other system software designs targeted for CC-NUMA machines. Running multiple copies of an operating system handles the challenges presented by CC-NUMA machines such as scalability and fault-containment. The virtual machine becomes the unit of scalability, analogous to the cell structure of Hurricane, Hive, and Cellular IRIX. With this approach, only the monitor itself and the distributed systems protocols need to scale to the size of the machine. The simplicity of the monitor makes this task easier than building a scalable operating system.

The virtual machine also becomes the unit of fault containment where failures in the system software can be contained in the virtual machine without spreading over the entire machine. To provide hardware fault-containment, the monitor itself must be structured into cells. Again, the simplicity of the monitor makes this easier than to protect a full-blown operating system against hardware faults.

NUMA memory management issues can also be handled by the monitor, effectively hiding the entire problem from the operating systems. With the careful placement of the pages of a virtual machine's memory and the use of dynamic page migration and page replication, the monitor can export a more conventional view of memory as a uniform memory access (UMA) machine. This allows the non-NUMA-aware memory management policies of commodity operating systems to work well, even on a NUMA machine.

Besides handling CC-NUMA multiprocessors, the approach also inherits all the advantages of traditional virtual machine monitors. Many of these benefits are still appropriate today and some have grown in importance. By exporting multiple virtual machines, a single CC-NUMA multiprocessor can have multiple different operating systems simultaneously running on it. Older versions of the system software can be kept around to provide a stable platform for keeping legacy applications running. Newer versions can be staged in carefully with critical applications residing on the older operating systems until the newer versions have proven themselves. This approach provides an excellent way of introducing new and innovative system software while still providing a stable computing base for applications that favor stability over innovation.

3.1 Challenges Facing Virtual Machines

Unfortunately, the advantages of using virtual machine monitors come with certain disadvantages as well. Among the well-documented problems with virtual machines are the overheads due to the virtualization of the hardware resources, resource management, sharing and communication.

Overheads. The overheads present in traditional virtual machine monitors come from many sources, including the additional exception processing, instruction execution and memory needed for virtualizing the hardware. Operations such as the execution of privileged instructions cannot be safely exported directly to the operating system and must be emulated in software by the monitor. Similarly, the access to I/O devices is virtualized, so requests must be intercepted and remapped by the monitor.

In addition to execution time overheads, running multiple independent virtual machines has a cost in additional memory. The code and data of each operating system and application is replicated in the memory of each virtual machine. Furthermore, large memory structures such as the file system buffer cache are also replicated resulting in a significant increase in memory usage. A similar waste occurs with the replication on disk of file systems for the different virtual machines.

Resource Management. Virtual machine monitors frequently experience resource management problems due to the lack of information available to the monitor to make good policy decisions. For example, the instruction execution stream of an operating system's idle loop or the code for lock busy-waiting is indistinguishable at the monitor's level from some important calculation. The result is that the monitor may schedule resources for useless computation while useful computation may be waiting. Similarly, the monitor does not know when a page is no longer being actively used by a virtual machine, so it cannot reallocate it to another virtual machine. In general, the monitor must make resource management decisions without the high-level knowledge that an operating system would have.

Communication and Sharing. Finally, running multiple independent operating systems makes sharing and communication difficult. For example under CMS on VM/370, if a virtual disk containing a user's files was in use by one virtual machine it could not be accessed by another virtual machine. The same user could not start two virtual machines, and different users could not easily share files. The virtual machines looked like a set of independent stand-alone systems that simply happened to be sharing the same hardware.

Although these disadvantages still exist, we have found their impact can be greatly reduced by combining recent advances in operating system technology with some new tricks implemented in the monitor. For example, the prevalence of support in modern operating systems for interoperating in a distributed environment greatly reduces the communication and sharing problems described above. In the following section we present techniques that lower the overheads associated with the use of virtual machines.

4. DISCO: A VIRTUAL MACHINE MONITOR

Disco is a virtual machine monitor designed for the FLASH multiprocessor [Kuskin et al. 1994], a scalable cache-coherent multiprocessor. The FLASH multiprocessor consists of a collection of nodes each containing a processor, main memory, and I/O devices. The nodes are connected together with a high-performance scalable interconnect. The machines use a directory to maintain cache coherency, providing to the software the view of a shared-memory multiprocessor with non-uniform memory access times. Although written for the FLASH machine, the hardware model assumed by Disco is also available on a number of commercial machines including the Convex Exemplar [Brewer and Astfalk 1997], Silicon Graphics Origin2000 [Laudon and Lenoski 1997], Sequent NUMAQ [Lovett and Clapp 1996], and DataGeneral NUMALiine.

This section describes the design and implementation of Disco. We first describe the key abstractions exported by Disco. We then describe the implementation of these abstractions. Finally, we discuss the operating system requirements to run on top of Disco.

4.1 Disco's Interface

Disco runs multiple independent virtual machines simultaneously on the same hardware by virtualizing all the resources of the machine. Each virtual machine can run a standard operating system that manages its virtualized resources independently of the rest of the system.

Processors. To match the FLASH machine, the virtual CPUs of Disco provide the abstraction of a MIPS R10000 processor. Disco correctly emulates all instructions, the memory management unit, and the trap architecture of the processor allowing unmodified applications and existing operating systems to run on the virtual machine. Though required for the FLASH machine, the choice of the processor was unfortunate for Disco since the R10000 does not support the complete virtualization of the kernel virtual address space. Section 4.3.1 details the OS changes needed to allow kernel-mode code to run on Disco.

Besides the emulation of the MIPS processor, Disco extends the architecture to support efficient access to some processor functions. For example, frequent kernel operations such as enabling and disabling CPU interrupts and accessing privileged registers can be performed using load and store instructions on special addresses. This interface allows operating systems tuned for Disco to reduce the overheads caused by trap emulation.

Physical Memory. Disco provides an abstraction of main memory residing in a contiguous physical address space starting at address zero. This organization was selected to match the assumptions made by the operating system.

Since most commodity operating systems are not designed to effectively manage the non-uniform memory of the FLASH machine, Disco uses dynamic page migration and replication to export a nearly uniform memory access time memory architecture to the software. This allows a non-NUMA aware operating system to run well on FLASH without the changes needed for NUMA memory management.

I/O Devices. Each virtual machine is created with a specified set of I/O devices, such as disks, network interfaces, periodic interrupt timers, clock, and a console. As with processors and physical memory, most operating systems assume exclusive access to their I/O devices, requiring Disco to virtualize each I/O device. Disco must intercept all communication to and from I/O devices to translate or emulate the operation.

Because of their importance to the overall performance and efficiency of the virtual machine, Disco exports special abstractions for the SCSI disk and network devices. Disco virtualizes disks by providing a set of virtual disks that any virtual machine can mount. Virtual disks can be configured to support different sharing and persistency models. A virtual disk can either have modifications (i.e. disk write requests) stay private to the virtual machine or they can be visible to other virtual machines. In addition, these modifications can be made persistent so that they survive the shutdown of the virtual machine or non-persistent so that they disappear with each reboot.

To support efficient communication between virtual machines, as well as other real machines, the monitor virtualizes access to the networking devices of the underlying system. Each virtual machine is assigned a distinct link-level address on an internal virtual subnet handled by Disco. Besides the standard network interfaces such as Ethernet and FDDI, Disco supports a special network interface that can handle large transfer sizes without fragmentation. For communication with the world outside the machine, Disco acts as a gateway that uses the network interfaces of the machine to send and receive packets.

4.2 Implementation of Disco

Like most operating systems that run on shared-memory multiprocessors, Disco is implemented as a multi-threaded shared-memory program. Disco differs from existing systems in that careful attention has been given to NUMA memory placement, cache-aware data structures, and interprocessor communication patterns. For example, Disco does not contain linked lists or other data structures with poor cache behavior. The small size of Disco, about 13,000 lines of code, allows for a higher degree of tuning than is possible with million line operating systems.

To improve NUMA locality, the small code segment of Disco, currently 72KB, is replicated into all the memories of FLASH machine so that all instruction cache misses can be satisfied from the local node. Machine-wide data structures are partitioned so that the parts that are accessed only or mostly by a single processor are in a memory local to that processor.

For the data structures accessed by multiple processors, very few locks are used and wait-free synchronization [Herlihy 1991] using the MIPS LL/SC instruction pair is heavily employed. Disco communicates through shared-memory in most cases. It uses inter-processor interrupts for specific actions that change the state of a remote virtual processor, for example TLB shootdowns and posting of an interrupt to a given virtual CPU. Overall, Disco is structured more like a highly tuned and scalable SPLASH application [Woo et al. 1995] than like a general-purpose operating system.

4.2.1 *Virtual CPUs.* Like previous virtual machine monitors, Disco emulates the execution of the virtual CPU by using direct execution on the real CPU. To schedule a virtual CPU, Disco sets the real machines' registers to those of the virtual CPU and jumps to the current PC of the virtual CPU. By using direct execution, most operations run at the same speed as they would on the raw hardware. The challenge of using direct execution is the detection and fast emulation of those operations that cannot be safely exported to the virtual machine. These operations are primarily the execution of privileged instructions performed by the operating system such as TLB modification, and the direct access to physical memory and I/O devices.

For each virtual CPU, Disco keeps a data structure that acts much like a process table entry in a traditional operating system. This structure contains the saved registers and other state of a virtual CPU when it is not scheduled on a real CPU. To perform the emulation of privileged instructions, Disco additionally maintains the privileged registers and TLB contents of the virtual CPU in this structure.

On the MIPS processor, Disco runs in kernel mode with full access to the machine's hardware. When control is given to a virtual machine to run, Disco puts the processor in supervisor mode if running the virtual machine's operating system, and in user mode otherwise. Supervisor mode allows the operating system to use a protected portion of the address space (the supervisor segment) but does not give access to privileged instructions or physical memory. Applications and kernel code can however still be directly executed since Disco emulates the operations that cannot be issued in supervisor mode. When a trap such as page fault, system call, or bus error occurs, the processor traps to the monitor that emulates the effect of the trap on the currently scheduled virtual processor. This is done by updating the privileged registers of the virtual processor and jumping to the virtual machine's trap vector.

Disco contains a simple scheduler that allows the virtual processors to be time-shared across the physical processors of the machine. The scheduler cooperates with the memory management to support affinity scheduling that increases data locality.

4.2.2 *Virtual Physical Memory.* To virtualize physical memory, Disco adds a level of address translation and maintains physical-to-machine address mappings. Virtual machines use *physical addresses* that have memory starting at address zero and continuing for the size of virtual machine's memory. Disco maps these physical addresses to the 40 bit *machine addresses* used by the memory system of the FLASH machine.

Disco performs this physical-to-machine translation using the software-reloaded translation-lookaside buffer (TLB) of the MIPS processor[1]. When an operating system attempts to insert a virtual-to-physical mapping into the TLB, Disco emulates this operation by translating the physical address into the corresponding machine address and inserting this corrected TLB entry into the TLB. Once the TLB entry has been established, memory references through this mapping are translated with no additional overhead by the processor. To quickly compute the corrected TLB entry, Disco keeps a per virtual machine pmap data structure that contains one entry for each physical page of a virtual machine. Each pmap entry contains a pre-computed TLB entry that references the physical page location in real memory. Disco merges that entry with the protection bits of the original entry before inserting it into the TLB. For example, a writeable mapping is only inserted in the TLB when the virtual machine requests it and the page is not marked copy-on-write. The pmap entry also contains backmaps pointing to the virtual addresses that are used to invalidate mappings from the TLB when a page is taken away from the virtual machine by the monitor.

On MIPS processors, all user mode memory references must be translated by the TLB but kernel mode references used by operating systems may directly access physical memory and I/O devices through the unmapped segment of the kernel virtual address space. Many operating systems place both the operating system code and data in this segment. Unfortunately, the MIPS architecture bypasses the TLB for this direct access segment making it impossible for Disco to efficiently remap these addresses using the TLB. Having each operating system instruction trap into the monitor would lead to unacceptable performance. We were therefore required to re-link the operating system code and data to a mapped region of the address space. This problem seems unique to MIPS as other architectures such as Alpha can remap these regions using the TLB.

The MIPS processors tag each TLB entry with an address space identifier (ASID) to avoid having to flush the TLB on MMU context switches. To avoid the complexity of virtualizing the ASIDs, Disco flushes the machine's TLB when scheduling a different virtual CPU on a physical processor. This approach speeds up the translation of the TLB entry since the ASID field provided by the virtual machine can be used directly.

A workload executing on top of Disco will suffer an increased number of TLB misses since the TLB is additionally used for all operating system references and since the TLB must be flushed on virtual CPU switches. In addition, each TLB miss

[1]A similar technique is applied on processors with a hardware-reloaded TLB such as the Intel x86. The virtual machine monitor manages the page table and prevents the virtual machine from directly inserting entries into it.

is now more expensive because of the emulation of the trap architecture, the emulation of privileged instructions in the operating systems's TLB-miss handler, and the remapping of physical addresses described above. To lessen the performance impact, Disco caches recent virtual-to-machine translations in a second-level software TLB. On each TLB miss, Disco's TLB miss handler first consults the second-level TLB. If it finds a matching virtual address it can simply place the cached mapping in the TLB, otherwise it forwards the TLB miss exception to the operating system running on the virtual machine. The effect of this optimization is that virtual machines appear to have much larger TLBs than the MIPS processors.

4.2.3 *NUMA Memory Management.* Besides providing fast translation of the virtual machine's physical addresses to real machine pages, the memory management part of Disco must also deal with the allocation of real memory to virtual machines. This is a particularly important task on CC-NUMA machines since the commodity operating system is depending on Disco to deal with the non-uniform memory access times. Disco must try to allocate memory and schedule virtual CPUs so that cache misses generated by a virtual CPU will be satisfied from local memory rather than having to suffer the additional latency of a remote cache miss. To accomplish this, Disco implements a dynamic page migration and page replication system [Bolosky et al. 1989; Cox and Fowler 1989] that moves or replicates pages to maintain locality between a virtual CPU's cache misses and the memory pages to which the cache misses occur.

Disco targets machines that maintain cache-coherence in hardware. On these machines, NUMA memory management is strictly an optimization that enhances data locality and is not required for correct exection. Disco uses a robust policy that moves only pages that will likely result in an eventual performance benefit [Verghese et al. 1996]. Pages that are heavily accessed by only one node are migrated to that node. Pages that are primarily read-shared are replicated to the nodes most heavily accessing them. Pages that are write-shared are not moved because remote accesses cannot be eliminated for all processors. Disco's policy also limits the number of times a page can move to avoid excessive overheads.

Disco's page migration and replication policy is driven by the cache miss counting facility provided by the FLASH hardware. FLASH counts cache misses to each page from every physical processor. Once FLASH detects a hot page, the monitor chooses between migrating and replicating the hot page based on the cache miss counters. To migrate a page, the monitor transparently changes the physical-to-machine mapping. It first invalidates all TLB entries mapping the old machine page and then copies the data to a local machine page. To replicate a page, the monitor must first downgrade all TLB entries mapping the machine page to ensure read-only accesses. It then copies the page to the local node and updates the relevant TLB entries mapping the old machine page. The resulting configuration after replication is shown in Figure 2. In this example, two different virtual processors of the same virtual machine logically read-share the same physical page, but each virtual processor accesses a local copy.

Disco maintains a memmap data structure that contains an entry for each real machine memory page. To perform the necessary TLB shootdowns during a page migration or replication, the memmap entry contains a list of the virtual machines

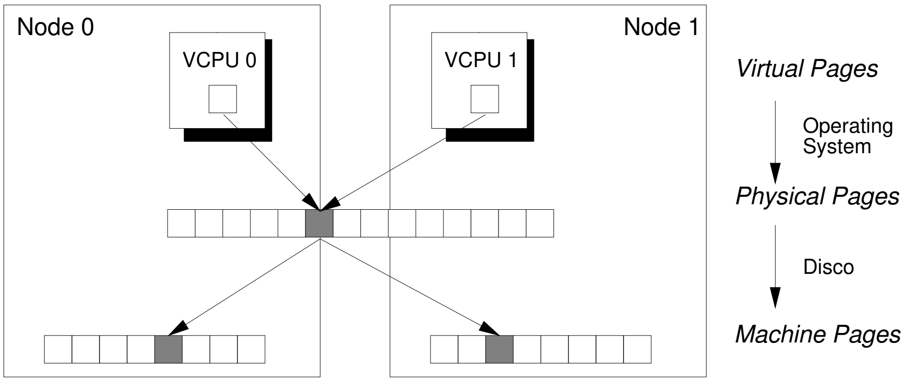

Fig. 2. Transparent Page Replication

using the page and the virtual addresses used to access them. A memmap entry also contains pointers to any replicated copies of the page.

Figure 3 summarizes the key data structures of Disco's memory management described in Sections 4.2.2 and 4.2.3 and their interactions. We discuss two examples of operations on these data structures. The first example describes the impact of a TLB miss. If the virtual address is not in the hardware TLB of the MIPS R10000, Disco's TLB miss handler will first check if the TLB entry is present in the `l2tlb` (second-level TLB) of the `vcpu` (virtual processor). If this is not the case, Disco will forward the exception to the virtual machine. The operating system's TLB miss handler will contain a TLB write instruction that is emulated by Disco. Disco uses the physical address specified by the operating system to index into the `pmap` to determine the corresponding machine address, allocating one if necessary. The `memmap` is used to determine which replica is closest to the physical processor that currently schedules the `vcpu`. Finally, the virtual-to-machine translation is inserted into the `l2tlb` and the R10000 TLB.

The second example shows the impact of a page migration action. The hardware of the FLASH machine determines that a given machine page is "hot" and Disco determines that it is suitable for migration. The transparent migration requires that all mappings that point to that page be removed from all processors. The entry in the `memmap` of that machine address contains the list of the `pmap` entries that refer to the page. The `pmap` entry contains a backmap to the virtual address and a bitmask of `vcpus` that possibly have the mapping to that machine address. Finally, all matching entries in the relevant `l2tlbs` and R10000 TLBs are invalidated before the page is actually migrated.

4.2.4 *Virtual I/O Devices.* To virtualize access to I/O devices, Disco intercepts all device accesses from the virtual machine and forwards them to the physical devices. Although it would be possible for Disco to interpose on the programmed input/output (PIOs) from the operating system device drivers and emulate the functionality of the hardware device, this approach would be complex, specific to each device, and require many traps. We found it was much cleaner to simply add special device drivers into the operating system. Each Disco device defines a

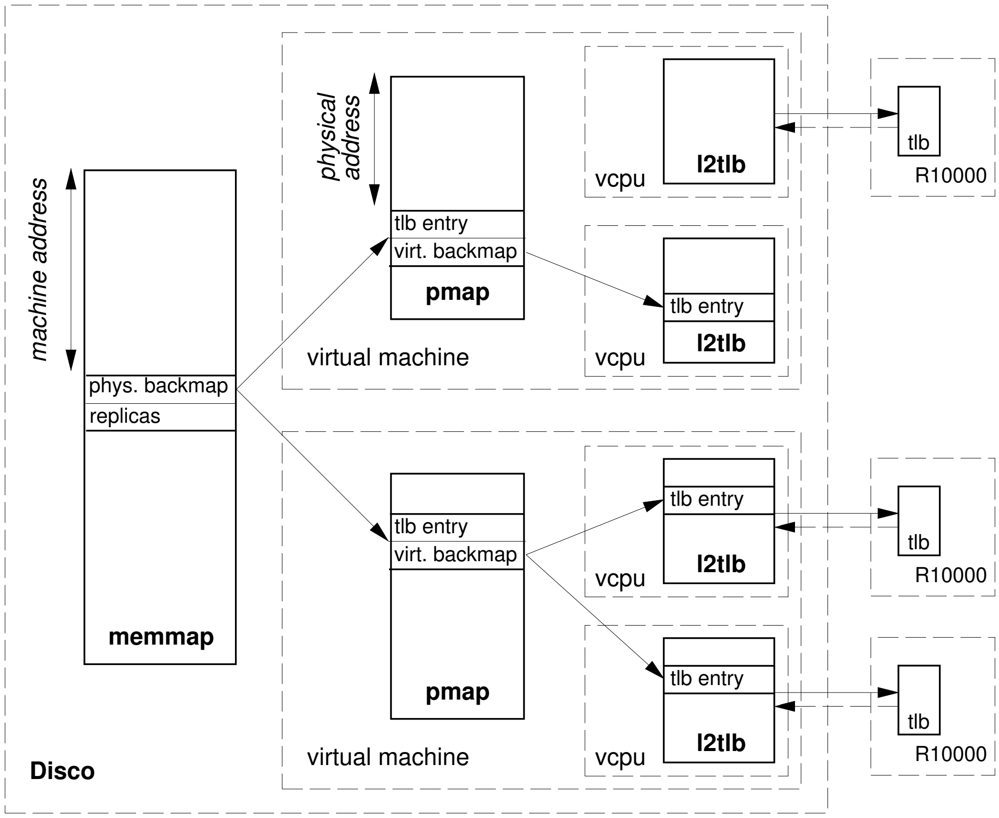

Fig. 3. Major Data Structures of Disco

monitor call used by the device driver to pass all command arguments in a single trap.

Devices such as disks and network interfaces include a DMA map as part of their arguments. A DMA map consists of a list of physical address-length pairs that specify the memory source or destination of the I/O operation. Disco must intercept such DMA requests to translate the physical addresses specified by the operating systems into machine addresses. Disco's device drivers then interact directly with the physical device. For devices accessed by a single virtual machine, Disco only needs to guarantee the exclusivity of this access and translate the physical memory addresses of the DMA, but does not need to virtualize the I/O resource itself.

The interposition on all DMA requests offers an opportunity for Disco to share disk and memory resources among virtual machines. Disco's copy-on-write disks allow virtual machines to share both main memory and disk storage resources. Disco's virtual network devices allow virtual machines to communicate efficiently. The combination of these two mechanisms, detailed in Section 4.2.5 and Section 4.2.6, allows Disco to support a system-wide cache of disk blocks in memory that can be transparently shared between all the virtual machines.

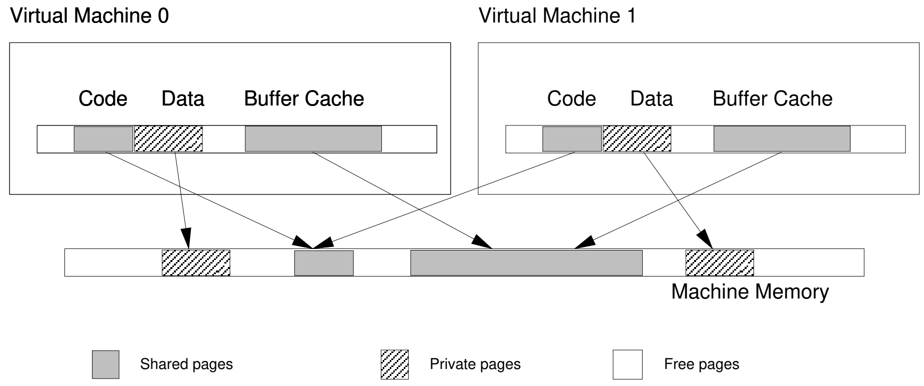

Fig. 4. Memory Sharing in Disco

4.2.5 *Copy-on-write Disks.* Disco intercepts every disk request that DMAs data into memory. When a virtual machine requests to read a disk block that is already in main memory, Disco can process the request without going to disk. Furthermore, if the disk request is a multiple of the machine's page size, Disco can process the DMA request by simply mapping the page into the virtual machine's physical memory. In order to preserve the semantics of a DMA operation, Disco maps the page read-only into the destination address page of the DMA. Attempts to modify a shared page will result in a copy-on-write fault handled internally by the monitor.

Using this mechanism, multiple virtual machines accessing a shared disk end up sharing machine memory. The copy-on-write semantics means that the virtual machine is unaware of the sharing with the exception that disk requests can finish nearly instantly. Consider an environment running multiple virtual machines for scalability purposes. All the virtual machines can share the same root disk containing the kernel and application programs. The code and other read-only data stored on the disk will be DMA-ed into memory by the first virtual machine that accesses it. Subsequent requests will simply map the page specified to the DMA engine without transferring any data. The result is shown in Figure 4 where all virtual machines share these read-only pages. Effectively we get the memory sharing patterns expected of a single shared memory multiprocessor operating system even though the system runs multiple independent operating systems.

To preserve the isolation of the virtual machines, disk writes must be kept private to the virtual machine that issues them. Disco logs the modified sectors so that the copy-on-write disk is never actually modified. For persistent disks, these modified sectors would be logged in a separate disk partition managed by Disco. To simplify our implementation, we only applied the concept of copy-on-write disks to non-persistent disks and kept the modified sectors in main memory whenever possible.

The implementation of this memory and disk sharing feature of Disco uses two data structures. For each disk device, Disco maintains a B-Tree indexed by the range of disk sectors being requested. This B-Tree is used to find the machine memory address of the sectors in the global disk cache. A second B-Tree is kept for each disk and virtual machine to find any modifications to the block made by that virtual machine. We used B-Trees to efficiently support queries on ranges of sectors [Cormen et al. 1990].

The copy-on-write mechanism is used for file systems such as the root disk whose modifications as not intended to be persistent or shared across virtual machines. For persistent disks such as the one containing user files, Disco enforces that only a single virtual machine can mount the disk at any given time. As a result, Disco does not need to virtualize the layout of the disk. Persistent disks can be accessed by other virtual machines through a distributed file system protocol such as NFS.

4.2.6 *Virtual Network Interface.* The copy-on-write mechanism for disks allows the sharing of memory resources across virtual machines, but does not allow virtual machines to communicate with each other. To communicate, virtual machines use standard distributed protocols. For example, virtual machines share files through NFS. As a result, shared data will end up in both the client's and server's buffer cache. Without special attention, the data will be duplicated in machine memory. We designed a virtual subnet managed by Disco that allows virtual machines to communicate with each other, while avoiding replicated data whenever possible.

The virtual subnet and networking interfaces of Disco also use copy-on-write mappings to reduce copying and to allow for memory sharing. The virtual device uses ethernet-like addresses and does not limit the maximum transfer unit (MTU) of packets. A message transfer sent between virtual machines causes the DMA unit to map the page read-only into both the sending and receiving virtual machine's physical address spaces. The virtual network interface accepts messages that consist of scattered buffer fragments. Our implementation of the virtual network in Disco and in the operating system's device driver always respects the data alignment of the outgoing message so that properly aligned message fragments that span a complete page are always remapped rather than copied.

Using this mechanism, a page of data read from disk into the file cache of a file server running in one virtual machine can be shared with client programs that request the file using standard distributed file system protocols such as NFS.

Figure 5 illustrates the case when the NFS reply to read request includes a data page. In (1) the monitor's networking device remaps the data page from the source's machine address space to the destination's. In (2) the monitor remaps the data page from the driver's mbuf to the clients buffer cache. This remap is initiated by the operating system through a monitor call. As a result, Disco supports a global disk cache even when a distributed file system is used to connect the virtual machines. In practice, the combination of copy-on-write disks and the access to persistent data through the specialized network device provides a global buffer cache that is transparently shared by independent virtual machines.

As a result, all read-only pages can be shared between virtual machines. Although this reduces the memory footprint, this may adversely affect data locality as most sharers will access the page remotely. However, Disco's page replication policy selectively replicates the few "hot" pages that suffer the most cache misses. Pages are therefore shared whenever possible and replicated only when necessary to improve performance.

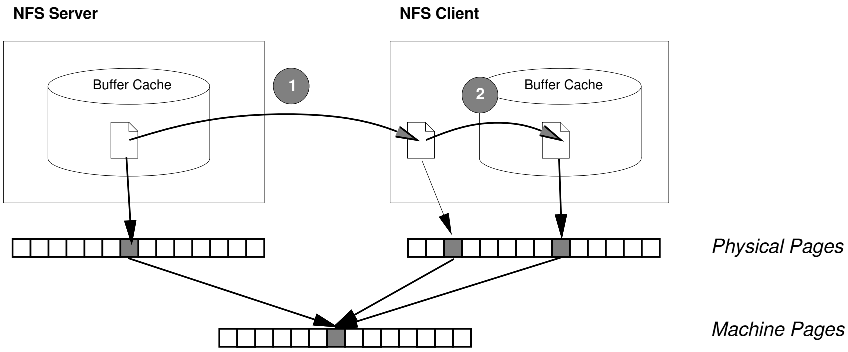

Fig. 5. Example of transparent sharing of pages Over NFS

4.3 Running Commodity Operating Systems

The "commodity" operating system we run on Disco is IRIX, a UNIX SVR4 based operating system from Silicon Graphics. Disco is however independent of any specific operating system and we plan to support others such as Windows NT and Linux.

In their support for portability, modern operating systems present a hardware abstraction level (HAL) that allows the operating system to be effectively "ported" to run on new platforms. Typically the HAL of modern operating systems changes with each new version of a machine while the rest of the system can remain unchanged. Our experience has been that relatively small changes to the HAL can reduce the overhead of virtualization and improve resource usage.

Most of the changes made in IRIX were part of the HAL[2]. All of the changes were simple enough that they are unlikely to introduce a bug in the software and did not require a detailed understanding of the internals of IRIX. Although we performed these changes at the source level as a matter of convenience, many of them were simple enough to be performed using binary translation or augmentation techniques.

4.3.1 *Necessary Changes for MIPS Architecture.* Virtual processors running in supervisor mode cannot efficiently access the KSEG0 segment of the MIPS virtual address space, that always bypasses the TLB. Unfortunately, many MIPS operating systems including IRIX 5.3 place the kernel code and data in the KSEG0 segment. As a result, we needed to relocate the unmapped segment of the virtual machines into a portion of the mapped supervisor segment of the MIPS processor. This allowed Disco to emulate the direct memory access efficiently using the TLB. The need for relocating the kernel appears to be unique to MIPS and is not present in other modern architecture such as Alpha, x86, SPARC, and PowerPC.

Making these changes to IRIX required changing two header files that describe the virtual address space layout, changing the linking options, as well as 15 assembly statements in *locore.s*. Unfortunately, this meant that we needed to re-compile and

[2]Unlike other operating systems, IRIX does not contain a documented HAL interface. In this paper, the HAL includes all the platform and processor-specific procedures of the operating system.

re-link the IRIX kernel to run on Disco.

4.3.2 *Device Drivers.* Disco's monitor call interface reduces the complexity and overhead of accessing I/O devices. We implemented UART, SCSI disks, and ethernet drivers that match this interface. Since the monitor call interface provides the view of an idealized device, the implementation of these drivers was straightforward. Since kernels are normally designed to run with different device drivers, this kind of change can be made without the source and with only a small risk of introducing a bug. The complexity of the interaction with the specific devices is left to the virtual machine monitor. Fortunately, we designed the virtual machine monitor's internal device driver interface to simplify the integration of existing drivers written for commodity operating systems. Disco uses IRIX's original device drivers.

4.3.3 *Changes to the HAL.* Having to take a trap on every privileged register access can cause significant overheads when running kernel code such as synchronization routines and trap handlers that frequently access privileged registers. To reduce this overhead, we patched the HAL of IRIX to convert these frequently used privileged instructions to use non-trapping load and store instructions to a special page of the address space that contains these registers. This optimization is only applied to instructions that read and write privileged registers without causing other side-effects. Although for this experiment we performed the patches by hand to only a few critical locations, the patches could easily be automatically applied when the privileged instruction first generates a trap. As part of the emulation process, Disco could overwrite certain instructions with the special load and store so it would not suffer the overhead of the trap again.

To help the monitor make better resource management decisions, we have added code to the HAL to pass hints to the monitor giving it higher-level knowledge of resource utilization. We inserted a small number of monitor calls in the physical memory management module of the operating systems. The first monitor call requests a zeroed page. Since the monitor must clear pages to ensure the isolation of virtual machines anyway, the operating system is freed from this task. A second monitor call informs Disco that a page has been put on the operating system's free page list without a chance of reclamation, so that Disco can immediately reclaim the memory.

To improve the utilization of processor resources, Disco assigns special semantics to the reduced power consumption mode of the MIPS processor. This mode is used by the operating system whenever the system is idle. Disco will deschedule the virtual CPU until the mode is cleared or an interrupt is posted. A monitor call inserted in the HAL's idle loop would have had the same effect.

4.3.4 *Other Changes to IRIX.* For some optimizations Disco relies on the co-operation of the operating system. For example, the virtual network device can only take advantage of the remapping techniques if the packets contain properly aligned, complete pages that are not written. We found that the operating system's networking subsystem naturally meets most of the requirements. For example, it preserves the alignment of data pages, taking advantage of the scatter/gather options of networking devices. Unfortunately, IRIX's mbuf management is such that the data pages of recently freed mbufs are linked together using the first word of the page. This guarantees that every packet transferred by the monitor's networking device using remaps will automatically trigger at least one copy-on-write fault on the receiving end. A simple change to the mbuf freelist data structure fixed this problem.

The kernel implementation of NFS always copies data from the incoming mbufs to the receiving file buffer cache, even when the packet contained un-fragmented, properly aligned pages. This would have effectively prevented the sharing of the file buffer cache across virtual machines. To have clients and servers transparently share the page, we specialized the call to bcopy to a new remap function offered by the HAL. This remap function has the semantics of a bcopy routine but uses a monitor call to remap the page whenever possible. Figure 5 shows how a data page transferred during an NFS read or write call is first remapped from the source virtual machine to the destination memory buffer (mbuf) page by the monitor's networking device, and then remapped into its final location by a call to the HAL's remap function.

4.4 SPLASHOS: A Specialized Operating System

The ability to run a thin or specialized operating system allows Disco to support large-scale parallel applications that span the entire machine. These applications may not be well served by a full function operating system. In fact, specialized operating systems such as Puma [Shuler et al. 1995] are commonly used to run scientific applications on parallel systems.

To illustrate this point, we developed a specialized library operating system [Kaashoek et al. 1997], SPLASHOS, that runs directly on top of Disco. SPLASHOS contains the services needed to run SPLASH-2 applications [Woo et al. 1995]: thread creation and synchronization routines, "libc" routines, and an NFS client stack for file I/O. The application is linked with the library operating system and runs in the same address space as the operating system. As a result, SPLASHOS does not need to support a virtual memory subsystem, deferring all page faulting responsibilities directly to Disco.

Although one might find SPLASHOS to be an overly simplistic and limited operating system if it were to run directly on hardware, the ability to run it in a virtual machine alongside commodity operating systems offers a powerful and attractive combination.

5. EXPERIMENTAL RESULTS

We have implemented Disco as described in the previous section and performed a collection of experiments to evaluate it. We describe our simulation-based experimental setup in Section 5.1. The first set of experiments presented in Sections 5.2 and 5.3 demonstrate that Disco overcomes the traditional problems associated with virtual machines, such as high overheads and poor resource sharing. We then demonstrate in Sections 5.4 and 5.5 the benefits of using virtual machines, including improved scalability and data locality.

5.1 Experimental Setup and Workloads

Disco targets the FLASH machine, which is unfortunately not yet available. As a result, we use SimOS [Rosenblum et al. 1997] to develop and evaluate Disco. SimOS is a machine simulator that models the hardware of MIPS-based multiprocessors in enough detail to run essentially unmodified system software such as the IRIX operating system and the Disco monitor. For this study, we configured SimOS to resemble a multiprocessor with performance characteristics similar to FLASH.

Although SimOS contains simulation models of the MIPS R10000 processor used in FLASH, these simulation models are too slow for the workloads that we chose to study. As a result, we model a much simpler statically scheduled, non-superscalar processors running at twice the clock rate. These simpler pipelines can be modeled an order of magnitude faster than the R10000. The processors have the on-chip caches of the MIPS R10000 (32KB split instruction/data) and a 1MB board-level cache. In the absence of memory system contention, the minimum latency of a cache miss is 300 nanoseconds to local memory and 900 nanoseconds to remote memory.

Although SimOS allows us to run realistic workloads and examine their behavior in detail with its non-intrusive annotation mechanism, the simulation slowdowns prevent us from examining long running workloads in detail. Using realistic but short workloads, we were able to study issues like the CPU and memory overheads of virtualization, the benefits on scalability, and NUMA memory management. However, studies that would require long running workloads, such as those fully evaluating Disco's resource management policies, are not possible in this environment and will hence have to wait until we have a real machine.

We chose workloads that were representative of four typical uses of scalable compute servers:

(1) Software Development (Pmake). This workload consists of the parallel compilation using Pmake of the GNU chess application using the gcc compiler. This workload is multiprogrammed, consists of many short-lived processes. It is both operating system intensive and I/O intensive. This is a particularly stressfull workload for the operating system and virtual machine monitor running below it.

(2) Hardware Development (Engineering). This multiprogrammed engineering workload consists of the concurrent simulation of part of the FLASH MAGIC chip using the VCS Verilog simulator from Chronologics and of the simulation of the memory system of the FLASH machine using the Flashlite simulator. This workload consists of long-running processes, has a large memory footprint, but makes little use of operating systems services.

(3) Scientific Computing (Raytrace, Radix). These workloads consist of the execution of a single shared-memory parallel application. The Raytrace workload renders the "car" model from the SPLASH-2 suite [27]. The Radix sorting algorithm, also from SPLASH-2, sorts 4 million integers (-n4194304 -r256 -m1073741824). Such workloads typically require very few operating systems services other than setting up shared memory regions.

(4) Commercial Database. Our commercial database workload uses the Sybase Relational Database Server to perform a decision-support workload. The decision-support queries operate on a warm database cache. As a result, the workload is not I/O intensive, but consists of a single memory-intensive application.

Table 1. Workloads and Execution Time

Environment	Workload	Execution Time
Software development	pmake -J2	3.9 sec.
Hardware development	1 VCS + 1 Flashlite	3.5 sec.
Scientific computing	raytrace -p1 -z -l0 -m64 inputs/car.env	12.9 sec.
Commercial database	Decision Support (Sybase Server)	2.0 sec.

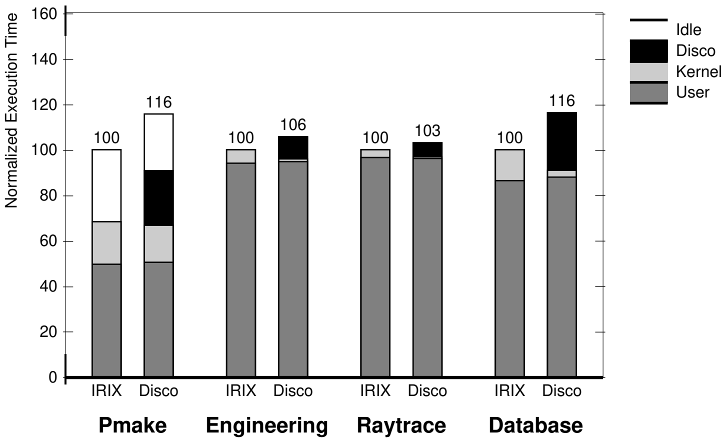

Fig. 6. Overhead of virtualization

Table 1 lists the execution time of the four uniprocessor workloads. Each workload is scaled differently for the uniprocessor and multiprocessor experiments. The reported execution time is for the uniprocessor workloads running on IRIX without Disco. The execution time does not include the time to boot the operating, ramp-up the applications and enter a steady execution state. This setup time is at least two orders of magnitude longer and performed using SimOS's fast emulation mode.

Although the simulated execution times are small, the SimOS environment allowed us to study the workload's behavior in great detail and determine that the small execution regions exhibit similar behavior to longer-running workloads. We also used the fast mode of SimOS to ensure that the workloads did not include any cold start effects. As a result, we believe that these short-running workloads exhibit a behavior similar than corresponding longer-running workload.

5.2 Execution Overheads

To evaluate the overheads of running on Disco, we ran each workload on a uniprocessor, once using IRIX directly on the simulated hardware, and once using Disco running IRIX in a single virtual machine on the same hardware. Figure 6 compares the execution time for the four uniprocessor workloads, normalized to the execution on IRIX. The execution time is separated between the the time spent in user programs, the IRIX kernel, Disco, and the idle loop.

Overall, the overhead of virtualization ranges from a modest 3% for Raytrace to a high of 16% in the Pmake and Database workloads. For the compute-bound Engineering, Raytrace and Database workloads, the overheads are mainly due to the Disco trap emulation of TLB reload misses. The Database workload has an exceptionally high TLB miss rate and hence suffer from a larger overhead. Nevertheless, the overheads of virtualization for these applications are less than 16%. The heavy use of OS services for file system and process creation in the Pmake workload makes it a particularly stressful workload for Disco. Overall, the workload suffers from a 16% slowdown and the system time (Kernel+Disco) increases by a factor of 2.15.

Figure 6 also shows a reduction in overall kernel time of some workloads. Some of the work of the operating system is being handled directly by the monitor. The reduction in Pmake is primarily due to the monitor initializing pages on behalf of the kernel and hence suffering the memory stall and instruction execution overhead of this operation. The reduction of kernel time in Raytrace, Engineering and Database workloads is due to the monitor's second-level TLB handling most TLB misses.

To understand the sources of overhead, we compare the performance of each kernel service separately. Table 2 shows the overhead of the virtualization separately for the top OS services for the Pmake workload. We focus on that workload as it is the most stressful one for the operating system and the monitor. The table first lists the importance, frequency and latency of these services for IRIX on the bare hardware. The table then decomposes the slowdown due to the virtualization in five categories: the time spent in the kernel, spent emulating TLB write instructions, spent emulating other privileged instructions, spent performing monitor services in the form of monitor calls or page faults, and finally spent handling TLB misses to emulate the virtual machine's unmapped address space segments. Kernel services include all system calls (in lower case), faults, exceptions and interrupts handlers (in upper case). DEMAND-ZERO is demand zero page fault, QUICK-FAULT is a slow TLB refill, UTLB-MISS is a fast TLB refill. COPY-ON-WRITE services a protection fault.

From Table 2, we see the overheads can significantly lengthen system services and trap handling. Short running services such as the quick page fault handler (QUICK-FAULT), where the trap overhead itself is a significant portion of the service, show slowdowns over a factor of 3. Even longer running services such as *execve* and *open* system calls show slowdowns of 1.6. These slowdowns can be explained by the common path to enter and leave the kernel for all page faults, system calls and interrupts. This path includes many privileged instructions that must be individually emulated by Disco. A restructuring of the HAL of IRIX could remove most of this overhead. For example, IRIX 5.3 uses the same TLB wired entry for different purposes in user mode and in the kernel. The path on each kernel entry and exit contains many privileged instructions that exchange this wired TLB entry and individually emulated.

We have recently brought up a 64-bit version of IRIX, IRIX 6.2, on top of a 64-bit version of Disco. IRIX6.2 differs from IRIX5.3 primarily through its use of a larger word size (64-bit vs. 32-bit) and a larger page size (16KB vs. 4KB). IRIX6.2 and the 64-bit Disco are used for our experiments on "real" hardware described in Section 6. We therefore ran the Pmake workload in this new configuration in SimOS both to predict the performance of the "real" hardware and to compare it with the IRIX 5.3 simulation results.

Table 2. Overhead breakdown for the top kernel services of the pmake workload

	Execution on IRIX				Relative Execution Time on Disco				
OS Service	Time	Count	Avg time	Slowdown	Kernel Exec	TLB Writes	Other Privs	Monitor Services	Kernel TLB faults
	IRIX 5.3 – 32 bit – 4KB pages								
DEMAND-ZERO	30%	4328	21 μs	1.42	0.43	0.21	0.16	0.47	0.16
QUICK-FAULT	10%	5745	5 μs	3.17	1.27	0.80	0.56	0.00	0.53
open	9%	667	42 μs	1.63	1.16	0.08	0.06	0.02	0.30
UTLB-MISS	7%	630 K	0.035 μs	1.35	0.07	1.22	0.05	0.00	0.02
write	6%	1610	12 μs	2.14	1.01	0.24	0.21	0.31	0.17
read	6%	733	23 μs	1.53	1.10	0.13	0.09	0.01	0.20
execve	6%	42	437 μs	1.60	0.97	0.03	0.05	0.17	0.40
	IRIX 6.2 – 64 bit – 16KB pages								
DEMAND-ZERO	27%	1134	57 μs	1.01	0.17	0.05	0.10	0.68	0.01
execve	16%	42	608μs	1.30	0.81	0.01	0.03	0.33	0.11
open	10%	667	37 μs	1.37	1.17	0.00	0.08	0.02	0.11
read	6%	733	21 μs	1.32	1.13	0.00	0.12	0.00	0.07
write	6%	1640	9 μs	1.71	0.98	0.00	0.31	0.35	0.06
COPY-ON-WRITE	5%	120	94 μs	1.16	0.59	0.03	0.06	0.41	0.06
QUICK-FAULT	5%	2196	5 μs	2.83	1.28	0.49	0.89	0.00	0.16

Table 2 shows the performance of IRIX6.2's top services. The use of larger pages reduces the pressure on the TLB and the overhead of emulating the unmapped segments. IRIX6.2 also no longer uses the same TLB entry for two different purposes in user and kernel mode, eliminating most TLB write instructions. Overall, the reduction in virtual memory faults and the cooperation of the operating system substantially reduce the overheads of the virtualization for that that workload. For the same Pmake workload, the system time "only" increases by a factor of 1.62, whereas it increased by 2.15 in the 32-bit kernel. The overall workload executes only 6% slower than IRIX6.2 on the bare hardware.

5.3 Memory Overheads

To evaluate the effectiveness of Disco's transparent memory sharing and quantify the memory overheads of running multiple virtual machines, we use a single workload running under different system configurations. The workload consists of eight different copies of the basic Pmake workload. Each Pmake instance reads and writes files from a different disk, but the different instances use the same binaries and use the same system input files. Such a workload has a great memory sharing potential in the buffer cache and performs well on multiprocessors where a single operating system manages all the resources of the machine. In contrast, these resources would have to be replicated if the workload ran on a cluster of workstations, or on a cluster of virtual machines unable to share resources. This experiment evaluates the effectiveness of Disco's memory sharing optimizations and compares its memory requirements with those of IRIX.

In all configurations we use an eight processor machine with 256 megabytes of memory and ten disks. The configurations differ in the number of virtual machines used and the access to the workload file systems. The first configuration (IRIX)

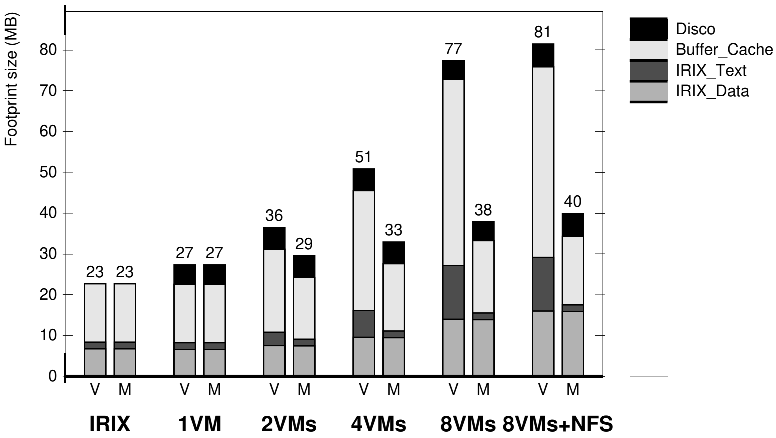

Fig. 7. Data sharing in Disco between virtual machines

runs IRIX on the bare hardware with all disks local. The next four configurations split the workload across one (1VM), two (2VMs), four (4VMs), and eight virtual machines (8VMs). Each VM has the virtual resources that correspond to an equal fraction of the physical resources. As a result, the total virtual processor and memory resources are equivalent to the total physical resources of the machine, i.e. eight processors and 256 MB of memory. For example, the 4VMs configuration consists of four dual-processor virtual machines, each with 64 MB of memory. The root disk and workload binaries are mounted from copy-on-write disks and shared among all the virtual machines. The workload file systems are mounted from different private exclusive disks.

The last configuration runs eight virtual machines but accesses workload files over NFS rather than from private disks. One of the eight virtual machines also serves as the NFS server for all file systems and is configured with 96 megabytes of memory. The seven other virtual machines have only 32MB of memory. This results in more memory configured to virtual machines than is available on the real machine. This workload shows the ability to share the file cache using standard distributed system protocols such as NFS.

Figure 7 compares the memory footprint of each configuration at the end of the workload. The virtual physical footprint (V) is the amount of memory that would be needed if Disco did not support any sharing across virtual machines. The machine footprint (M) is the amount of memory actually needed with the sharing optimizations. Pages are divided between the IRIX data structures, the IRIX text, the file system buffer cache and the Disco monitor itself.

Overall, we see that the effective sharing of the kernel text and buffer cache limits the memory overheads of running multiple virtual machines. The read-shared data is kept in a single location in memory. The kernel private data is however not shareable across virtual machines. The footprint of the kernel private data increases with the number of virtual machines, but remains overall small. For the eight virtual machine configuration, the eight copies of IRIX's data structures take less than 20 megabytes of memory.

In the NFS configuration, the virtual buffer cache is larger than the comparable local configuration as the server holds a copy of all workload files. However, that data is transparently shared with the clients and the machine buffer cache is of comparable size to the other configurations. Even using a standard distributed file system such as NFS, Disco can maintain a global buffer cache and avoid the memory overheads associated with multiple caching of data.

5.4 Scalability

We use two workloads to demonstrate the scalability benefits offered by virtual machines. The first workload is the Pmake workload and uses the six configurations of Section 5.3. This workload is particularly system-intensive and scales poorly on IRIX. The second workload is a parallel scientific application that interacts poorly with IRIX's virtual memory system and benefits from a simple specialized operating system.

IRIX5.3 is not a NUMA-aware kernel and tends to allocate its kernel data structures from a single node of FLASH causing large hot-spots. To compensate for this, we changed the physical memory layout of FLASH so that machine pages are allocated to nodes in a round-robin fashion. This round-robin allocation eliminates hot spots and results in significantly better performance for the IRIX runs. Since Disco is NUMA-aware, we were able to use the actual layout of machine memory, that allocates consecutive pages to each node. To further simplify the comparison, we disabled dynamic page migration and replication for the Disco runs.

Figure 8 shows the execution time of each workload, normalized to the execution on IRIX on the bare hardware. Radix runs in a specialized operating system (SPLASHOS) in a single virtual machine. For each workload, the execution time is broken down into user, kernel, kernel synchronization, monitor and idle time. All configurations use the same physical resources, eight processors and 256MB of memory, but the Pmake workload uses different virtual configurations.

We see in in Figure 8 that IRIX suffers from high synchronization and memory system overheads for the Pmake workload, even at just eight processors. For example, about one quarter of the overall time is spent in the kernel synchronization routines and the 67% of the remaining kernel time is spent stalled in the memory system on communication misses. The version of IRIX that we used has a known primary scalability bottleneck, memlock, the spinlock that protects the memory management data structures of IRIX [Rosenblum et al. 1995]. Other operating systems such as NT also have comparable scalability problems, even with small numbers of processors [Perl and Sites 1996].

Using a single virtual machine leads to higher overheads than in the comparable uniprocessor Pmake workload. The increase is primarily due to additional idle time. The execution of the operating system in general and of the critical regions in particular is slower on top of Disco, which increases the contention for semaphores and spinlocks in the operating system. For this workload, the increased idle time is due to additional contention on certain semaphores that protect the virtual memory subsystem of IRIX, forcing more processes to be descheduled. This interaction causes a non-linear effect in the overheads of virtualization.

However, partitioning the problem into different virtual machines significantly

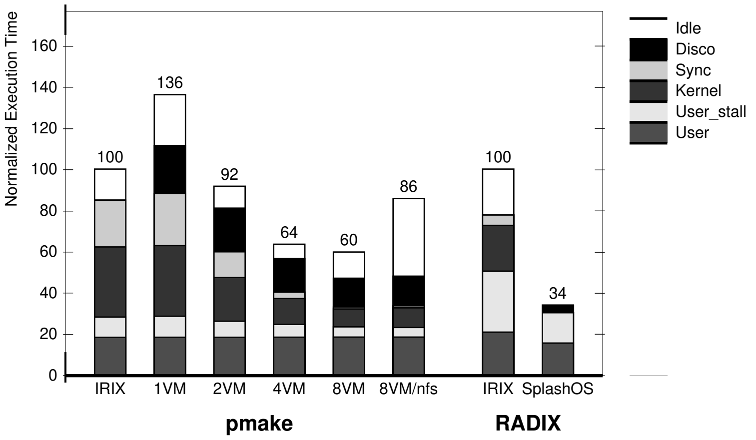

Fig. 8. Workload scalability under Disco

improves the scalability of the system. With only two virtual machines, the scalability benefits already outweigh the overheads of the virtualization. When using eight virtual machines, the execution time is reduced to 60% of its base execution time, primarily because of a significant reduction in the kernel stall time and kernel synchronization.

We see significant performance improvement even when accessing files using NFS. In the NFS configuration we see an increase in the idle time that is primarily due to the serialization of NFS requests on the single server that manages all eight disks. Even with the overheads of the NFS protocol and the increase in idle time, this configuration executes faster than the base IRIX time.

The other workload of Figure 8 compares the performance of the radix sorting algorithm, one of the SPLASH-2 applications [Woo et al. 1995]. Radix has an unfortunate interaction with the lazy evaluation policies of the IRIX virtual memory system. IRIX defers setting up the page table entries of each parallel thread until the memory is touched by the thread. When the sorting phase starts, all threads suffer many page faults on the same region causing serialization on the various spinlocks and semaphores used to protect virtual memory data structures. The contention makes the execution of these traps significant in comparison to the work Radix does for each page touched. The result is Radix spends one half of its time in the operating system.

Although it would not have been difficult to modify Radix to setup its threads differently to avoid this problem, other examples are not as easy to fix. Rather than modifying Radix, we ran it on top of SPLASHOS rather than IRIX. Because it does not manage virtual memory, SPLASHOS does not suffer from the same performance problems as IRIX. Figure 8 shows the drastic performance improvements of running the application in a specialized operating system (on top of Disco) over using a full-blown operating system (without Disco). Both configurations suffer from the same number of page faults, whose processing accounts for most of the system time. This system time is an order of magnitude larger for IRIX than it is for SPLASHOS on

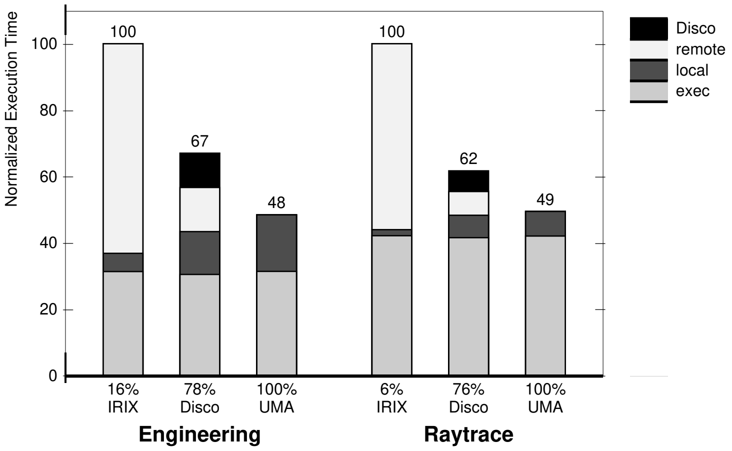

Fig. 9. Performance benefits of page migration and replication

top of Disco. The NUMA-aware allocation policy of Disco also reduces hot spots and improves user stall time.

5.5 Dynamic Page Migration and Replication

To show the benefits of Disco's page migration and replication implementation, we concentrate on workloads that exhibit poor memory system behavior, specifically the Engineering and Raytrace workloads. The Engineering workload consists of six Verilog simulations and six memory system simulations on eight processors of the same virtual machine. The Raytrace workload spans 16 processors. Because Raytrace's largest available data set fully fits in a 1MB cache, we ran the Raytrace experiments with a 256KB cache to show the impact of data locality.

Figure 9 shows the overall reduction in execution time of the workload. For each workload, the figure compares the execution on IRIX, on Disco running IRIX with migration and replication, and finally on IRIX on a bus-based UMA memory system. The UMA memory system has a latency of 300ns which is equivalent to the local latency of the NUMA machine. As a result, the performance on the UMA machine determines a lower bound for the execution time on the NUMA machine. The execution time is divided between instruction execution time, local memory stall time, remote memory stall time, and Disco overhead. The percentage of cache misses satisfied locally is shown below each bar.

The comparison between Disco and the NUMA IRIX run shows the benefits of page migration and replication while the comparison with the UMA IRIX run shows how close Disco gets to completely hiding the NUMA memory system from the workload. Disco achieves significant performance improvements by enhancing the memory locality of these workloads. The Engineering workload sees a 33% performance improvement while Raytrace gets a 38% improvement. Both user and kernel modes see a substantial reduction in remote stall time. Disco increases data locality by satisfying a large fraction of the cache misses from local memory with only a small increase in Disco's overhead.

Although Disco cannot totally hide all the NUMA memory latencies from the kernel, it does greatly improve the situation. Comparing Disco's performance with that of the optimistic UMA where all cache misses are satisfied in 300 nanoseconds, Disco comes within 40% for the Engineering workload and 26% for Raytrace.

Our implementation of page migration and replication in Disco is significantly faster than a comparable kernel implementation [Verghese et al. 1996]. This improvement is due to Disco's streamlined data structures and optimized TLB shootdown mechanisms. As a result, Disco can be more aggressive in its policy decisions and provide better data locality.

6. EXPERIENCES ON REAL HARDWARE

Although the FLASH machine is not yet available, we have ported Disco to run on an SGI Origin200 board that will form the basis of the FLASH machine. The board contains a single 180 MHz MIPS R10000 processor and 128 MB of memory. Although this work is still in its early stages, it is mature enough for us to demonstrate Disco running on real hardware as well as provide confidence in the results we obtained using SimOS.

6.1 Porting Disco

The device drivers and hardware discovery mechanisms of Disco running on SimOS are relatively simple due to the clean and simple hardware models in SimOS. This is not the case of most real computer systems that support a range of different device and hardware configurations. To simplify the port of Disco to the Origin, we run a copy of IRIX in kernel mode alongside Disco and use its hardware discovery algorithm and device drivers. We first boot IRIX but then jump to Disco rather that the standard UNIX "init" process. Disco takes control of the machine by replacing the contents of the exception vectors. It calls back into IRIX's device drivers to initiate I/O operations and to process interrupts. This approach allows us to run Disco on any SGI machine that runs IRIX with only a small implementation effort. The approach has a small execution overhead required to setup and restore IRIX's environment, measured to be about 15 μs per crossing.

6.2 Overheads of virtualization

We evaluate the performance of Disco on the Origin200 with two uniprocessor workloads. The two workloads are scaled-up versions similar to the uniprocessor workloads of Section 5.2. Although these workloads are different than the ones used for SimOS, we expect similar slowdowns and similar execution breakdowns on both systems.

(1) Pmake. This workload compiles Disco itself using the SGI development tools, two files at a time.

(2) Engineering. This workload simulates the memory system of the FLASH machine. Unlike the Engineering workload of Section 5.2, we could not include a VCS simulation due to a limitation of our VCS license.

Table 3 shows a breakdown of the execution time for the two workloads and a comparison between IRIX and Disco running IRIX. The execution time is broken down into the user, system and idle time. For IRIX, we use the built-in `time`

Table 3. Origin200 Execution Time

	Pmake			Engineering		
Breakdown	IRIX (sec.)	Disco (sec.)	Ratio	IRIX (sec.)	Disco (sec.)	Ratio
User	11.3	11.7	1.03	65.2	69.7	1.07
Kernel	5.9	9.6	1.62	0.2	0.2	1.00
Idle	13.1	11.4	0.87	0	0	
Total	30.3	32.7	1.08	65.4	69.9	1.07

command to get the breakdown. For the Disco runs, the virtual machine monitor itself maintains counters for user, system and idle execution and increments them on each timer interrupt. Since Disco runs with interrupts disabled, this approach is unable to track the time spent in Disco. The Disco overhead will be spread out among the other categories based on where interrupts are re-enabled. Services such as TLB misses in user mode will result in an increase in user time while most other Disco services will increase the kernel time.

Table 3 confirms the simulation results that the overheads of virtualization are relatively small. The Pmake workload shows a 8% overall slowdown while the simulation results of Section 5.2 showed a slowdown of 6%. The slowdown is caused by a 62% increase in system time. For the comparable IRIX6.2 Pmake workload on SimOS, described in Section 5.2, the system time also increased by 62%.

The Engineering workload shows an overall slowdown of 7%, essentially due to the workload's high TLB miss rate of 1.2 misses per 1000 cycles. A micro-benchmark measurement on the Origin200 shows that the effective cost of a TLB miss is about 92 cycles in Disco and 62 cycles in IRIX, a 48% increase. The difference is due to the software reload handler that accesses a set-associative hash table in Disco and a conventional mapped page table in IRIX.

7. RELATED WORK

We start by comparing Disco's approach to building system software for large-scale shared-memory multiprocessors with other research and commercial projects that target the same class of machines. We then compare Disco to virtual machine monitors and to other system software structuring techniques. Finally, we compare our implementation of dynamic page migration and replication with previous work.

7.1 System Software for Scalable Shared Memory Machines

Two opposite approaches are currently being taken to deal with the system software challenges of scalable shared-memory multiprocessors. The first one is to throw a large OS development effort at the problem and effectively address these challenges in the operating system. Examples of this approach are the Hive [Rosenblum et al. 1996] and Hurricane [Unrau et al. 1995] research prototypes and the Cellular-IRIX system recently announced by SGI. These multi-kernel operating systems handle the scalability of the machine by partitioning resources into "cells" that communicate to manage the hardware resources efficiently and export a single system image, effectively hiding the distributed system from the user. In Hive, the cells are also used to contain faults within cell boundaries. In addition, these systems incorporate resource allocators and schedulers for processors and memory that can handle the scalability and the NUMA aspects of the machine. This approach is innovative, but requires a large development effort.

The virtual machines of Disco are similar to the cells of Hive and Cellular-IRIX in that they support scalability and form system software fault containment boundaries. Like these systems, Disco can balance the allocation of resources such as processors and memory between these units of scalability. Also like these systems, Disco handles the NUMA memory management by doing careful page migration and replication. The benefit of Disco over the OS intensive approach is in the reduction in OS development effort. It provides a large fraction of the benefits of these systems at a fraction of the cost. Unlike the OS-intensive approach that is tied to a particular operating system, Disco is independent of any particular OS, and can even support different OSes concurrently.

The second approach is to statically partition the machine and run multiple, independent operating systems that use distributed system protocols to export a partial single system image to the users. An example of this approach is the Sun Enterprise10000 machine that handles software scalability and hardware reliability by allowing users to hard partition the machine into independent failure units each running a copy of the Solaris operating system. Users still benefit from the tight coupling of the machine, but cannot dynamically adapt the partitioning to the load of the different units. This approach favors low implementation cost and compatibility over innovation.

Like the hard partitioning approach, Disco only requires minimal OS changes. Although short of providing a full single system image, both systems build a partial single system image using standard distributed systems protocols. For example, a single file system image is built using NFS. A single system administration interface is built using NIS. System administration is simplified in Disco by the use of copy-on-write disks that are shared by many virtual machines.

Yet, unlike the hard partitioning approach, Disco can share resources between the virtual machines and supports highly dynamic reconfiguration of the machine. The support of a shared buffer cache and the ability to schedule resources of the machine between the virtual machines allows Disco to excel on workloads that would not perform well with a relatively static partitioning. Furthermore, Disco provides the ability for a single application to span all resources of the machine using a specialized scalable OS.

Digital's announced Galaxies operating system, a multi-kernel version of VMS, also partitions the machine relatively statically like the Sun machine, with the additional support for segment drivers that allow applications to share memory across partitions. Galaxies reserves a portion of the physical memory of the machine for this purpose. A comparable implementation of application-level shared memory between virtual machines would be simple and would not require having to reserve memory in advance.

Disco is a compromise between the OS-intensive and the OS-light approaches. Given an infinite OS development budget, the OS is the right place to deal with issues such as resource management. The high-level knowledge and greater control available in the operating system can allow it to export a single system image and develop better resource management mechanisms and policies. Fortunately, Disco is capable of gradually getting out of the way as the OS improves. Operating systems with improved scalability can just request larger virtual machines that manage more of the machine's resources. Disco provides an adequate and low-cost solution that enables a smooth transition and maintains compatibility with commodity operating systems.

7.2 Virtual Machine Monitors

Disco is a virtual machine monitor that implements in software a virtual machine identical to the underlying hardware. The approach itself is far from being novel. Golberg's 1974 survey paper [Goldberg 1974] lists over 70 research papers on the topic and IBM's VM/370 [IBM Corporation 1972] system was introduced in the same period. Disco shares the same approach and features as these systems, and includes many of the same performance optimizations as VM/370 [Creasy 1981]. Both allow the simultaneous execution of independent operating systems by virtualizing all the hardware resources. Both can attach I/O devices to single virtual machines in an exclusive mode. VM/370 mapped virtual disks to distinct volumes (partitions), whereas Disco has the notion of shared copy-on-write disks. Both systems support a combination of persistent disks and temporary disks. Interestingly, Creasy argues in his 1981 paper that the technology developed to interconnect virtual machines will later allow the interconnection of real machines [Creasy 1981]. The opposite occurred and Disco benefits today from the advances in distributed systems protocols.

The basic approach used in Disco as well as many of its performance optimizations were present in VM/370 and other virtual machines. Disco differs in its support of scalable shared-memory multiprocessors, handling of modern operating systems, and the transparent sharing capabilities of copy-on-write disks and the global buffer cache.

The idea of virtual machines remains popular to provide backward compatibility for legacy applications or architectures. Microsoft's Windows 95 operating system [King 1995] uses virtual machines to run older Windows 3.1 and DOS applications. Disco differs in that it runs all the system software in a virtual machine and not just the legacy applications. DAISY [Ebcioglu and Altman 1997] uses dynamic compilation techniques to run a single virtual machine with a different instruction set architecture than the host processor. Disco exports the same instruction set as the underlying hardware and can therefore use direct execution rather than dynamic compilation.

Virtual machine monitors have been recently used to provide fault-tolerance to sensitive applications. The Hypervisor system [Bressoud and Schneider 1996] virtualizes only certain resources of the machine, specifically the interrupt architecture. By running the OS in supervisor mode, it disables direct access to I/O resources and physical memory, without having to virtualize them. While this is sufficient to provide fault-tolerance, it does not allow concurrent virtual machines to coexist.

7.3 Other System Software Structuring Techniques

As an operating system structuring technique, Disco could be described as a microkernel with an unimaginative interface. Rather than developing the clean and elegant interface used by microkernels, Disco simply mirrors the interface of the raw hardware. By supporting different commodity and specialized operating systems, Disco also shares with microkernels the idea of supporting multiple operating system personalities [Accetta et al. 1986].

It is interesting to compare Disco with Exokernel [Engler et al. 1995; Kaashoek et al. 1997], a software architecture that allows application-level resource management. The Exokernel safely multiplexes resources between user-level library operating systems. Both Disco and Exokernel support specialized operating systems such as ExOS for the Aegis exokernel and SplashOS for Disco. These specialized operating systems enable superior performance since they are freed from the general overheads of commodity operating systems. Disco differs from Exokernel in that it virtualizes resources rather than multiplexing them, and can therefore run commodity operating systems without significant modifications.

The Fluke system [Ford et al. 1996] uses the virtual machine approach to build modular and extensible operating systems. Recursive virtual machines are implemented by their nested process model, and efficiency is preserved by allowing inner virtual machines to directly access the underlying microkernel of the machine. Ford et al. show that specialized system functions such as checkpointing and migration require complete state encapsulation. Like Fluke, Disco totally encapsulates the state of virtual machines, and can therefore trivially implement these functions.

7.4 CC-NUMA Memory Management

Disco provides a complete CC-NUMA memory management facility that includes page placement as well as a dynamic page migration and page replication policy. Dynamic page migration and replication was first implemented in operating systems for machines that were not cache-coherent, such as the IBM Ace [Bolosky et al. 1989] or the BBN Butterfly [Cox and Fowler 1989]. In these systems, migration and replication is triggered by page faults and the penalty of having poor data locality is greater due to the absence of caches.

The implementation in Disco is most closely related to our kernel implementation in [Verghese et al. 1996]. Both projects target the FLASH multiprocessor. Since the machine supports cache-coherency, page movement is only a performance optimization. Our policies are derived from this earlier work. Unlike the in-kernel implementation that added NUMA awareness to an existing operating system, our implementation of Disco was designed with these features in mind from the beginning, resulting in lower overheads.

8. CONCLUSIONS

This paper tackles the problem of developing system software for scalable shared memory multiprocessors without a massive development effort. Our solution involves adding a level of indirection between commodity operating systems and the raw hardware. This level of indirection uses another old idea, virtual machine monitors, to hide the novel aspects of the machine such as its size and NUMA-ness.

In a prototype implementation called Disco, we show that many of the problems of traditional virtual machines are no longer significant. Our experiments show that the overheads imposed by the virtualization are modest both in terms of processing time and memory footprint. Disco uses a combination of innovative emulation of the DMA engine and standard distributed file system protocols to support a global buffer cache that is transparently shared across all virtual machines. We show how the approach provides a simple solution to the scalability, reliability and NUMA management problems otherwise faced by the system software of large-scale machines.

Although Disco was designed to exploit shared-memory multiprocessors, the techniques it uses also apply to more loosely-coupled environments such as networks of workstations (NOW). Operations that are difficult to retrofit into clusters of existing operating systems such as checkpointing and process migration can be easily supported with a Disco-like monitor. As with shared-memory multiprocessors, this can be done with a low implementation cost and using commodity operating systems.

This return to virtual machine monitors is driven by a current trend in computer systems. While operating systems and application programs continue to grow in size and complexity, the machine-level interface has remained fairly simple. Software written to operate at this level remains simple, yet provides the necessary compatibility to leverage the large existing body of operating systems and application programs. We are interested in further exploring the use of virtual machine monitors as a way of dealing with the increasing complexity of modern computer systems.

ACKNOWLEDGMENTS

The authors would like to thank John Chapin, John Gerth, Mike Nelson, Steve Ofsthun, Rick Rashid, Volker Strumpen, and our SOSP shepherd Rich Draves for their feedback. Our colleagues Dan Teodosiu and Ben Verghese participated in many lively discussions on Disco and carefully read drafts of the paper.

REFERENCES

Accetta, M. J., Baron, R. V., Bolosky, W. J., B., G. D., Rashid, R. F., Tevanian, A., and Young, M. 1986. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition* (June 1986).

Bolosky, W. J., Fitzgerald, R. P., and Scott, M. L. 1989. Simple but effective techniques for NUMA memory management. In *Proceedings of the 12th ACM Symposium on Operating System Principles* (Dec. 1989), pp. 19–31.

Bressoud, T. C. and Schneider, F. B. 1996. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems 14*, 1 (Feb.), 80–107.

Brewer, T. and Astfalk, G. 1997. The evolution of the HP/Convex Exemplar. In *Proceedings of COMPCON Spring '97* (1997), pp. 81–96.

Cormen, T. H., Leiserson, C. E., and Rivest, R. L. 1990. *Introduction to Algorithms.* McGraw-Hill.

Cox, A. L. and Fowler, R. J. 1989. The implementation of a coherent memory abstraction on a NUMA multiprocessor: experiences with PLATINUM. In *Proceedings of the 12th ACM Symposium on Operating System Principles* (Dec. 1989), pp. 32–44.

Creasy, R. 1981. The origin of the VM/370 time-sharing system. *IBM J. Res. Develop 25*, 5, 483–490.

Custer, H. 1993. *Inside Windows NT.* Microsoft Press.

Ebcioglu, K. and Altman, E. R. 1997. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th International Symposium on Computer Architecture* (Jun 1997), pp. 26–37.

Engler, D. R., Kaashoek, M. F., and Jr., J. O. 1995. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM*

Symposium on Operating Systems Principles (Dec 1995).

Ford, B., Hibler, M., Lepreau, J., Tullman, P., Back, G., and Clawson, S. 1996. Microkernels meet recursive virtual machines. In USENIX Ed., *2nd Symposium on Operating Systems Design and Implementation* (Oct. 1996), pp. 137–151. USENIX.

Goldberg, R. P. 1974. Survey of virtual machine research. *IEEE Computer Magazine 7(6)*, 34–45.

Herlihy, M. 1991. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems 13*, 1 (Jan.), 124–149.

IBM Corporation. 1972. *IBM Virtual Machine /370 Planning Guide.*

Kaashoek, M. F., Engler, D. R., Ganger, G. R., Briceno, H. M., Hunt, R., Mazieres, D., Pinckney, T., Grimm, R., Jannotti, J., and Mackenzie, K. 1997. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles* (Oct 1997).

King, A. 1995. *Inside Windows 95.* Microsoft Press.

Kuskin, J., Ofelt, D., Heinrich, M., Heinlein, J., Simoni, R., Gharachorloo, K., Chapin, J., Nakahira, D., Baxter, J., Horowitz, M., Gupta, A., Rosenblum, M., and Hennessy, J. 1994. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture* (April 1994), pp. 302–313.

Laudon, J. and Lenoski, D. 1997. The SGI Origin: A ccNUMA highly scalable server. In *Proceedings of the 24rd Annual International Symposium on Computer Architecture* (June 1997), pp. 241–251.

Lovett, T. and Clapp, R. 1996. STiNG: A CC-NUMA computer system for the commercial marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture* (May 1996), pp. 308–317.

Perez, M. 1995. (Compaq corporation) Scalable hardware evolves, but what about the network OS? In *PCWeek interview* (Dec 1995).

Perl, S. E. and Sites, R. L. 1996. Studies of windows NT performance using dynamic execution traces. In *Proceedings of the Second Symposium on Operating System Design and Implementation* (Oct. 1996), pp. 169–184.

Rosenblum, M., Bugnion, E., Herrod, S. A., and Devine, S. 1997. Using the simOS machine simulator to study complex computer systems. *ACM Transactions on Modeling and Computer Simulation 7*, 1 (Jan.), 78–103.

Rosenblum, M., Bugnion, E., Herrod, S. A., Witchel, E., and Gupta, A. 1995. The impact of architectural trends on operating system performance. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Dec. 1995), pp. 285–298.

Rosenblum, M., Chapin, J., Teodosiu, D., Devine, S., Lahiri, T., and Gupta, A. 1996. Implementing efficient fault containment for multiprocessors: confining faults in a shared-memory multiprocessor environment. *Communications of the ACM 39*, 9 (Sept.), 52–61.

Shuler, L., Jong, C., Rieser, R., van Dresser, D., Maccabe, A. B., Fisk, L., and Stallcup, T. 1995. The Puma operating system for massively parallel computers. In *Proceedings of the Intel Supercomputer User Group Conference* (1995).

Unrau, R. C., Krieger, O., Gamsa, B., and Stumm, M. 1995. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *Journal of Supercomputing 9*, 1/2, 105–134.

Verghese, B., Devine, S., Gupta, A., and Rosenblum, M. 1996. Operating system support for improving data locality on CC-NUMA computer servers. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1996), pp. 279–289.

Woo, S. C., Ohara, M., Torrie, E., Shingh, J. P., and Gupta, A. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture* (June 1995), pp. 24–36.