Cellular Disco: Resource Management Using Virtual Clusters on Shared-Memory Multiprocessors

KINSHUK GOVIL, DAN TEODOSIU, YONGQIANG HUANG, and MENDEL ROSENBLUM
Stanford University

Despite the fact that large-scale shared-memory multiprocessors have been commercially available for several years, system software that fully utilizes all their features is still not available, mostly due to the complexity and cost of making the required changes to the operating system. A recently proposed approach, called Disco, substantially reduces this development cost by using a virtual machine monitor that leverages the existing operating system technology. In this paper we present a system called Cellular Disco that extends the Disco work to provide all the advantages of the hardware partitioning and scalable operating system approaches. We argue that Cellular Disco can achieve these benefits at only a small fraction of the development cost of modifying the operating system. Cellular Disco effectively turns a large-scale shared-memory multiprocessor into a virtual cluster that supports fault containment and heterogeneity, while avoiding operating system scalability bottlenecks. Yet at the same time, Cellular Disco preserves the benefits of a shared-memory multiprocessor by implementing dynamic, fine-grained resource sharing, and by allowing users to overcommit resources such as processors and memory. This hybrid approach requires a scalable resource manager that makes local decisions with limited information while still providing good global performance and fault containment. In this paper we describe our experience with a Cellular Disco prototype on a 32-processor SGI Origin 2000 system. We show that the execution time penalty for this approach is low, typically within 10% of the best available commercial operating system for most workloads, and that it can manage the CPU and memory resources of the machine significantly better than the hardware partitioning approach.

Categories and Subject Descriptors: D.4.1 [**Software**]: Operating Systems—*Process Management*; D.4.2 [**Software**]: Operating Systems—*Storage Management*; D.4.5 [**Software**]: Operating Systems—*Reliability*; C.1.2 [**Computer Systems Organization**]: Processor Architectures—*Parallel Architectures*

General Terms: Design, Management, Reliability, Performance

Additional Key Words and Phrases: Resource management, fault containment, virtual machines, scalable multiprocessors

Disco was developed as part of the Stanford FLASH project, funded by DARPA grant DABT63-94-C-0054.

Authors' address: Computer Systems Laboratory, Stanford University, Stanford, CA, 94305; email: {kinshuk,dant,yhuang,mendel}@cs.stanford.edu.

1. INTRODUCTION

Shared-memory multiprocessor systems with up to a few hundred processors have been commercially available for the past several years. These machines are designed as a set of tightly coupled nodes connected together with a high-speed interconnect. Each node contains a few processors, a portion of the globally distributed memory, a node controller, and possibly some I/O devices. The node controller handles all the memory coherency and I/O traffic going through the node. This design is very different from earlier bus-based multiprocessors and requires substantial modifications to the operating system to efficiently utilize the machine. Unfortunately, due to the development cost and the complexity of the required changes, most operating systems have not been adapted for these machines.

Poor scalability restricts the size of machines that can be supported by most current commercial operating systems to at most a few dozen processors. Memory allocation algorithms that are not aware of the large difference in local versus remote memory access latencies on NUMA (Non-Uniform Memory Access time) systems lead to suboptimal application performance. Resource management policies not designed to handle a large number of resources can lead to contention and inefficient usage. Finally, the inability of the operating system to survive any hardware or system software failure results in the loss of all the applications running on the system, requiring the entire machine to be rebooted.

The solutions that have been proposed to date are either based on hardware partitioning [Sequent Computer Systems 2000; Sun Microsystems 2000; Unisys 2000] or require developing new operating systems with improved scalability and fault containment characteristics [Chapin et al. 1995; Gamsa et al. 1999; IBM Corporation 2000; SGI 2000]. Unfortunately, both of these approaches suffer from serious drawbacks. Hardware partitioning limits the flexibility with which allocation and sharing of resources in a large system can be adapted to dynamically changing load requirements. Since partitioning effectively turns the system into a cluster of smaller machines, applications requiring a large number of resources will not perform well. New operating system designs can provide excellent performance, but require a considerable investment in development effort and time before reaching commercial maturity.

A recently proposed alternative approach, called Disco [Bugnion et al. 1997], uses a virtual machine monitor to run unmodified commodity operating systems on scalable multiprocessors. With a low implementation cost and a small run-time virtualization overhead, the Disco work shows that a virtual machine monitor can be used to address scalability and NUMA-awareness issues. By running multiple copies of an off-the-shelf operating system, the Disco approach is able to leverage existing operating system technology to form the system software for scalable machines.

Although Disco demonstrated the feasibility of this new approach, it left many unanswered questions. In particular, the Disco prototype lacked several major features, such as fault containment and resource management, that made it difficult to compare Disco to other approaches. For example, while other approaches such as hardware partitioning support hardware fault containment, the Disco prototype lacked such support. In addition, the Disco prototype lacked the resource management mechanisms and policies required to make it competitive compared to a customized operating system approach.

In this work we present a system called Cellular Disco that extends the basic Disco approach by supporting hardware fault containment and aggressive global resource management, and by running on actual scalable hardware. Our system effectively turns a large-scale shared-memory machine into a *virtual cluster* by combining the scalability and fault containment benefits of clusters with the resource allocation flexibility of shared-memory systems. Our experience with Cellular Disco shows that:

(1) Hardware fault containment can be added to a virtual machine monitor with very low run-time overheads and implementation costs. With a negligible performance penalty over the existing virtualization overheads, fault containment can be provided in the monitor at only a very small fraction of the development effort that would be needed for adding this support to the operating system.

(2) The virtual cluster approach can quickly and efficiently correct resource allocation imbalances in scalable systems. This capability allows Cellular Disco to manage the resources of a scalable multiprocessor significantly better than a hardware partitioning scheme and almost as well as a highly tuned operating-system-centric approach. Virtual clusters do not suffer from the resource allocation constraints of actual hardware clusters, since large applications can be allowed to use all the resources of the system, instead of being confined to a single partition.

(3) The small-scale, simulation-based results of Disco appear to match the experience of running workloads on real scalable hardware. We have built a Cellular Disco prototype that runs on a 32-processor SGI Origin 2000 [Laudon and Lenoski 1997] and is able to host multiple instances of SGI's IRIX 6.2 operating system running complex workloads. Using this system, we have shown that Cellular Disco provides all the features mentioned above while keeping the run-time overhead of virtualization below 10% for most workloads.

This paper focuses on our experience with the mechanisms and policies implemented in Cellular Disco for dealing with the interrelated challenges of hardware fault containment and global resource management.

Fault Containment. Although a virtual machine monitor automatically provides software fault containment in that a failure of one operating system instance is unlikely to harm software running in other virtual machines, the large potential size of scalable shared-memory multiprocessors also requires the ability to contain hardware faults. Cellular Disco is internally structured into a number of semiindependent cells. This design allows the impact of most hardware failures to be confined to a single cell, a behavior very similar to that of clusters, where most failures remain limited to a single node.

While Cellular Disco is organized in a cellular structure similar to the one in the Hive operating system [Chapin et al. 1995], providing fault containment in Cellular Disco required only a fraction of the development effort needed for Hive, and it does not impact performance once the virtualization cost has been factored out. A key design decision that reduced cost compared to Hive was to assume that the code of Cellular Disco itself is correct. This assumption is warranted by the fact that the size of the virtual machine monitor (50K lines of C and assembly) is small enough to be thoroughly tested.

Resource Management. In order to support better resource management than hardware clusters, Cellular Disco allows virtual machines to overcommit the actual physical resources present in the system. This offers an increased degree of flexibility by allowing Cellular Disco to dynamically adjust the fraction of the system resources assigned to each virtual machine. This approach can lead to a significantly better utilization of the system, assuming that resource requirement peaks do not occur simultaneously.

Cellular Disco multiplexes physical processors among several virtual machines, and supports memory paging in addition to any such mechanism that may be provided by the hosted operating system. These features have been carefully implemented to avoid the inefficiencies that have plagued virtual machine monitors in the past [Seawright and MacKinnon 1979]. For example, Cellular Disco tracks operating system memory usage and paging disk I/O to eliminate double-paging overheads.

Cellular Disco must manage the physical resources in the system while satisfying the often conflicting constraints of providing good fault containment and scalable resource load balancing. Since a virtual machine becomes vulnerable to faults in a cell once it starts using any resources from that cell, fault containment will only be effective if all of the resources for a given virtual machine are allocated from a small number of cells. However, a naive policy may suboptimally use the resources due to load imbalance. Resource load balancing is required to achieve efficient resource utilization in large systems. The Cellular Disco implementation of both CPU and memory load balancing was designed to preserve fault containment, avoid contention, and scale to hundreds of nodes.

In the process of virtualizing the hardware, Cellular Disco can also make many of the NUMA-specific resource management decisions for the operating system. The physical memory manager of our virtual machine monitor implements first-touch allocation and dynamic migration or replication of "hot" memory pages [Verghese et al. 1996]. These features are coupled with a physical CPU scheduler that is aware of memory locality issues.

By virtualizing the underlying hardware, Cellular Disco provides an additional level of indirection that offers an easier and more effective alternative to changing the operating system. For instance, we have added

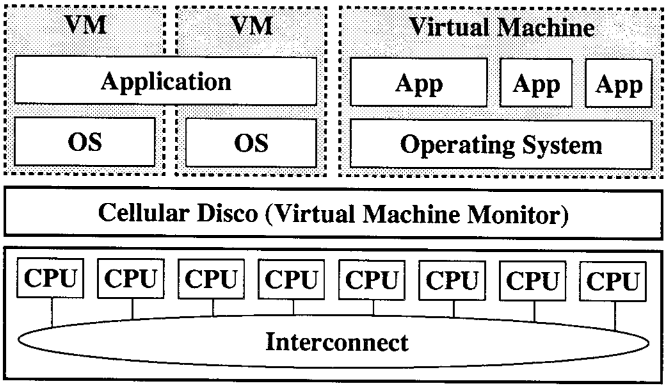

Fig. 1. Cellular Disco architecture.

support that allows large applications running across multiple virtual machines to interact directly through shared memory by registering their shared memory regions directly with the virtual machine monitor. This support allows a much more efficient interaction than through standard distributed-system protocols and can be provided transparently to the hosted operating system.

This paper is structured as follows. We start by describing the Cellular Disco architecture in Section 2. Section 3 describes the prototype implementation and the basic virtualization and fault containment overheads. Next, we discuss our resource management mechanisms and policies: CPU management in Section 4 and memory management in Section 5. Section 6 discusses hardware fault recovery. We conclude after comparing our work to hardware- and operating-system-centric approaches and discussing related work.

2. THE CELLULAR DISCO ARCHITECTURE

Compared to previous work on virtual machine monitors, Cellular Disco introduces a number of novel features: support for hardware fault containment, scalable resource management mechanisms and policies that are aware of fault containment constraints, and support for large, memory-intensive applications. For completeness, we first present a high-level overview of hardware virtualization that parallels the descriptions given in Bugnion et al. [1997] and Creasy [1981]. We then discuss each of the distinguishing new features of Cellular Disco in turn.

2.1 Overview of Hardware Virtualization

Cellular Disco is a virtual machine monitor that can execute multiple instances of an operating system by running each instance inside its own virtual machine (see Figure 1). Since the virtual machines export an interface that is similar to the underlying hardware, the operating system instances need not be aware that they are actually running on top of

Cellular Disco. This subsection presents a high-level overview of hardware virtualization and the techniques used to virtualize the MIPS architecture in Cellular Disco; see Bugnion et al. [1997] for more details.

For each newly created virtual machine, the user specifies the amount of resources that will be visible to that virtual machine by indicating the number of virtual CPUs (VCPUs), the amount of memory, and the number and type of I/O devices. The resources visible to a virtual machine are called *physical resources*. Cellular Disco allocates the actual *machine resources* to each virtual machine as required by the dynamic needs and the priority of the virtual machine, similar to the way an operating system schedules physical resources based on the needs and the priority of user applications.

To be able to virtualize the hardware, the virtual machine monitor needs to intercept all privileged operations performed by a virtual machine. This can be implemented efficiently by using the privilege levels of the processor. Although the complexity of a virtual machine monitor depends on the underlying hardware, even complex architectures such as the Intel x86 have been successfully virtualized [VMware 2000]. The MIPS processor architecture [Laudon and Lenoski 1997] that is supported by Cellular Disco has three privilege levels: *user mode* (least privileged, all memory accesses are mapped), *supervisor mode* (semiprivileged, allows mapped accesses to supervisor and user space), and *kernel mode* (most privileged, allows use of both mapped and unmapped accesses to any location, and allows execution of privileged instructions). Without virtualization, the operating system runs at kernel level, and applications execute in user mode; supervisor mode is not used. Under Cellular Disco, only the virtual machine monitor is allowed to run at kernel level, and thus to have direct access to all machine resources in the system. An operating system instance running inside a virtual machine can only access resources visible to the supervisor and user levels. Whenever a virtualized operating system kernel executes a privileged instruction, the process will trap into Cellular Disco where that instruction is emulated.

Since in supervisor mode all memory accesses are mapped, an additional level of indirection thus becomes available to map physical resources to actual machine resources. While the operating system inside a virtual machine allocates physical memory to satisfy the needs of applications, Cellular Disco allocates machine memory as needed to back the physical memory requirements of each virtual machine. Similar to Mach [Rashid et al. 1988], the virtual machine monitor maintains a *pmap* data structure for each virtual machine to map its physical addresses to actual machine addresses. In addition to the pmap, Cellular Disco needs to maintain a *memmap* structure that allows it to translate back from machine to physical pages; this structure is used for dynamic page migration and replication, and for fault recovery (see Section 6).

Performing the physical-to-machine translation using the pmap at every software reload of the MIPS TLB can lead to very high overheads. Cellular Disco reduces this overhead by maintaining for every VCPU a 1024-entry translation cache called the *second-level software TLB* (L2TLB). The entries in the L2TLB correspond to complete virtual-to-machine translations, and servicing a TLB miss from the L2TLB is much faster than generating a virtual exception to be handled by the operating system inside the virtual machine.

The limited privileges of the supervisor level also help to virtualize the I/O devices, as the operating system executing inside a virtual machine does not have enough privilege to perform I/O operations. Cellular Disco checks the validity of the I/O request and either forwards it to the real I/O device or performs the necessary actions itself in the case of devices such as the virtual paging disk (see Section 5.3). Before forwarding the I/O request to the real device, Cellular Disco translates the physical memory addresses in the DMA map to the corresponding machine addresses. Because virtualizing the device I/O interface for each type of device from every vendor is too complex, Cellular Disco only virtualizes a generic device interface for each type of device, such as a keyboard, disk, and ethernet. We installed the device drivers for these generalized devices in the virtualized operating system. After translating the memory addresses, Cellular Disco issues the actual I/O request using the device driver for the real device installed on the hardware. The details of this process are covered in Section 3.1.

Although, theoretically, hardware can be virtualized without modifying the operating system, we made some changes to IRIX 6.2 for simplicity and efficiency. A complete list of these modifications can be found in Section 4.3 of Bugnion et al. [1997].

2.2 Support for Hardware Fault Containment

As the size of shared-memory machines increases, reliability becomes a key concern for two reasons. First, one can expect to see an increase in the failure rate of large systems: a technology that fails once a year for a small workstation corresponds to a failure rate of once every three days when used in a 128-processor system. Second, since a failure will usually bring down the entire system, it can cause substantially more state loss than on a small machine. Fault tolerance does not necessarily offer a satisfactory answer for most users, due to the system cost increase and to the fact that it does not prevent operating system crashes from bringing down the entire machine.

Support for *software fault containment* (of faults occurring in the operating systems running inside the virtual machines) is a straightforward benefit of any virtual machine monitor, since the monitor can easily restrict the resources that are visible to each virtual machine. If the operating system running inside a virtual machine crashes, this will not impact any other virtual machines.

To address the reliability concerns for large machines, we designed Cellular Disco to support *hardware fault containment*, a technique that can limit the impact of faults to only a small portion of the system. After a fault, only a small fraction of the machine will be lost, together with any

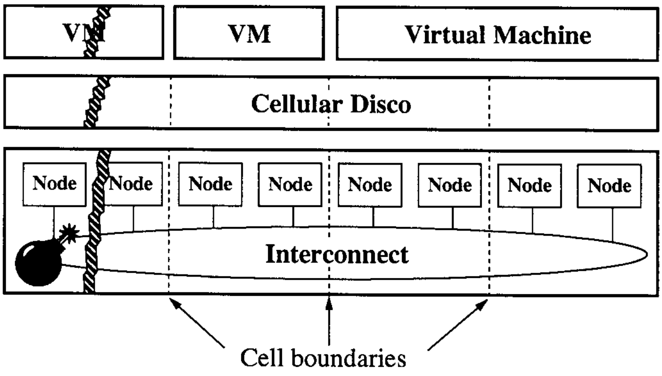

Fig. 2. Cellular structure.

applications running on that part of the system, while the rest of the system can continue executing unaffected. This behavior is similar to the one exhibited by a traditional cluster, where hardware and system software failures tend to stay localized to the node on which they occurred.

Software techniques by themselves cannot provide hardware fault containment; they require the cooperation of the hardware, as described by Teodosiu et al. [1997]. We assume that the hardware exhibits fail-stop behavior, which implies that after a fault the hardware stops working without generating erroneous results. To limit the impact of a fault, the hardware is divided into *fault containment units*. Once a fault occurs, the hardware shuts down the fault containment unit where the fault occurred, while the rest of the machine recovers itself. A fault containment unit has to be self-sufficient; therefore, it cannot be smaller than a node because a node controller failure will render the entire node useless. Further details are beyond the scope of this paper; they are covered in Teodosiu et al. [1997].

To support hardware fault containment, Cellular Disco is internally structured as a set of semiindependent cells, as shown in Figure 2. Cells consist of one or more fault containment units. Each cell contains a complete copy of the monitor code and manages its own memmap to track all the machine memory pages belonging to the cell's nodes. A failure in one cell will only bring down the virtual machines that were using resources from that cell, while virtual machines executing elsewhere will be able to continue unaffected. We designed the system to favor a smaller overhead during normal execution but a higher cost when a component fails, hopefully an infrequent occurrence. The details of the fault recovery algorithm are covered in Section 6.

One of our basic assumptions when designing Cellular Disco was that the monitor can be kept small enough to be thoroughly tested so that its probability of failure is extremely low. Cellular Disco is thus considered to be a *trusted system software layer*. This assumption is warranted by the fact that with a size of less than 50K lines, the monitor is about as complex as other trusted layers in the shared-memory machine (e.g., the cache coherence protocol implementation), and it is about two orders of magnitude simpler than modern operating systems, which may contain up to several million lines of code.

The trusted-layer decision can lead to substantially smaller overheads compared to a design in which the system software layer cannot be trusted due to its complexity, such as in the case of the Hive operating system [Chapin et al. 1995]. If cells do not trust each other, they have to use expensive distributed protocols to communicate and to update their data structures. This is substantially less efficient than directly using shared memory. The overheads become evident when one considers the case of a single virtual machine straddling multiple cells, all of which need to update the monitor data structures corresponding to the virtual machine. An example of a structure requiring frequent updates is the pmap address translation table.

Although Cellular Disco cells can use shared memory for updating virtual machine-specific data structures, Cellular Disco procedures that directly manipulate remote memory must be able to handle memory exceptions raised by the hardware if the remote cell fails. Handling these exceptions usually only requires an extra if check before accessing remote memory. Cells are not allowed to directly modify data structures in other cells that are essential for the survival of those cells. For these cases, as well as when the monitor needs to request that operations be executed on a given node or VCPU, a carefully designed communication mechanism is provided in Cellular Disco that offers low latency and exactly-once semantics.

The basic communication primitive is a fast interprocessor RPC (Remote Procedure Call). For our prototype Origin 2000 implementation, we measured the round-trip time for an RPC carrying a cache line-sized argument and reply (128 bytes) at 16 μs. Simulation results indicate that this time can be reduced to under 7 μs if appropriate support is provided in the node controller, such as in the case of the FLASH multiprocessor [Kuskin et al. 1994].

A second communication primitive, called a message, is provided for executing an action on the machine CPU that currently owns a virtual CPU. This obviates most of the need for locking, since per-VCPU operations are serialized on the owner. The cost of sending a message is on average the same as that of an RPC. Messages are based on a fault-tolerant, distributed registry that is used for locating the current owner of a VCPU given the ID of that VCPU. Since the registry is completely rebuilt after a failure, VCPUs can change owners (that is, migrate around the system) without having to depend on a fixed home. Our implementation guarantees exactly-once message semantics in the presence of contention, VCPU migration, and hardware faults.

2.3 Resource Management Under Constraints

Compared to traditional resource management issues, an additional requirement that increases complexity in Cellular Disco is fault containment.

The mechanisms and policies used in our system must carefully balance the often conflicting requirements of efficiently scheduling resources and maintaining good fault containment. While efficient resource usage requires that every available resource in the system be used when needed, good fault containment can only be provided if the set of resources used by any given virtual machine is confined to a small number of cells. Additionally, our algorithms had to be designed to scale to system sizes of up to a few hundred nodes. The above requirements had numerous implications for both CPU and memory management.

CPU Management. Operating systems for shared-memory machines normally use a global run queue to perform load sharing; each idle CPU looking for work examines the run queue to attempt to find a runnable task. Such an approach is inappropriate for Cellular Disco because it violates fault containment requirements and because it is a source of contention in large systems. In Cellular Disco, each machine processor maintains its own run queue of VCPUs. However, even with proper initial load placement, separate run queues can lead to an imbalance among the processors, due to variability in processor usage over the lifetime of the VCPUs. A load-balancing scheme is used to avoid the situation in which one portion of the machine is heavily loaded while another portion is idle. The basic load-balancing mechanism implemented in Cellular Disco is VCPU migration; our system supports intranode, intracell, and intercell migration of VCPUs. VCPU migration is used by a balancing-policy module that decides when and which VCPU to migrate, based on the current load of the system and on fault containment restrictions.

An additional feature provided by the Cellular Disco scheduler is that all nonidle VCPUs belonging to the same virtual machine are gang-scheduled. Since the operating systems running inside the virtual machines use spinlocks for their internal synchronization, gang-scheduling is necessary to avoid wasting precious cycles spinning for a lock held by a descheduled VCPU.

Memory Management. Fault containment requires that each Cellular Disco cell manage its own memory allocation. However, this can lead to a case in which a cell running a memory-intensive virtual machine may run out of memory, while other cells have free memory reserves. In a static partitioning scheme there would be no choice but to start paging data out to disk. To avoid an inefficient use of the shared-memory system, Cellular Disco implements a *memory-borrowing* mechanism through which a cell may temporarily obtain memory from other cells. Since memory borrowing may be limited by fault containment requirements, we also support paging as a fall-back mechanism.

An important aspect of our memory-balancing policies is that they carefully weigh the performance gains obtained by allocating borrowed memory versus the implications for fault containment, since using memory from a remote cell can make a virtual machine vulnerable to failures on that cell.

2.4 Support for Large Applications

In order to avoid operating system scalability bottlenecks, each operating system instance is given only as many resources as it can handle well. Applications that need fewer resources than those allocated to a virtual machine run as they normally would in a traditional system. However, large applications are forced to run across multiple virtual machines.

The solution proposed in Disco was to split large applications and have the instances on the different virtual machines communicate using distributed systems protocols that run over a fast shared-memory-based virtual ethernet provided by the virtual machine monitor. This approach is similar to the way such applications are run on a cluster or a hardware-partitioning environment. Unfortunately, this approach requires that shared-memory applications be rewritten, and incurs significant overhead introduced by communication protocols such as TCP/IP.

Cellular Disco's virtual cluster environment provides a much more efficient sharing mechanism that allows large applications to bypass the operating system and register shared-memory regions directly with the virtual machine monitor. Since every system call is intercepted first by the monitor before being reflected back to the operating system, it is easy to add in the monitor additional system call functionality for mapping global shared-memory regions. Applications running on different virtual machines can communicate through these shared-memory regions without any extra overhead because they simply use the cache-coherence mechanisms built into the hardware. The only drawback of this mechanism is that it requires relinking the application with a different shared-memory library, and possibly a few small modifications to the operating system for handling misbehaving applications.

Since the operating system instances are not aware of application-level memory sharing, the virtual machine monitor needs to provide the appropriate paging mechanisms and policies to cope with memory overload conditions. When paging out to disk, Cellular Disco needs to preserve the sharing information for pages belonging to a shared-memory region. In addition to the actual page contents, the Cellular Disco pager writes out a list of virtual machines using that page, so that sharing can be properly restored when the page is faulted back in.

3. THE CELLULAR DISCO PROTOTYPE

In this section we start by discussing our Cellular Disco prototype implementation that runs on actual scalable hardware. After describing the experimental setup, we provide evaluations of our virtualization and fault containment overheads.

3.1 Prototype Implementation

The Cellular Disco virtual machine monitor was designed to support shared-memory systems based on the MIPS R10000 processor architecture [Kane and Heinrich 1992]. Our prototype implementation consists of about

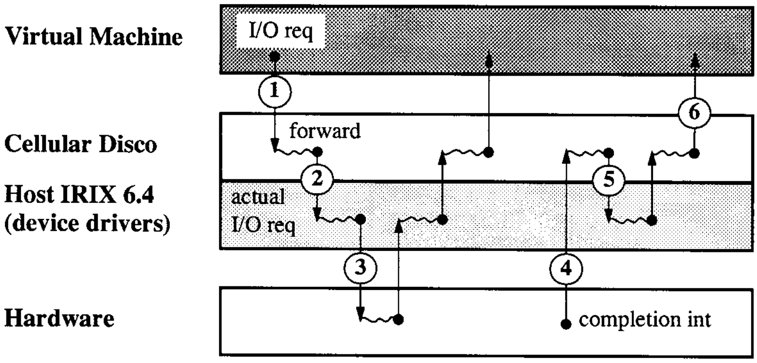

Fig. 3. I/O handling using IRIX device drivers.

50K lines of C and assembly and runs on a 32-processor SGI Origin 2000 [Laudon and Lenoski 1997].

One of the main hurdles we had to overcome in the prototype was the handling of I/O devices. Since coping with all the details of the Origin I/O hardware was beyond our available resources, we decided to leverage the device driver functionality already present in the SGI IRIX 6.4 operating system for our prototype. Our Cellular Disco implementation thus runs *piggybacked* on top of IRIX 6.4.

To run our Cellular Disco prototype, we first boot the IRIX 6.4 operating system with a minimal amount of memory. Cellular Disco is implemented as a multithreaded kernel process that spawns a thread on each CPU. The threads are pinned to their designated processors to prevent the IRIX scheduler from interfering with the control of the virtual machine monitor over the machine's CPUs. Subsequent actions performed by the monitor violate the IRIX process abstraction, effectively taking over the control of the machine from the operating system. After saving the kernel registers of the host operating system, the monitor installs its own exception handlers and takes over all remaining system memory. The host IRIX 6.4 operating system remains dormant but can be reactivated any time Cellular Disco needs to use a device driver.

Whenever one of the virtual machines created on top of Cellular Disco requests an I/O operation, the request is handled by the procedure illustrated in Figure 3. The I/O request causes a trap into Cellular Disco (1), which checks access permissions and simply forwards the request to the host IRIX (2) by restoring the saved kernel registers and exception vectors, and requesting the host kernel to issue the appropriate I/O request (3). From the perspective of the host operating system, it looks as if Cellular Disco had been running all the time just like any other well-behaved kernel process. After IRIX initiates the I/O request, control returns to Cellular Disco, which puts the host kernel back into the dormant state. Upon I/O completion the hardware raises an interrupt (4), which is handled by Cellular Disco because the exception vectors have been overwritten. To allow the host drivers to properly handle I/O completion the monitor

Table I. SGI Origin 2000 Configuration

Component	Characteristics
Processors	32 × MIPS R10000 @ 195MHz
Node controllers	16 × SGI Hub @ 100MHz
Memory	3.5GB
L2 cache size	4MB
Disks	5 (total capacity: 40GB)

reactivates the dormant IRIX, making it look as if the I/O interrupt had just been posted (5). Finally, Cellular Disco posts a virtual interrupt to the virtual machine to notify it of the completion of its I/O request (6). Since some drivers require that the kernel be aware of time, Cellular Disco forwards all timer interrupts in addition to device interrupts to the host IRIX.

Our piggybacking technique allowed us to bring up our system on real hardware quickly, and enabled Cellular Disco to handle any hardware device IRIX supports. By measuring the time spent in the host IRIX kernel, we found the overhead of the piggybacking approach to be small, less than 2% of the total running time for all the benchmarks we ran. The main drawback of our current piggybacking scheme is that it does not support hardware fault containment, given the monolithic design of the host operating system. While the fault containment experiments described in Section 6 do not use the piggybacking scheme, a solution running one copy of the host operating system per Cellular Disco cell would be possible with appropriate support in the host operating system.

3.2 Experimental Setup

We evaluated Cellular Disco by executing workloads on a 32-processor SGI Origin 2000 system configured as shown in Table I. On this machine we ran the following four workloads: Database, Pmake, Raytrace, and Web server. These workloads, described in detail in Table II, were chosen because they stress different parts of the system and because they are a representative set of applications that commercial users run on large machines. The execution times reported in this paper are the average of two stable runs after an initial warm-up run. The running times range from 4 to 6 minutes, and the noise is within 2%.

3.3 Virtualization Overheads

The performance penalty that must be paid for virtualization depends largely on the processor architecture of the virtualized system. The dominant portion of this overhead is the cost of handling the traps generated by the processor for each privileged instruction executed by the kernel.

To measure the impact of virtualization we compared the performance of the workloads executing under two different setups. First, we ran the workloads on IRIX 6.4 executing directly on top of the bare hardware. We then ran the same workloads on IRIX 6.2 executing on top of the Cellular

Table II. Workloads

Workload	Description
Database	Decision support workload based on the TPC-D [Transaction Processing Performance Council 1997] query suite on Informix Relational Database version 7.1.2 using a 200MB and a 1GB database. We measure the sum of the run times of the 17 nonupdate queries.
Pmake	I/O-intensive parallel compilation of the SGI IRIX 5.3 operating system (about 500K lines of C and assembly code).
Raytrace	CPU-intensive ray tracer from the SPLASH-2 [Woo et al. 1995] parallel benchmark suite. We used the balls4 data set with varying amounts of antialiasing so that it runs four to six minutes for single- and multiprocess configurations.
Web	Kernel-intensive Web server workload. SpecWEB96 [SPEC 2000] running on an Apache web server. Although the workload always runs for 5 minutes, we scaled the execution times so that each run performs the same number of requests.

Disco virtual machine monitor. We used two different versions of IRIX to demonstrate that Cellular Disco can leverage an off-the-shelf operating system that has only limited scalability to provide essentially the same functionality and performance as an operating system specifically designed for large-scale machines. IRIX 6.2 was designed for small-scale Challenge bus-based multiprocessors [Galles and Williams 1994], while IRIX 6.4 was the latest operating system available for the Origin 2000 when we started our experimental work. Another reason for using two different versions of IRIX is that IRIX 6.2 does not run directly on the Origin 2000. Except for scalability fixes in IRIX 6.4, the two versions are fairly similar; therefore, the uniprocessor numbers presented in this section provide a good estimate of the virtualization cost. However, multiprocessor numbers may be distorted by the scalability limitations of IRIX 6.2.

The Cellular Disco uniprocessor virtualization overheads are shown in Figure 4. As shown in the figure, the worst-case uniprocessor virtualization penalty is only 9%. For each workload, the bar on the left shows the time (normalized to 100) needed to complete the run on IRIX 6.4, while the bar on the right shows the relative time to complete the same run on IRIX 6.2 running on top of the monitor. The execution time is broken down into time spent in idle mode, in the virtual machine monitor (this portion also includes the time spent in the host kernel's device drivers), in the operating system kernel, and in user mode. This breakdown was measured by using the hardware counters of the MIPS R10000 processors.

Figure 4 also shows the virtualization overheads for 8- and 32-processor systems executing a single virtual machine that spans all the processors. We have included two cases (loaded and unloaded) for the Web workload because the two systems perform very differently depending on the load. The unloaded case limits the number of server and client processes to 16 each (half the number of processors), while the loaded case starts 32 clients and does not limit the number of server processes (the exact value is

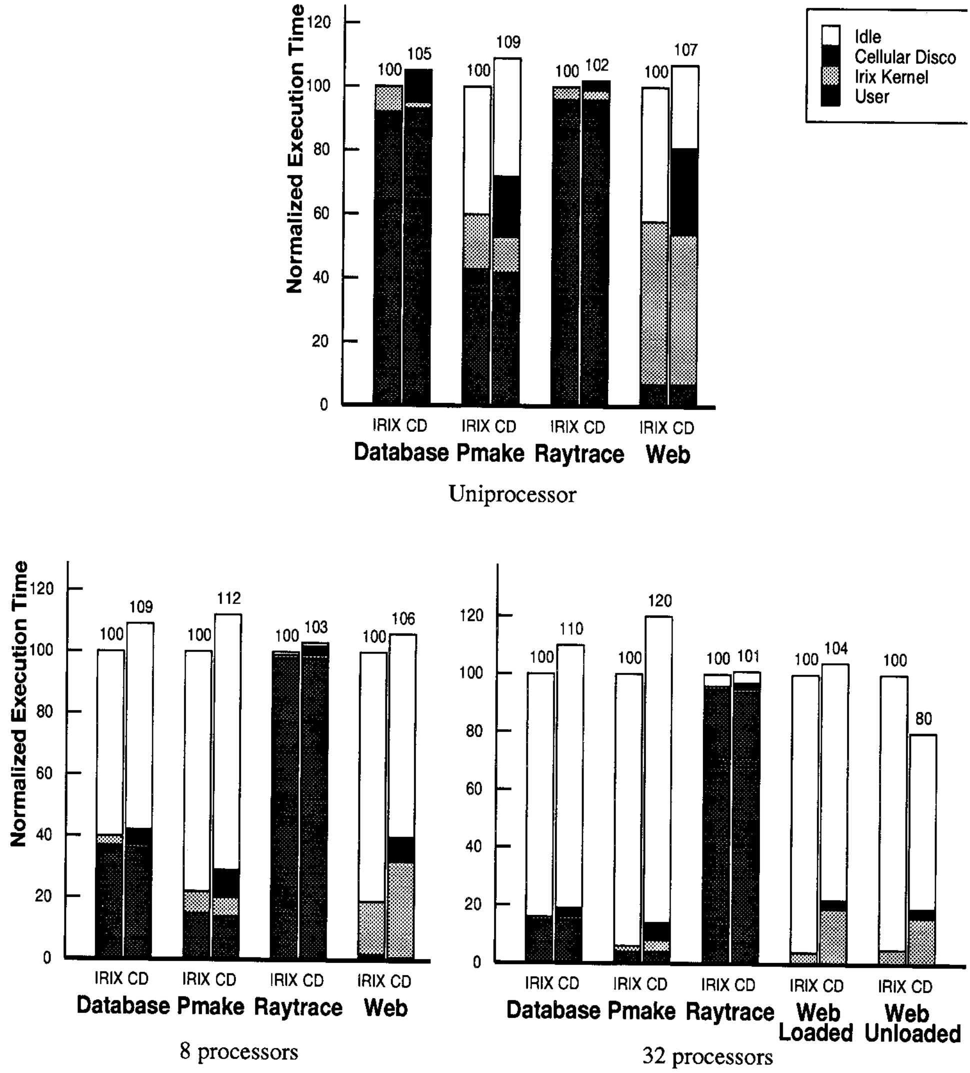

Fig. 4. Virtualization overheads.

determined by the Web server). IRIX 6.4 uses blocking locks in the networking code, which results in better performance under heavy load, while IRIX 6.2 uses spin locks, which increases kernel time but performs better under light load. The Database, Pmake, and Web benchmarks have a large amount of idle time due to their inability to fully exploit the available parallelism; a significant fraction of those workloads is serialized on a single processor. Note that on a multiprocessor virtual machine, any virtualization overheads occurring in the serial part of a workload are magnified, since they increase the idle time of the unused VCPUs. Even under such circumstances, Cellular Disco introduces only 20% overhead in the worst case.

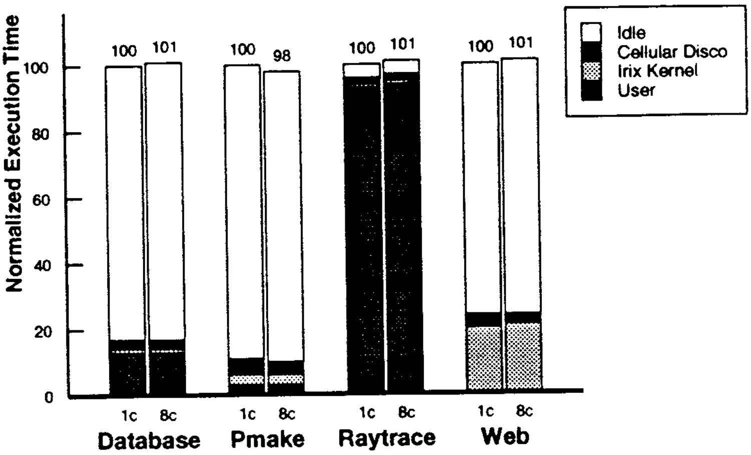

Fig. 5. Overhead of fault containment.

3.4 Fault Containment Overheads

In order to gauge the overheads introduced by the cellular structure of Cellular Disco, we ran our benchmarks on top of the virtual machine monitor using two configurations. First, the monitor was run as a single cell spanning all 32 processors in the machine, corresponding to a setup that does not provide any fault containment. Second, we booted Cellular Disco in an 8-cell configuration, with 4 processors per cell. We ran our workloads inside a 32-processor virtual machine that was completely contained in the single cell in the first case, and that spanned all 8 cells in the second one.

Figure 5 shows that the running time for virtual machines spanning cell boundaries is practically the same as when executing in a single cell (except for some small differences due to scheduling artifacts). This result shows that in Cellular Disco, hardware fault containment can be provided at practically no loss in performance once the virtualization overheads have been factored out. This result stands in sharp contrast to earlier fault containment work [Chapin et al. 1995].

4. CPU MANAGEMENT

In this section, we first describe the processor-load-balancing mechanisms provided in Cellular Disco. We then discuss the policies we use to actually balance the system. Next we discuss our implementation of gang scheduling. We conclude with an evaluation of the performance of the system and with comments on some interesting issues regarding intercell migration.

4.1 CPU Balancing Mechanisms

Cellular Disco supports three different types of VCPU migration, each providing a different trade-off between performance and cost.

The simplest VCPU migration case occurs when a VCPU is moved to a different processor on the same node (the Origin 2000 has two CPUs per node). Although the time required to update the internal monitor data structures is only 37 μs, the real cost is paid gradually over time due to the loss of CPU cache affinity. To get a rough estimate of this cost, let us assume that half of the 128-byte lines in the 4MB second-level cache are in use, with half of the active lines local and the other half remote. Refilling this amount of cached information on the destination CPU requires about 8 ms.

The second type of migration occurs when a VCPU is moved to a processor on a different node within the same cell. Compared to the cost of intranode migration, this case incurs the added cost of copying the second-level software TLB (described in Section 2.1) which is always kept on the same node as the VCPU, since it is accessed very frequently. At 520 μs, the cost for copying the entire L2TLB (32KB) is still much smaller than the gradual cost of refilling the CPU cache. However, internode migration has a higher long-term cost because the migrated VCPU is likely to access machine memory pages allocated on the previous node. Unlike the cost of cache affinity loss which is only paid once, accessing remote memory is a continuous penalty that is incurred every time the processor misses on a remote cache line. Cellular Disco alleviates this penalty by dynamically migrating or replicating frequently accessed pages to the node generating the cache misses [Verghese et al. 1996].

The third type of VCPU migration occurs when a VCPU is moved across a cell boundary; this migration costs 1520 μs including the time to copy the L2TLB. Besides losing cache and node affinity, this type of migration may also increase the fault vulnerability of the VCPU. If the latter has never before run on the destination cell and has not been using any resources from it, migrating it to the new cell will make it vulnerable to faults in that cell. However, Cellular Disco provides a mechanism through which dependencies to the old cell can be entirely removed by moving all the data used by the virtual machine over to the new cell; this process is covered in detail in Section 4.5.

4.2 CPU Balancing Policies

Cellular Disco employs two separate CPU load-balancing policies: the idle balancer and the periodic balancer. The idle balancer runs whenever a processor becomes idle, and performs most of the balancing work. The periodic balancer redistributes those VCPUs that are not handled well by the idle balancer.

When a processor becomes idle, the idle balancer runs on that processor to search for VCPUs that can be "stolen" from the run queues of neighboring processors in the same cell, starting with the closest neighbor. However, the idle balancer cannot arbitrarily select any VCPU on the remote queues due to gang-scheduling constraints. Cellular Disco will schedule a VCPU only when all the nonidle VCPUs of that virtual machine are

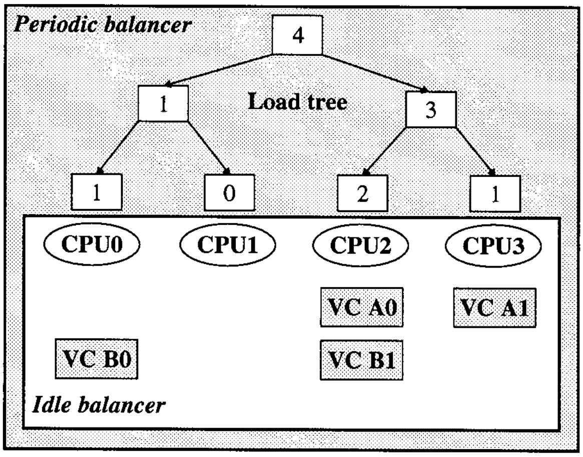

Fig. 6. CPU load-balancing scenario.

runnable. Annotations on the idle loop of the kernel inform Cellular Disco when a VCPU becomes idle. The idle balancer checks the remote queues for VCPUs that, if moved, would allow that virtual machine to run. For example, consider the case shown in Figure 6. VCPUs in the top row are currently executing on the actual machine CPUs; CPU 0 is idle due to gang-scheduling constraints. After checking the remote queues, the idle balancer running on CPU 1 will migrate VCPU B1 because the migration will allow VCPUs B0 and B1 to run on CPUs 0 and 1, respectively. Although migrating VCPU B1 would allow it to start executing right away, it may have enough cache and node affinity on CPU 2 to cancel out the gains. Cellular Disco tries to match the benefits with the cost of migration by delaying migration until a VCPU has been descheduled for some time, depending on the migration distance: 4 ms for intranode, and 6 ms for internode. These were the optimal values after testing a range from 1 ms to 10 ms; however, the overall performance only varies by 1–2% in this range.

We noticed an inefficiency with the policy in the previous paragraph after experimenting with virtual machines with many VCPUs. The previous policy migrates a VCPU only when that VCPU prevents the entire virtual machine from running. However, large virtual machines may frequently have two or more blocked VCPUs. We addressed this by modifying the idle balancer such that if it is unable to find a virtual machine with only a single blocked VCPU, and if the processor has been idle for a long time (8 ms), then it will consider migrating a VCPU even if the migration does not directly make the virtual machine runnable right away. The balancer chooses the virtual machine with the highest priority and the least number of blocked VCPUs.

The idle balancer performs well even in a fairly loaded system because there are usually still a few idle cycles available for balancing decisions due to the fragmentation caused by gang scheduling. However, by using only local load information to reduce contention, the idle balancer is not always able to make globally optimal decisions. For this reason, we included in our system a periodic balancer that uses global load information to balance load in heavily loaded systems and across different cells. Querying each processor individually is impractical for systems with hundreds of processors. Instead, each processor periodically updates the *load tree*, a low-contention distributed data structure that tracks the load of the entire system.

The load tree, shown in Figure 6, is a binary tree encompassing the entire machine. Each leaf of the tree represents a processor, and stores the load on that processor. Each inner node in the tree contains the sum of the loads of its children. To reduce memory contention, the tree nodes are physically spread across the machine. Starting from its corresponding leaf, each processor updates the tree on every 10 ms timer interrupt. Cellular Disco reduces the contention on higher-level nodes by halving the number of processors that can update a level at every level with more than 8 processors.

The periodic balancer traverses this tree depth-first, checking the load disparity between the two children. If the disparity is larger than one VCPU, the balancer will try to find a VCPU from the loaded side that is a good candidate for migration. Gang scheduling requires that two VCPUs of the same VM not be scheduled on the same processor; therefore, one of the requirements for a good candidate is that the less-loaded side must have a processor that does not already have another VCPU of the same virtual machine. If the two sides belong to different cells, then migrating a VCPU will make it vulnerable to faults in the new cell. To prevent VCPUs from being vulnerable to faults in many cells, Cellular Disco keeps track of the list of cells each VCPU is vulnerable to, and the periodic balancer prefers migrating VCPUs that are already vulnerable to faults on the less-loaded cell.

Executing the periodic balancer across the entire system can be expensive for large machines. However, running it too infrequently can cause load imbalance because heavily loaded systems can have local load imbalances that are not handled by the idle balancer due to the lack of idle cycles. Cellular Disco addresses this problem by running the periodic balancer at each level in the tree with more than 8 processors. The lowest level (8 CPU region) balancer runs every 10 ms, and each subsequent higher-level balancer runs at half that frequency. This combination allows the periodic balancer to run frequently enough at lower levels of the tree to handle local imbalances, and still balance global load without significant overhead.

4.3 Scheduling Policy

Both of the balancing schemes described in the previous section would be ineffective without a scalable gang scheduler. Most gang schedulers use either space or time partitioning, but these schemes require a centralized manager that becomes a scalability bottleneck. Cellular Disco's scheduler uses a distributed algorithm similar to the IRIX gang scheduler [Barton and Bitar 1995].

When selecting the next VCPU to run on a processor, our scheduler always picks the highest-priority gang-runnable VCPU that has been waiting the longest. A VCPU becomes gang-runnable when all the nonidle VCPUs of that virtual machine are either running or waiting on run queues of processors executing lower-priority virtual machines. After selecting a VCPU, the scheduler sends RPCs to all the processors that have VCPUs belonging to this virtual machine waiting on the run queue. On receiving this RPC, those processors deschedule the VCPU they were running, follow the same scheduling algorithm, and converge on the desired virtual machine. Each processor makes its own decisions, but ends up converging on the correct choice without employing a central global manager.

4.4 CPU Management Results

We tested the effectiveness of the complete CPU management system by running the following three-part experiment. First, we ran a single virtual machine with 8 VCPUs executing an 8-process raytrace, leaving 24 processors idle. Next, we ran four such virtual machines, each one running an 8-process raytrace. Finally, we ran eight virtual machines configured the same way, a total of 64 VCPUs running raytrace processes. An ideal system would run the first two configurations in the same time, while the third case should take twice as long. We measured only a 0.8% increase in the second case, and the final configuration took 2.04 times as long. The extra time can be attributed to migration overheads, cache affinity loss due to scheduling, and some load imbalance. To get a baseline number for the third case, we ran the same experiment on IRIX 6.4 and found that IRIX actually exhibits a higher overhead of 2.15.

Unfortunately, we did not have access to a larger machine to fully explore system scalability; therefore, we used simulation to study the effect on systems with more than 32 processors. We wrote a scheduling simulator, Schedsim, which uses the same scheduling code as Cellular Disco. The parameters of the simulator are the number of processors, cells, virtual machines, and VCPUs; and the required execution time for each VCPU. We calibrated the simulator using the numbers obtained from the 32-processor experiment performed on hardware. We estimated the impact of scalability by simulating a 128-processor system, running 16 virtual machines, each with 16 VCPUs. The end-to-end execution time was 2.17 times as long as executing just a single 16-VCPU virtual machine.

4.5 Intercell Migration Issues

Migrating VCPUs across cell boundaries raises a number of interesting issues. One of these is when to migrate the data structure associated with the entire virtual machine, not just a single VCPU. The size of this data structure is dominated by the pmap, which is proportional to the amount of physical memory the virtual machine is allowed to use. Although the

L2TLB reduces the number of accesses to the pmap, it is still desirable to place the pmap close to the VCPUs so that software-reloaded TLB misses can be satisfied quickly. Also, if all the VCPUs have migrated out of a cell, keeping the pmap in the old cell leaves the virtual machine vulnerable to faults in the old cell. We could migrate the virtual machinewide data structures when most of the VCPUs have migrated to a new cell, but the pmap is big enough that we do not want to move it that frequently. Therefore, we migrate it only when all the VCPUs have migrated to a different cell. We have carefully designed this mechanism to avoid blocking the VCPUs, which can run concurrently with this migration. This operation takes 80 ms to copy I/O-related data structures other than the pmap, and copying the pmap takes 161 μs per MB of physical memory the virtual machine is allowed to use.

Although Cellular Disco migrates the virtual machine data structures when all the VCPUs have moved away from a cell, this is not sufficient to remove vulnerability to faults occurring in the old cell. To become completely independent from the old cell, any data pages being used by a virtual machine must be migrated as well. This operation takes 25 ms per MB of memory being used by the virtual machine and can be executed without blocking any of the VCPUs.

5. MEMORY MANAGEMENT

In this section, we focus on the problem of managing machine memory across cells. We will present the mechanisms to address this problem, policies that use those mechanisms, and an evaluation of the performance of the complete system. The section concludes by looking at issues related to paging.

5.1 Memory-Balancing Mechanism

Before describing the Cellular Disco memory-balancing mechanism, it is important to discuss the memory allocation module. Each cell maintains its own freelist (list of free pages) indexed by the home node of each memory page. Initially, the freelist entries for nodes not belonging to this cell are empty, as the cell has not yet borrowed any memory. Every page allocation request is tagged with a list of cells that can supply the memory (this list is initialized when a virtual machine is created). When satisfying a request, preference is given to memory from the local node and local cell, in order to reduce the memory access latency on NUMA systems (first-touch allocation strategy).

The memory-balancing mechanism is fairly straightforward. A cell wishing to borrow memory issues a fast RPC to a cell which has available memory. The lender cell allocates memory from its freelist and returns a list of machine pages as the result of the RPC. The borrower adds those pages to its freelist, indexed by their home node. This operation takes 758 μs to borrow 4MB of memory.

5.2 Memory-Balancing Policies

A cell starts borrowing memory when its number of free pages reaches a low threshold, but before completely running out of pages. This policy seeks to avoid forcing small virtual machines that fit into a single cell to have to use remote memory. For example, consider the case of a cell with two virtual machines: one with a large memory footprint, and one that entirely fits into the cell. The large virtual machine will have to use remote memory to avoid paging, but the smaller one can achieve good performance with just local memory, without becoming vulnerable to faults in other cells. The cell must carefully decide when to allocate remote memory so that enough local memory is available to satisfy the requirements of the smaller virtual machine.

Depending on their fault containment requirements, users can restrict the set of cells from which a virtual machine can use borrowed memory. Paging must be used as a last recourse if free memory is not available from any of the cells in this list. To avoid paging as much as possible, a cell should borrow memory from cells that are listed in the allocation preferences of the virtual machines it is executing. Therefore, every cell keeps track of the combined allocation preferences of all the virtual machines it is executing, and adjusts that list whenever a virtual machine migrates into or out of the cell.

A policy we have found to be effective is the following: when the local free memory of a cell drops below 16MB, the cell tries to maintain at least 4MB of free memory from each cell in its allocation preferences list; the cell borrows 4MB from each cell in the list from which it has less than 4MB available. This heuristic biases the borrowing policy to solicit memory from cells that actively supply pages to at least one virtual machine. Cells will agree to loan memory as long as they have more than 32MB available. Cells periodically publish the amount of available memory to stop other cells from requesting memory when free memory is running low. The above thresholds are all tunable parameters. These default values were selected to provide hysteresis for stability, and they are based on the number of pages that can be allocated during the interval between consecutive executions of the policy, every 10 ms. In this duration, each CPU can allocate at most 732KB, which means that a typical cell with 8 CPUs can only allocate 6MB in 10 ms if all the CPUs allocate memory as fast as possible, a very unlikely scenario; therefore, we decided to borrow 4MB at a time. Cells start borrowing when only 16MB are left because we expect the resident size of small virtual machines to be in the range of 10–15 megabytes.

We measured the effectiveness of this policy by running a 4-processor Database workload. First, we ran the benchmark with the monitor configured as a single cell, in which case there is no need for balancing. Next, we ran in an 8-cell configuration, with 4 CPUs per cell. In the second configuration, the cell executing the Database virtual machine did not have enough memory to satisfy the workload and ended up borrowing 596MB of memory from the other cells. Borrowing this amount of memory had a negligible impact on the overall execution time (less than 1% increase).

5.3 Issues Related to Paging

If all the cells are running low on memory, there is no choice but to page data out to disk. In addition to providing the basic paging functionality, our algorithms had to solve three additional challenges: identifying actively used pages, handling memory pages shared by different virtual machines, and avoiding redundant paging.

Cellular Disco implements a second-chance FIFO queue to approximate LRU page replacement, similar to VMS [Levy and Lipman 1982]. Each virtual machine is assigned a resident set size that is dynamically trimmed when the system is running low on memory. Although any LRU approximation algorithm can find frequently used pages, it cannot separate the infrequently used pages into pages that contain active data and unallocated pages that contain garbage. Cellular Disco avoids having to write unallocated pages out to disk by nonintrusively monitoring the physical pages actually being used by the operating system. Annotations on the operating system's memory allocation and deallocation routines provide the required information to the virtual machine monitor.

A machine page can be shared by multiple virtual machines if the page is used in a shared memory region as described in Section 2.4, or as a result of a COW (Copy-On-Write) optimization. The sharing information is usually kept in memory in the control data structures for the actual machine page. However, this information cannot remain there once the page has been written out if the machine page is to be reused. In order to preserve the sharing, Cellular Disco writes the sharing information out to disk along with the data. The sharing information is stored on a contiguous sector following the paged data so that it can be written out using the same disk I/O request; this avoids the penalty of an additional disk seek.

Redundant paging is a problem that has plagued early virtual machine implementations [Seawright and MacKinnon 1979]. This problem can occur since there are two separate paging schemes in the system: one in Cellular Disco, the other in the operating systems running in the virtual machines. With these schemes making independent decisions, some pages may have to be written out to disk twice, or read in just to be paged back out. Cellular Disco avoids this inefficiency by trapping every read and write to the kernel's paging disk, identified by designating for every virtual machine a special disk that acts as the virtual paging disk. Figure 7 illustrates the problem and the way Cellular Disco avoids it. In both cases shown, the virtual machine kernel wishes to write a page to its paging disk that Cellular Disco has already paged out to its own paging disk. Without the paging disk, as shown in Case A, the kernel's pageout request appears to the monitor as a regular disk write of a page that has been paged out to Cellular Disco's paging disk. Therefore, Cellular Disco will first fault that page in from its paging disk, and then issue the write for the kernel's

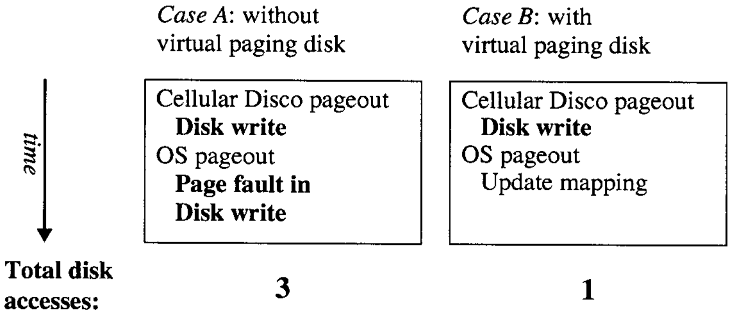

Fig. 7. Redundant paging.

paging disk. Case B shows the optimized version with the virtual paging disk. When the operating system issues a write to this disk, the monitor notices that it has already paged out the data, so it simply updates an internal data structure to make the sectors of the virtual paging disk point to the real sectors on Cellular Disco's paging disk. Any subsequent operating system read from the paging disk is satisfied by looking up the actual sectors in the indirection table and reading them from Cellular Disco's paging disk.

We measured the impact of the paging optimization by running the following microbenchmark, called stressMem. After allocating a very large chunk of memory, stressMem writes a unique integer on each page; it then loops through all the pages again, verifying that the value it reads is the same as what it wrote out originally. StressMem ran for 258 seconds when executing without the virtual paging disk optimization, but it took only 117 seconds with the optimization (a 55% improvement).

6. HARDWARE FAULT RECOVERY

Due to the tight coupling provided by shared-memory hardware, the effects of any single hardware fault in a multiprocessor can very quickly ripple through the entire system. Current commercial shared-memory multiprocessors are thus extremely likely to crash after the occurrence of any hardware fault. To resume operation on the remaining good resources after a fault, these machines require a hardware reset and a reboot of the operating system.

As shown in Teodosiu et al. [1997], it is possible to design multiprocessors that limit the impact of most faults to a small portion of the machine, called a hardware fault containment unit. Cellular Disco requires that the underlying hardware be able to recover itself with such a recovery mechanism. After detecting a hardware fault, the fault recovery support described in Teodosiu et al. [1997] diagnoses the system to determine which resources are still operational and reconfigures the machine in order to allow the resumption of normal operation on the remaining good resources. An important step in the reconfiguration process is to determine which cache lines have been lost as a result of the failure. Following a failure, cache lines can be either coherent (lines that were not affected by the fault) or incoherent (lines that have been lost because of the fault). Since the shared-memory system is unable to supply valid data for incoherent cache lines, any cache miss to these lines must be terminated by raising an exception.

After completing hardware recovery, the hardware informs Cellular Disco that recovery has taken place by posting an interrupt on all the good nodes. This interrupt will cause Cellular Disco to execute its own recovery sequence to determine the set of still-functioning cells and to decide which virtual machines can continue execution after the fault. This recovery process is similar to that done in Hive [Chapin et al. 1995], but our design is much simpler for two reasons: we did not have to deal with operating system data structures, and we can use shared-memory operations because cells can trust each other. Our simpler design results in a much faster recovery time.

In the first step of the Cellular Disco recovery sequence, all cells agree on a liveset (set of still-functioning nodes) that forms the basis of all subsequent recovery actions. While each cell can independently obtain the current liveset by reading hardware registers [Teodosiu et al. 1997], the possibility of multiple hardware recovery rounds resulting from back-to-back hardware faults requires the use of a standard n-round agreement protocol [Lynch 1996] to guarantee that all cells operate on a common liveset.

The agreed-upon liveset information is used in the second recovery step to “unwedge” the communication system, which needs to be functional for subsequent recovery actions. In this step, any pending RPCs or messages to failed cells are aborted; subsequent attempts to communicate with a failed cell will immediately return an error.

The final recovery step determines which virtual machines had essential dependencies on the failed cells and terminates those virtual machines. Memory dependencies are determined by scanning all machine memory pages and checking for incoherent cache lines; the hardware provides a mechanism to perform this check. Using the memmap data structure, bad machine memory pages are translated back to the physical memory pages that map to them, and then to the virtual machines owning those physical pages. A tunable recovery policy parameter determines whether a virtual machine that uses a bad memory page will be immediately terminated or will be allowed to continue running until it tries to access an incoherent cache line. I/O device dependencies are treated similarly to memory dependencies.

The experimental setup used throughout the rest of this paper could not be used for testing the Cellular Disco fault recovery support, since the necessary hardware fault containment support required by Cellular Disco is not implemented in the Origin 2000 multiprocessor, and since in the piggybacking solution of Section 3.1 the host operating system represents a single point of failure. Fortunately, Cellular Disco was originally designed to run on the FLASH multiprocessor [Kuskin et al. 1994], for which the

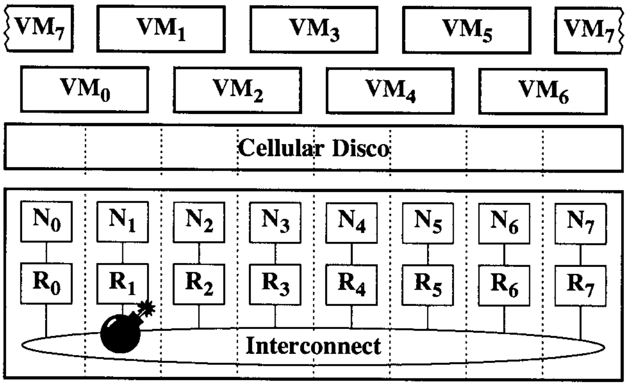

Fig. 8. Setup for fault injection experiments.

hardware fault containment support described in Teodosiu et al. [1997] was designed. When running on FLASH, Cellular Disco can fully exploit the machine's hardware fault containment capabilities. The main difference between FLASH and the Origin 2000 is the use in FLASH of a programmable node controller called MAGIC. Most of the hardware fault containment support in FLASH is implemented using MAGIC firmware.

We tested the hardware fault recovery support in Cellular Disco by using a simulation setup that allowed us to perform a large number of fault injection experiments. We did not use the FLASH hardware because the current FLASH prototype only has four nodes and because injecting multiple controlled faults is extremely difficult and time consuming on real hardware. The SimOS [Rosenblum et al. 1997] and FlashLite [Kuskin et al. 1994] simulators provide enough detail to accurately observe the behavior of the hardware fault containment support and of the system software after injecting any of a number of common hardware faults into the simulated FLASH system.

Figure 8 shows the setup used in our fault injection experiments. We simulated an 8-node FLASH system running Cellular Disco. The size of the Cellular Disco cells was chosen to be one node, the same as that of the FLASH hardware fault containment units. We ran 8 virtual machines, each with essential dependencies on two different cells. Each virtual machine executed a parallel compile of a subset of the GnuChess source files.

On the configuration shown in Figure 8 we performed the fault injection experiments described in Table III. After injecting a hardware fault, we allowed the FLASH hardware recovery and the Cellular Disco recovery to execute, and ran the surviving virtual machines until their workloads completed. We then checked the results of the workloads by comparing the checksums of the generated object files with the ones obtained from a reference run. An experiment was deemed successful if exactly one Cellular Disco cell and the two virtual machines with dependencies on that cell were lost after the fault, and if the surviving six virtual machines produced the correct results. Table III shows that the Cellular Disco hardware fault

Table III. Results of Fault Injection Experiments

Simulated Hardware Fault	Number of Experiments	Success Rate
Node power supply failure	250	100%
Router power supply failure	250	100%
Link cable or connector failure	250	100%
MAGIC firmware failure	250	100%

recovery support was 100% effective in 1000 experiments that covered router, interconnect link, node, and MAGIC firmware failures.

In order to evaluate the performance impact of a fault on the surviving virtual machines, we measured the recovery times in a number of additional experiments. Figure 9 shows how the recovery time varies with the number of nodes in the system and the amount of memory per node. The figure shows that the total recovery time is small (less than half a second) for all tested hardware configurations. While the recovery time only shows a modest increase with the number of nodes in the system, there is a steep increase with the amount of memory per node. For large memory configurations, most of the time is spent in two places. First, to determine the status of cache lines after a failure, the hardware fault containment support must scan all node coherence directories. Second, Cellular Disco uses MAGIC firmware support to determine which machine memory pages contain inaccessible or incoherent cache lines. Both of these operations involve expensive directory-scanning operations that are implemented using MAGIC firmware. The cost of these operations could be substantially reduced in a machine with a hardwired node controller.

7. COMPARISON TO OTHER APPROACHES

In the previous sections we have shown that Cellular Disco combines the features of both hardware partitioning and traditional shared-memory multiprocessors. In this section we compare the performance of our system against both hardware partitioning and traditional operating-system-centric approaches. The hardware-partitioning approach divides a large-scale machine into a set of small-scale machines, and a separate operating system is booted on each one, similar to a cluster of small machines with a fast interconnect. This approach is also similar to Cellular Disco without intercell resource sharing. In fact, because IRIX 6.2 does not run on the SGI Origin, we evaluated the performance of this approach using Cellular Disco without intercell sharing. We used IRIX 6.4 as the representative of operating-system-centric approaches.

Small applications that fit inside a single hardware partition run equally well on all three systems, except for the small virtualization overheads of Cellular Disco. Large resource-intensive applications that do not fit inside a single partition, however, can experience significant slowdown when running on a partitioned system due to the lack of resource sharing. In this

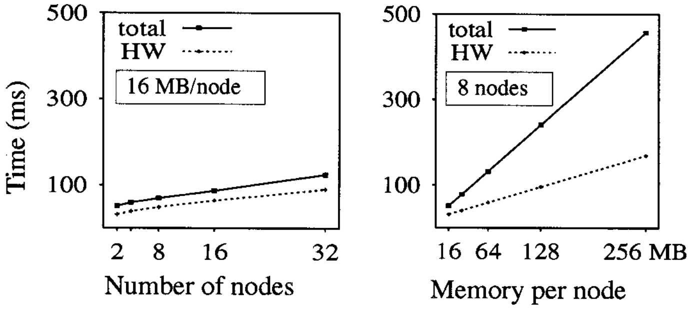

Fig. 9. Fault recovery times.

section we evaluate all three systems using such a resource-intensive workload to demonstrate the need for resource sharing.

For our comparison, we use a workload consisting of a mix of applications that resembles the way large-scale machines are used in practice: we combine an 8-process Database workload with a 16-process Raytrace run. By dividing the 32-processor Origin system into 4 cells (each with 8 processors), we obtain a configuration in which there is neither enough memory in any single cell to satisfy Database, nor enough CPUs in any cell to satisfy Raytrace. Because the hardware-partitioning approach cannot automatically balance the load, we explicitly placed the two applications on different partitions. In all three cases, we started both applications at the same time, and measured the time it took them to finish, along with the overall CPU utilization. Table IV summarizes the results of our experimental comparison. As expected, the performance of our virtual clusters solution is very close to that of the operating-system-centric approach, as both applications are able to access as many resources as they need. Also, as expected, the hardware-partitioning approach suffers serious performance degradation due to the lack of resource sharing.

The hardware partitioning and cluster approaches typically avoid such serious problems by allocating enough resources in each partition to meet the expected peak demand; for example, the database partition would have been allocated with more memory and the raytrace partition with more processors. However, during normal operation this configuration wastes resources, and prevents efficient resource utilization because a raytrace workload will not perform well on the partition configured for databases, and similarly, a database workload will not perform well on the partition configured for raytrace.

8. RELATED WORK

In this section we compare Cellular Disco to other projects that have some similarities to our work: virtual machines, hardware partitioning, operating-system-based approaches, fault containment, and resource load balancing.

Table IV. Performance Comparison to Other Approaches

Approach	Raytrace	Database	CPU Utilization
Operating system	216 s	231 s	55%
Virtual cluster	221 s	229 s	58%
Hardware partitioning	434 s	325 s	31%

8.1 Virtual Machines

Virtual machines are not a new idea: numerous research projects in the 1970's [Goldberg 1974], as well as commercial product offerings [Creasy 1981; Seawright and MacKinnon 1979] attest to the popularity of this concept in its heyday. The VAX VMM Security Kernel [Karger et al. 1991] used virtual machines to build a compatible secure system at a low development cost. While Cellular Disco shares some of the fundamental framework and techniques of these virtual machine monitors, it is quite different in that it adapts the virtual machine concept to address new challenges posed by modern scalable shared-memory servers.

Disco [Bugnion et al. 1997] first proposed using virtual machines to provide scalability and to hide some of the characteristics of the underlying hardware from NUMA-unaware operating systems. Compared to Disco, Cellular Disco provides a complete solution for large-scale machines by extending the Disco approach with the following novel aspects: the use of a virtual machine monitor for supporting hardware fault containment; the development of both NUMA- and fault-containment-aware scalable resource balancing and overcommitment policies; and the development of mechanisms to support those policies. We have also evaluated our approach on real hardware using long-running realistic workloads that more closely resemble the way large machines are currently used.

8.2 Hardware-Centric Approaches

Hardware partitioning has been proposed as a way to solve the system software issues for large-scale shared-memory machines. Some of the systems that support partitioning are Sequent's Application Region Manager [Sequent Computer Systems 2000], Sun Microsystems' Dynamic System Domains [Sun Microsystems 2000], and Unisys' Cellular MultiProcessing (CMP) architecture [Unisys 2000]. The benefits of this approach are that it only requires very small operating system changes, and that it provides limited fault isolation between partitions [Sun Microsystems 2000; Unisys 2000]. The major drawback of partitioning is that it lacks resource sharing, effectively turning a large and expensive machine into a cluster of smaller systems that happen to share a fast network. As shown in Section 7, the lack of resource sharing can lead to serious performance degradation.

To alleviate the resource-sharing problems of static partitioning, dynamic partitioning schemes have been proposed that allow a limited redistribution of resources (CPUs and memory) across partitions. Unfortunately, repartitioning is usually a very heavyweight operation requiring extensive hardware and operating system support. An additional drawback is that even though whole nodes can be dynamically reassigned to a different partition, the resources within a node cannot be multiplexed at a fine granularity between two partitions.

8.3 Software-Centric Approaches

Attempts to provide the support for large-scale multiprocessors in the operating system can be divided into two strategies: tuning an existing SMP operating system to make it scale to tens or hundreds of processors, and developing new operating systems with better scalability characteristics.

The advantages of adapting an existing operating system are backward compatibility and the benefit of an existing sizable code base, as illustrated by SGI's IRIX 6.4 and IRIX 6.5 operating systems. Unfortunately, such an overhaul usually requires a significant software development effort. Furthermore, adding support for fault containment is a daunting task in practice, since the base operating system is inherently vulnerable to faults.

Compaq's Open VMS Galaxy [Compaq 2000] reduces some of this development cost by partitioning the resources in software and running multiple instances of the kernel. The various instances cooperatively share and manage resources. Although this approach does not require that the operating system scale to a large number of resources, it does require that the operating system be able to handle dynamically varying number of resources. Furthermore, the Galaxy architecture does not provide fault containment.

New operating systems have been proposed to address scalability requirements (Tornado [Gamsa et al. 1999] and K42 [IBM 2000]) and fault containment (Hive [Chapin et al. 1995]). While these approaches tackle the problem at the basic level, they require a very significant development time and cost before reaching commercial maturity. Compared to these approaches, Cellular Disco is about two orders of magnitude simpler, while providing almost the same performance.

8.4 Fault Containment

While a considerable amount of work has been done on fault tolerance, this technique does not seem to be very attractive for large-scale shared-memory machines, due to the increase in cost and to the fact that it does not defend well against operating system failures. An alternative approach that has been proposed is fault containment, a design technique that can limit the impact of a fault to a small fraction of the system. Fault containment support in the operating system has been explored in the Hive project [Chapin et al. 1995], while the necessary hardware and firmware support has been implemented in the FLASH multiprocessor [Kuskin et al. 1994]. Cellular Disco requires the presence of hardware fault containment support such as that described in Teodosiu et al. [1997], and is thus complementary. Hive and Cellular Disco are two attempts to provide fault containment support in the system software; the main advantage of Cellular Disco is its extreme simplicity when compared to Hive. Our approach is the first practical demonstration that end-to-end hardware fault containment can be provided at a realistic cost in terms of implementation effort. Cellular Disco also shows that if the basic system software layer can be trusted, fault containment does not add any performance overhead.

A production version of Cellular Disco would have to include many device drivers, which may contain some bugs, as they cannot be expected to be as thoroughly tested as the core virtual machine monitor. This concern could be addressed by executing the device drivers in a less privileged level to ensure they cannot harm the rest of the system; a similar approach is used by microkernels such as Mach [Rashid et al. 1988].

8.5 Load Balancing

CPU and memory load balancing have been studied extensively in the context of networks of workstations, but not on single shared-memory systems. Traditional approaches to process migration [Milojicic et al. 1996] that require support in the operating system are too complex and fragile, and very few have made it into the commercial world so far. Cellular Disco provides a much simpler approach to migration that does not require any support in the operating system, while offering the flexibility of migrating at the granularity of individual CPUs or memory pages.

Research projects such as GMS [Feeley et al. 1995] have investigated using remote memory in the context of clusters of machines, where remote memory is used as a fast cache for VM pages and file system buffers. Cellular Disco can directly use the hardware support for shared memory, thus allowing substantially more flexibility.

9. CONCLUSION

With a size often exceeding a few million lines of code, current commercial operating systems have grown too large to adapt quickly to the new features that have been introduced in hardware. Off-the-shelf operating systems currently suffer from poor scalability, lack of fault containment, and poor resource management for large systems. This lack of good support for large-scale shared-memory multiprocessors stems from the tremendous difficulty of adapting the system software to the new hardware requirements.

Instead of modifying the operating system, our approach inserts a software layer between the hardware and the operating system. By applying an old idea in a new context, we show that our virtual machine monitor (called Cellular Disco) is able to supplement the functionality provided by the operating system and to provide new features. In this paper, we argue that Cellular Disco is a viable approach for providing scalability, scalable resource management, and fault containment for large-scale shared-memory systems at only a small fraction of the development cost required for changing the operating system. Cellular Disco effectively turns those large machines into "virtual clusters" by combining the benefits of clusters and those of shared-memory systems.

Our prototype implementation of Cellular Disco on a 32-processor SGI Origin 2000 system shows that the virtualization overhead can be kept below 10% for many practical workloads, while providing effective resource management and fault containment. Cellular Disco is the first demonstration that end-to-end fault containment can be achieved in practice with a reasonable implementation effort. Although the results presented in this paper are based on virtualizing the MIPS processor architecture and on running the IRIX operating system, our approach can be extended to other processor architectures and operating systems. A straightforward extension of Cellular Disco could support the simultaneous execution on a scalable machine of several operating systems, such as a combination of Windows NT, Linux, and UNIX.

Some of the remaining problems that have been left open by our work so far include efficient virtualization of low-latency I/O devices (such as fast network interfaces), system management issues, and checkpointing and cloning of whole virtual machines.

ACKNOWLEDGMENTS

We would like to thank SGI for kindly providing us access to a 32-processor Origin 2000 machine for our experiments, and to the IRIX 5.3, IRIX 6.2 and IRIX 6.4 source code. The experiments in this paper would not have been possible without the invaluable help we received from John Keen, Casey Leedom, and Simon Patience.

The FLASH and Hive teams built most of the infrastructure needed for this paper, and provided an incredibly stimulating environment for this work. Our special thanks go to the Disco, SimOS, and FlashLite developers whose work has enabled the development of Cellular Disco and the fault injection experiments presented in the paper.

REFERENCES

Barton, J. M. and Bitar, N. 1995. A scalable multi-discipline, multiple-processor scheduling framework for IRIX. *Lecture Notes in Computer Science 949*, 45–69.

Bugnion, E., Devine, S., Govil, K., and Rosenblum, M. 1997. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems 15*, 4 (November), 412–447.

Chapin, J., Rosenblum, M., Devine, S., Lahiri, T., Teodosiu, D., and Gupta, A. 1995. Hive: Fault containment for shared-memory multiprocessors. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)* (December 1995), pp. 12–25.

Compaq. 2000. OpenVMS Galaxy. Available: http://www.openvms.compaq.com/availability/galaxy.html.

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

Feeley, M. J., Morgan, W. E., Pighin, F. H., Karlin, A. R., Levy, H. M., and Thekkath, C. 1995. Implementing global memory management in a workstation cluster. In *Proceedings* *of the 15th ACM Symposium on Operating Systems Principles (SOSP)* (December 1995), pp. 201–212.

GALLES, M. AND WILLIAMS, E. 1994. Performance optimizations, implementation, and verification of the SGI Challenge multiprocessor. In T. N. Mudge and B. D. Shriver Eds., *Proceedings of the 27th Hawaii International Conference on System Sciences. Volume 1: Architecture* (Los Alamitos, CA, USA, January 1994), pp. 134–143. IEEE Computer Society Press.

GAMSA, B., KRIEGER, O., APPAVOO, J., AND STUMM, M. 1999. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)* (February 1999), pp. 87–100.

GOLDBERG, R. P. 1974. Survey of virtual machine research. *IEEE Computer Magazine 7*, 4 (June), 34–45.

IBM. 2000. The K42 Project. Available: http://www.research.ibm.com/K42/index.html.

KANE, G. AND HEINRICH, J. 1992. *MIPS RISC Architecture*. Prentice Hall.

KARGER, P. A., ZURKO, M. E., BONIN, D. W., MASON, A. H., AND KAHN, C. E. 1991. A retrospective on the VAX VMM security kernel. *IEEE Transactions on Software Engineering 17*, 11 (November), 1147–1165. Special Section on Security and Privacy.

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 (ISCA)* (April 1994), pp. 302–313.

LAUDON, J. AND LENOSKI, D. 1997. The SGI Origin: A ccNUMA highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA)* (June 1997), pp. 241–251.

LEVY, H. M. AND LIPMAN, P. H. 1982. Virtual memory management in the VAX/VMS operating system. *Computer 15*, 3 (March), 35–41.

LYNCH, N. A. 1996. *Distributed Algorithms*. Morgan Kaufmann series in data management systems. Morgan Kaufmann Publishers, Los Altos, CA.

MILOJICIC, D. S., DOUGLIS, F., PAINDAVEINE, Y., WHEELER, R., AND ZHOU, S. 1996. Process migration. Technical report (December), TOG Research Institute.

RASHID, R., TEVANIAN, A., YOUNG, M., GOLUB, D., BARON, R., BALCK, D., BOLOSKY, W. J., AND CHEW, J. 1988. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computer 37*, 8 (August), 896–908.

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 (January), 78–103.

SEAWRIGHT, L. H. AND MACKINNON, R. A. 1979. VM/370: a study of multiplicity and usefulness. *IBM Systems Journal 18*, 1, 4–17.

SEQUENT COMPUTER SYSTEMS. 2000. Application Region Manager. Available: http://www.sequent.com/dcsolutions/agile_wp1.html.

SGI. 2000. IRIX6.5. Available: http://www.sgi.com/software/irix6.5.

SPEC. 2000. SPECweb96 Benchmark. Available: http://www.spec.org/osg/web96.

SUN MICROSYSTEMS. 2000. Sun Enterprise 10000: Dynamic System Domains. Available: http://www.sun.com/servers/highend/10000/Tour/domains.html.

TEODOSIU, D., BAXTER, J., GOVIL, K., CHAPIN, J., ROSENBLUM, M., AND HOROWITZ, M. 1997. Hardware fault containment in scalable shared-memory multiprocessors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA)*, Volume 25, 2 of *Computer Architecture News* (New York, June 1997), pp. 73–84. ACM Press.

TRANSACTION PROCESSING PERFORMANCE COUNCIL. 1997. *TPC Benchmark D (Decision Support) Standard Specification*. TPC, San Jose, CA.

UNISYS. 2000. Cellular MultiProcessing Architecture. Available: http://www.unisys.com/marketplace/ent/cmp.html.

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 (ASPLOS) (October 1996), pp. 279–289.

VMWARE. 2000. VMware Virtual Platform. Available: http://www.vmware.com/products/virtualplatform.html.

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* (Santa Margherita Ligure, Italy, June 1995), pp. 24–36. ACM SIGARCH and IEEE Computer Society TCCA.

Received January 2000; accepted May 2000