Performance Isolation: Sharing and Isolation in Shared-Memory Multiprocessors

Ben Verghese[1], Anoop Gupta[2,3], and Mendel Rosenblum[2]

[1]Western Research Laboratory, Compaq Computer Corporation, Palo Alto, CA 94301.
[2]Computer Systems Laboratory, Stanford University, Stanford, CA 94305.
[3]Microsoft Corporation, Redmond, WA 98052.

verghese@pa.dec.com, ag@pepper.stanford.edu, mendel@crissy.stanford.edu

Abstract

Shared-memory multiprocessors (SMPs) are being extensively used as general-purpose servers. The tight coupling of multiple processors, memory, and I/O provides enormous computing power in a single system, and enables the efficient sharing of these resources.

The operating systems for these machines (UNIX or Windows NT) provide very few controls for sharing the resources of the system among the active tasks or users. This unconstrained sharing model is a serious limitation for a server because the load placed by one user can adversely affect other users' performance in an unpredictable manner. We show that this lack of isolation is caused by the resource allocation scheme (or lack thereof) carried over from single-user workstations. Multi-user multiprocessor systems require more sophisticated resource management, and we show how the proposed "performance isolation" scheme can address the current weaknesses of these systems. We have implemented performance isolation in the Silicon Graphics IRIX operating system for three important system resources: CPU time, memory, and disk bandwidth. Running a number of workloads we show that our proposed scheme is successful at providing workstation-like isolation under heavy load, SMP-like latency under light load, and SMP-like throughput in all cases.

1. Introduction

The emergence of the client-server computing paradigm has generated new interest in servers. Shared-memory multiprocessors are being widely used as these servers because they aggregate a large collection of computing resource — multiple processors, large amounts of memory, and I/O — in a tightly-coupled system. These resources can be efficiently utilized through flexible and automatic reallocation to accommodate the disparate resource requirements of applications in compute-server workloads. However, a compute server often has to serve many masters. Unrelated jobs belonging to various groupings, such as different users or projects need to co-exist on the system. Such an environment requires sophisticated controls in the operating system to carefully allocate resources to different tasks, making trade-offs where required to get the benefits of good isolation and good throughput.

Current operating systems have little support for controlling the allocation of resources to groups of processes, or for providing fairness by any abstraction other than individual processes. With unconstrained sharing and contention for resources, jobs belonging to one group can severely impact the performance of jobs belonging to other groups in an unpredictable manner. The existing controls for fairness regulate only access to the CPU by a process. There are few controls for allocating other resources that can affect performance, such as memory and I/O bandwidth. Memory allocation is usually not done explicitly (static quotas per process at best), and allocation of I/O bandwidth is nonexistent. Resource management needs to be done to provide fairness to higher-level logical entities, such as individual users, a group of users, or a group of processes that comprise a task, not just between competing processes. A multiprocessor is often a shared resource with implicit or explicit contracts between users or tasks on how to share the machine. For example, project A owns a third of the machine and project B owns two thirds. Such contracts would be very difficult to implement in the absence of explicit mechanisms and policies for resource management.[1]

Current shared-memory systems represent one end of the spectrum for clustering computing resources. These systems seem to favor overall throughput at the cost of response time for individual jobs. This centralized model of computation is to be contrasted with the distributed network of workstations model (NOW) [ACP+94] that has implicit isolation because jobs are run on separate workstations. These systems provide good response time for an individual's jobs at the cost of overall throughput. The workstation solution, being a loosely coupled system, has a much higher overhead for sharing resources. Therefore, fine-grain sharing is difficult, and idle resources can only be allocated at a very coarse granularity.

A possible option to provide isolation is to enforce fixed quotas per user or task for the different resources. However,

ASPLOS VIII 10/98 CA, USA

[1] Although we focus on multiprocessors here, we believe that sophisticated resource management will become increasingly important to uniprocessors as well. How resources are prioritized/segmented between top-level applications, their background analysis and prefetching tasks, and other time-sensitive tasks raises similar resource management issues.

this method would not allow the sharing of idle resources, and significantly reduces the throughput and response time seen on these systems under light-load conditions. Another possibility is the equivalent of real-time systems with applications requesting guaranteed resources or deadlines, and the system guaranteeing this through admission control based on the availability of resources [Hyd94][Mer97]. However, this is too constrained a system for a general-purpose server that is to run a large number of unmodified applications. Finally, there is the highly sophisticated, but complex, functionality found in mainframe operating systems (OS390 WLM) that is able to accommodate different types of application goals, and automatically manage resource allocation to achieve these goals [AEE+97].

We seek a simple solution that provides isolation without compromising throughput. This paper presents "**Performance Isolation**", a resource management scheme for shared-memory multiprocessors. This scheme isolates processes belonging to a logical entity, such as a user, from the resource requirements of others, and also preserves the inherent sharing ability of these machines. The proposed scheme has been implemented on the Silicon Graphics IRIX operating system for three important system resources: CPU time, memory, and disk bandwidth. Running a number of workloads on this kernel using SimOS, we show that our proposed scheme is successful at providing isolation for tasks and efficient sharing of idle resources.

The outline of the rest of the paper is as follows. Section 2. provides a framework for performance isolation. In Section 3., we describe the implementation of performance isolation in the operating system, and discuss the various implementation issues that arise. In Section 4., we run different workloads to demonstrate that performance isolation works. Section 5. discusses related work, and Section 6. presents our conclusions.

2. A Framework for Performance Isolation

We now build a framework for the performance isolation model. We will first describe the SPU kernel abstraction that is the key component for performance isolation. We then discuss the two main issues: providing isolation between SPUs and policies for sharing resources between SPUs.

2.1 The Performance Isolation Model

The performance isolation model for shared-memory multiprocessors essentially partitions the computational resources of the multiprocessor into multiple flexible units based on a previously configured contract for sharing the machine. From a performance and resource allocation viewpoint the multiprocessor now looks like a collection of smaller machines.

At the heart of the model is a kernel abstraction called the *Software Performance Unit* (SPU), and each of these logical smaller machines is associated with an SPU. This is not a static permanent allocation of resources to SPUs as will become clear soon. SPUs can be created and destroyed dynamically, or could be suspended when they have no active processes and awakened at a later time.

The SPU abstraction has three parts:

1. The first is a criterion for assigning processes to an SPU. This decides which processes have access to the resources of the SPU. The performance of a process will be isolated from the resource requirements of any process that is not associated with its SPU. However, the SPU does not provide isolation between processes that are associated with the same SPU. The desired basis for the grouping of processes can vary greatly, and is dependent on the environment of the particular machine and the isolation goals. Some common possibilities are: Individual processes, groups of processes representing a task, processes belonging to a user, or processes belonging to a group of users.
2. The second is the specification of the share of system resources assigned to the SPU. We are primarily interested in the computing resources that directly affect user performance: CPU time, memory, and I/O bandwidth (disk, network, etc.). However, it would be possible to incorporate other resources if required. There are many possible ways of partitioning resources between SPUs, such as a fixed fraction of the machine, or a specified amount of each resource.
3. The third is a sharing policy. Resources can be lent to other SPUs, and revoked when needed again by the loaning SPU. The sharing policy decides when and to whom resources belonging to an SPU will be allocated when these resources are idle. There are many possible types of sharing policies, and the following is a nonexhaustive list:
   - Never give up any resources. This will approximate the case of each SPU being an entirely separate machine with its share of resources, or the machine being divided up with fixed quotas; there is no sharing.
   - Share all resources with everyone all the time, without consideration for whether the resources are idle or not. This approximates the behavior of current SMP systems.
   - A more interesting possibility is to share only idle resources with all or a subset of the SPUs that lack sufficient resources and could use the idle ones.

The SPU abstraction is quite versatile and should not be confused with a naive fixed quota policy that sets hard limits on resource usage by a single process or a group of processes. The sharing policy of the SPU abstraction can be set per SPU to customize the behavior of the system as seen by the users. The performance isolation model, which we discuss in the rest of this thesis, will use the SPU abstraction with a specific sharing policy to achieve its goals. An SPU will share idle resources with any SPU that needs the idle resources. With this sharing policy, the performance isolation model should achieve the following two performance goals:

1. **Isolation**: If the resource requirements of an SPU are less than its allocated fraction of the machine, the SPU should see no degradation in performance, regardless of the load placed on the system by others.
2. **Sharing**: If the resource requirements of an SPU exceeds its configured share of resources, the SPU should be able to easily utilize idle resources to improve its response time and throughput.

There are two parts to the solution for implementing the SPU abstraction for providing performance isolation, corresponding to the two goals presented above.

2.2 Providing Isolation

A key issue in providing performance isolation is that current SMP systems do not have the appropriate metrics to track short-term usage of all resources by processes or groups of processes, and cannot limit the usage of these resource by a specific process. In order to provide isolation between SPUs two new aspects of functionality are needed in the kernel. First, the utilization of resources by individual SPUs needed to be tracked. For example, the kernel needs to maintain a page-use count per SPU, and increment it every time a memory page is allocated to the SPU. Second, mechanisms are needed to limit the usage of resources by an SPU to allocated levels. For example, currently in the SGI IRIX operating system a request for a page of memory will fail only if there is no free memory in the system. With isolation a page request from a process will be denied if the SPU that owns the process has used its allocation of pages.

A particular problem area in providing isolation is accounting for resources that are actually shared by multiple SPUs, or that do not belong to any specific SPU. Examples of the former are pages of memory accessed simultaneously by multiple SPUs such as shared library pages or code, and delayed disk write requests that often contain dirty pages from multiple processes and multiple SPUs. Examples of the latter are kernel processes, such as the pager and swapper daemons and pages used for kernel code and data.

For this problem our current strategy has been to choose the simplest solutions that seem reasonable. More sophisticated solutions may easily be considered in the future, if we encounter instances where these proposed solutions clearly do not work. To address the above problem we introduce two default SPUs in the system: *kernel,* for kernel processes and memory; and *shared* for tracking resources used by multiple SPUs. The cost of memory pages that are referenced by multiple SPUs is counted in the shared SPU, and not explicitly allocated to any of the user SPUs. Memory pages other than those used by the kernel and shared SPUs are divided among user SPUs. Therefore, the cost of shared and kernel pages is effectively shared by all user SPUs. The cost of shared pages could be assigned more precisely if necessary, but this would incur a larger overhead. Shared disk writes get scheduled for service in the shared SPU. The cost of individual non-shared pages in these write requests is allocated to the appropriate user SPUs. The kernel SPU has unrestricted access to all resources.

2.3 Policies for Sharing

The second part of the performance isolation model is the careful sharing of idle resources between SPUs, based on the sharing policy feature of the SPU abstraction. Conceptually, each SPU maintains three resource levels to implement resource sharing. The first level is the amount of resources that the SPU is **entitled** to initially. This level is decided by the division of system resources based on the sharing contract for the system. The second level is the amount of resources that the SPU is **allowed** to use currently. The third level is the amount of resources currently **used** by the SPU.

Sharing is implemented by changing the allowed level for SPUs based on resource requirements and availability. In a system under load where all SPUs are utilizing their share of the resources, all three levels will be at about the same value for the SPUs. No sharing will happen. At some point one or more SPUs may go idle or be under utilized. Their **used** level will now be much less than their **entitled** level, indicating idle resources. The sharing policy can now transfer some of these idle resources from the under-utilized SPUs to the others by increasing the value of the **allowed** level for the latter. When the SPUs want their resources back, the sharing policy will lower the **allowed** level of the borrowing SPUs, potentially to the **entitled** level.

The key factor in making the decision to transfer resources is the **revocation cost** for these resources when they are needed again by the loaning SPU. The isolation of performance of an SPU can be adversely affected if the sharing policy is not careful when transferring idle resources. If the revocation cost were zero, then transferring all the resources would not be a problem as they could be instantly revoked when needed. However, most resources have a non-trivial revocation cost, and this cost plays a part in deciding when resources are transferred and how much of the idle resources are transferred. When making sharing decisions, the policy module needs to ensure that the cost of revocation does not adversely affect the performance of the loaning SPU and break isolation.

3. Implementing Performance Isolation

In the previous section we discussed a framework for implementing performance isolation in terms of providing isolation and sharing between SPUs. We now describe the details of our implementation of performance isolation in the IRIX5.3 kernel from Silicon Graphics. Most of the ideas in this implementation are not specific to IRIX, and would apply to other operating systems as well. The system resources included in the implementation are: CPU time, memory, and disk bandwidth (as an example of I/O bandwidth). Though we do not implement performance isolation for network bandwidth, the techniques we describe would apply to it as well. Our implementation is based on the assumption that all resources are to be divided equally among all the active SPUs, though it will be clear from the implementation that unequal shares can easily be supported.

For each resource we describe the metrics used to count usage, the mechanisms put in place to provide isolation, how these mechanisms differ from the ones currently in IRIX, and how the sharing policy enables sharing by reallocating idle resources. For our implementation we picked reasonable policies and mechanisms that allow us to clearly demonstrate the effectiveness of performance isolation. Other mechanisms are also possible for each of

the resources, and we will discuss them with related work in Section 5..

3.1 CPU Time

In IRIX, CPU time is allocated in time slices to processes — 30ms unless the process blocks before that for I/O. A priority-based scheduling scheme is used in which the priority of a process drops as it uses CPU time. A CPU normally picks the runnable process with the highest priority when scheduling a new process. This scheme maintains fairness for CPU time at a process level.

Isolation requires a mechanism to provide fairness at the SPU level, which usually includes more than one process. On a multiprocessor, CPU time can be allocated either through time multiplexing or space partitioning of the CPUs. We chose a hybrid approach[1]. First, each SPU is allocated an integral number of CPUs using space partitioning, depending on its entitlement. If in the division, fractions of CPUs need to be allocated to SPUs, then time partitioning is used for the remaining CPUs with the share of time allocated to an SPU corresponding to the fraction of the CPU. The SPU to which a CPU is allocated is its home SPU. Kernel processes can run on any CPU. To provide isolation the normal priority-based scheduling behavior is modified by having CPUs select processes only from their home SPUs when scheduling, thus ensuring that an SPU will get its share of CPU resources, regardless of the load on the system. Between processes of the same SPU, the standard IRIX priority scheduling disciplines apply.

Sharing is implemented by relaxing the SPU ID restriction when a processor becomes idle. If an SPU is lightly loaded, one or more processors belonging to this SPU may be idle. If a processor cannot find a process from its home SPU, it is allowed to consider processes from other SPUs. Currently, the process with the highest priority is chosen. As a result, the SPU getting the idle processor is not explicitly chosen, but the process with the highest priority is likely to be one from a relatively heavily-loaded SPU. An SPU could be explicitly picked if the home SPU's sharing policy indicated a preference.

Processors that have been loaned to SPUs are tracked. If a process from the home SPU now becomes runnable, and there are no allocated processors in the home SPU available to run this process, then the processor loan is revoked. In our policy, the revocation of the CPU happens either at the next clock tick interrupt (every 10 milliseconds), or when the process voluntarily enters the kernel. Therefore the maximum revocation latency for a CPU is 10 milliseconds. Another possibility would be to send an inter-processor interrupt (IPI) to get the processor back sooner. This might be needed to provide response time performance isolation guarantees to interactive processes.

There are other hidden costs to reallocating CPUs, such as cache pollution. A more sophisticated implementation of the sharing policy could try to reduce these costs by preventing frequent reallocation of CPUs for sharing, if the algorithm detects that the allocation is being revoked frequently.

3.2 Memory

The IRIX5.3 kernel has very few controls for memory allocation. It has a configurable limit to the total virtual memory a process can allocate. It also tries to place a fuzzy limit on the size of actual physical memory that a process can use. The problem is that these limits are per-process, and cannot provide the strict isolation that our model requires. Being essentially fixed quotas per-process, they may actually inhibit sharing of idle resources in the system.

Isolation and sharing for physical memory closely follows the method outlined in Section 2.3, keeping three counts of pages for each SPU — entitled, allowed and used. The page allocation function in the kernel is augmented to record the SPU ID of the process requesting the page, and to keep a count of the pages used by each SPU. In addition to regular code and data pages, SPU memory usage also includes pages used indirectly in the kernel on behalf of an SPU, such as the file buffer cache and file meta-data. Memory pages are conceptually space-partitioned among the SPUs, and the *entitled* count represents the initial share of memory for an SPU. Isolation between SPUs is enforced by not allowing an SPU to use more pages than the *allowed* limit. Corresponding changes are made to the paging and swapping functions to make them aware of SPUs and per-SPU memory limits

Sharing of idle memory is implemented by changing the *allowed* limit for SPUs. The SPU page usage counts are checked periodically to find SPUs with idle pages and SPUs that are under memory pressure. The sharing policy redistributes the excess pages in the system to the SPUs that are low on memory by increasing their allowed limits. The memory re-allocation is temporary, and can be reset if the memory situation in the lending or borrowing SPUs changes.

Excess pages are calculated as the total idle pages in the system less a small number of pages that are kept free. The small number of free pages is called the **Reserve Threshold**. The Reserve Threshold is needed to hide the revocation cost for memory, which is the time to reclaim any pages that have been lent to other SPUs. The revocation cost for pages of memory can be high, especially if they are dirty, because the dirty data will need to be written to disk before the page can be given back. The Reserve Threshold reduces the chance of a loaning SPU incorrectly being denied a page temporarily. The Reserve Threshold is

[1] Our choice of a hybrid scheduling policy, favoring space partitioning, fits well with our model of partitioning the machine, based on the assumption that there will be fewer active SPUs than CPUs. If this assumption does not hold, a more explicit time-partitioning policy may be appropriate. Also, parallel applications that use a space partitioning policy [ABL+91][Teo72] can be easily accommodated in our current scheme. Accommodating gang-scheduled [Ous82] parallel applications would require some modifications.

configurable, and we chose 8% of the total memory. This is the value that IRIX uses to decide if it is running low on memory.

A particular problem is tracking and accounting for pages that might be accessed by multiple SPUs, as mentioned in Section 2.2. When a page is first accessed, it is marked with the SPU ID of the accessing process. On a subsequent access by a different SPU before the page is freed, the page will be marked as a shared page (SPU ID of the shared SPU). The SPU ID for the page is reset when the page is finally freed. The cost of these shared pages is assigned to the shared SPU. Similarly the cost of pages used by the kernel is assigned to the kernel SPU. Only the remaining pages are actually divided among the SPUs based on their entitled share of memory. The allocation of pages to SPUs is periodically updated to account for changes in the usage of the shared and kernel SPUs.

3.3 Disk Bandwidth

IRIX5.3 schedules disk requests based only on the current head position of the disk using the standard C-SCAN algorithm [Teo72]. In the C-SCAN algorithm the outstanding disk requests are sorted by block number and serviced in order as the disk head sweeps from the first to the last sector on the disk. When the head reaches the request closest to the end of the disk, it then goes back to the beginning and starts again. This technique reduces the disk-head seek component of latency and prevents starvation. The process requesting the disk operation is not a factor in the algorithm, and there is total lack of isolation between SPUs. The sectors of a single file are often laid out contiguously on the disk. Therefore a read or write to a large file (e.g. a core dump) could monopolize the disk, causing all requests from one SPU to a file to be serviced before requests from other SPUs are scheduled.

To provide isolation we need to account for the disk bandwidth used by SPUs, and incorporate this information into the decision process for scheduling requests for the disk. We encountered a few difficulties in providing isolation for disk bandwidth. First, disk requests have variable sizes (one or more sectors), and breaking up requests into single sector operations would be inefficient. This implies that the granularity of allocation of bandwidth to SPUs will be in variable-size chunks. Therefore it is not enough to just count requests, rather the size of the request needs to be accounted for. Our metric for disk bandwidth is sectors transferred per second.

Second, the writes to disks are mostly done by system daemons that are flushing file-buffer-cache data or page-frame data. Therefore, these write requests contain pages belonging to multiple SPUs. Our implementation schedules these shared write requests as part of the shared SPU, which is given the lowest priority. Once the shared write request is done, the individual pages are charged to the appropriate user SPUs.

Third, disk bandwidth is a rate, and as such measuring the instantaneous rate is not possible. Therefore it is approximated by counting the total sectors transferred and decaying this count periodically. The decay period is configurable, and we currently decay the count by half every 500 milliseconds. A finer grain decay of the count would better approximate an instantaneous rate, but would have a higher overhead to maintain. This count of sectors transferred represents the bandwidth used by each SPU, and is kept for each disk.

Disk requests can incur a considerable latency for the disk head to seek to the appropriate spot on the disk. Implementing strict isolation requires a round-robin-type scheduling of requests by SPU based on bandwidth shares of each of the SPUs. However, completely ignoring the current disk-head position would result in poor throughput because of excessive delays caused by the extra seek time (see results in Section 4.5). Therefore, performance isolation employs a compromise that incorporates both disk-head position and a fairness criteria when making a scheduling decision.

In our policy, disk requests are scheduled based on the head position as long as all SPUs with active disk requests satisfy the fairness criteria. A SPU fails the fairness criteria if its bandwidth usage relative to its bandwidth share (current count of sectors/bandwidth share) exceeds the average value of all SPUs by a threshold (the **BW difference threshold**). Once an SPU fails the fairness criteria it is denied access to the disk until there are no more queued requests, or it once again passes the fairness criteria because other SPUs get their share of disk bandwidth. The fairness criteria is checked after each disk request. The choice of the BW difference threshold allows a trade-off. Smaller values imply better isolation, with a choice of zero resulting in round-robin scheduling. Larger values imply smaller seek times, and a very large value results in the normal disk-head-position scheduling.

Sharing happens naturally because an SPU cannot fail the fairness criterion if no other SPU has active requests. The revocation cost for the disk bandwidth resource is the time to finish any currently outstanding request, and for the disk head to scan to the desired position. Therefore, if a disk is shared then an SPU with high disk utilization can affect the performance of another SPU using the same disk. However, we will show that performance isolation can provide fairness and considerably reduce the impact of such shared access.

3.4 Shared Kernel Resources

In addition to the physical resources discussed above, there are shared kernel structures that are accessed from multiple processors and must be considered in the implementation of the performance isolation model. Additional stall time and contention for spinlocks and semaphores protecting these resources is a potential source of problems. In addition to straight contention, a high load SPU starved of resources and holding an important semaphore could block a process from a light load SPU. This could affect the ability of the kernel to provide isolation between SPUs. This problem is

Work load	System Parameters	Application	SPU Configuration
Pmake8	8 CPUs, 44Mbytes mem separate fast disks	Multiple Pmake jobs (two parallel compiles each)	Balanced: 8 SPUs (1 job) Unbalanced: 4 SPUs (1 job), 4 SPUs (2 jobs)
CPU Isolation	8 CPUs, 64Mbytes mem, separate fast disks	Ocean (4-way) 3 Flashlite 3 VCS	2 SPUs: 1 SPU Ocean, 1SPU Flashlite and VCS
Memory Isolation	4 CPUs, 16 Mbytes mem, separate fast disks	Multiple Pmake jobs (four parallel compiles each)	Balanced: 2 SPUs (1 job) Unbalanced: 1 SPU (1 job), 1 SPU (2 jobs)
Disk Isolation	2 CPUs, 44 Mbytes mem, shared HP97560 disk	Pmake and File copy	1 SPU pmake, 1 SPU file copy

TABLE 1. The workloads used for the performance results. For each workload we show the relevant system parameters, the applications used in the workload, and the SPU configuration for performance isolation.

similar to the well-studied priority inversion problem, and the solution is similar [SRL90]. A process blocking on a semaphore should transfer its resources to the process holding the semaphore until the semaphore is released. The severity of these problems scales with the number of processors and the kernel activity of the workload. However, most of these problems are not specific to performance isolation, and need to be addressed when designing scalable multiprocessor operating systems.

We encountered and fixed two such semaphore problem in our implementation of performance isolation. The first was the `inode_lock` semaphore that protects inodes in the file system. The contention for the root inode has the potential to completely break performance isolation. We changed this from a mutual exclusion semaphore to a multiple-readers/ one-writer semaphore because the dominant operation is lookups to the inode. We also reduced the granularity of the `page_insert_lock` semaphore that protects the mapping from file vnode and offset to pages of physical memory. These two changes were required to provide performance isolation, but also improved the response time of the base IRIX system. The improvement in response time was as much as 20-30% on a four processor system for some workloads.

4. Performance Results

This section demonstrates how well performance isolation is able to achieve its twin objectives of isolation and sharing. We will run a number of workloads using our implementation of *performance isolation* (**PIso**), as described in Section 3.. The workloads are summarized in Table 1. Each workload has jobs from multiple SPUs. To clearly demonstrate the effectiveness of performance isolation, we will also run each workload on two other resource allocation schemes, as shown in Table 2. The first uses *fixed quotas* (**Quo**) to statically allocate the resources on the system to the different SPUs, thus providing good isolation, but no sharing. The second is unmodified[1] IRIX5.3 (**SMP**) that provides only the unconstrained sharing of resources as seen in current shared-memory multiprocessors, but no isolation. For each workload, we will demonstrate that performance isolation is able to provide isolation comparable to fixed quotas and good throughput through careful sharing of resources comparable to SMP.

Configuration	Description
Fixed Quota (**Quo**)	Fixed quota for each SPU with no sharing. (Good isolation)
Performance Isolation (**PIso**)	Performance isolation with policies for isolation and sharing.
SMP operating system (**SMP**)	Unconstrained sharing with no isolation. (Good sharing)

TABLE 2. Resource allocation schemes for MPs. Each workload is run with three different resource allocation schemes, Performance Isolation (PIso). Fixed Quotas (Quo), and IRIX5.3 representative of current shared-memory operating systems (SMP).

4.1 Experimental Environment

We implemented the performance isolation model in the IRIX 5.3 kernel from Silicon Graphics as described in the previous section. This is an SMP kernel designed to run on bus-based machines. The hardware used is an eight processor bus-based shared-memory machine simulated using SimOS [Her98] [RHW+95], a complete machine simulator, configured to model the CHALLENGE family of SMP machines from Silicon Graphics. The relevant characteristics of the machine are as follows: 300 MHz R4000 CPUs, 1Mbyte L2 cache with 128 byte line size, nominal latency to memory on a secondary cache miss 500 nanoseconds. The main memory size used was varied for the different workloads. The disk model used for some of the runs is based on a HP97560 disk [KTR94]. All SPUs access separate disks, except in the fourth workload that shows performance isolation for disk bandwidth.

We run our experiments on SimOS instead of a real machine because SimOS allows us to easily configure different systems; change the number of processors, the size of main memory, and the number of disks. This was very important

[1] The IRIX5.3 kernel used for these experiments has been modified to include the semaphore fixes described in Section 2.2.4, and therefore has better performance than the standard IRIX5.3 kernel.

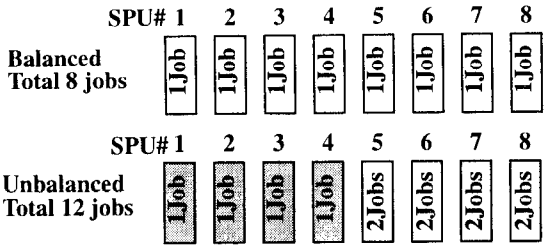

FIGURE 1. SPU configurations for the Pmake8 workload. The figure shows the distribution of jobs to SPUs in the balanced and unbalanced configurations for the Pmake8 workload.

for the results that we show in this section. SimOS also provides good support for kernel debugging and statistics collection, that would be quite difficult on a real system.

4.2 Experiments Using the Pmake8 Workload

The first workload consists of a number of pmake jobs as described in Table 1. There are eight SPUs for performance isolation corresponding to eight different users on an eight-way multiprocessor. The hardware resources are shared equally between the eight SPUs. The sharing policy is to share all *idle* resources with any of the other SPUs that need resources.

We consider two different scenarios for the distribution of processes to SPUs as shown in Figure 1. The first is a balanced configuration with eight jobs, one per SPU for performance isolation. This is our base configuration, and it should not be affected by isolation or sharing. The second is an unbalanced configuration where four SPUs (1 - 4) run one job each, and the other four SPUs (5 - 8) run two jobs each. SPUs 1 - 4 should see a benefit from being isolated from the more heavily loaded SPUs 5 - 8. On the other hand, SPUs 5 - 8 should see some benefit from sharing of resources that may be idle in SPUs 1 - 4. This workload will therefore be used to demonstrate performance isolation for both processor and memory resources.

4.2.1 Isolation

We will first study how well performance isolation can isolate SPUs from changes in system load. To do this we compare the performance of the jobs in SPUs 1-4 for the balanced and unbalanced configurations. The unbalanced configuration has higher system load because of the additional jobs in SPUs 5 – 8. In a system with good isolation, the performance of the jobs in SPUs 1 – 4 should not change. Figure 2 shows the average response time for these jobs in the balanced and unbalanced configurations, normalized to that of SMP in the balanced configuration.

Performance Isolation (PIso) is able to keep the performance of jobs in the lightly-loaded SPUs (1 – 4) the same in the balanced and unbalance configurations, despite the increase in overall system load in the unbalanced configuration. It does this by allocating resources based on SPUs, and effectively isolating jobs in an SPU from the load of jobs in other SPUs. Performance Isolation (PIso) is able to achieve the same level of isolation for jobs as the fixed quotas scheme (Quo), which is the ideal for providing

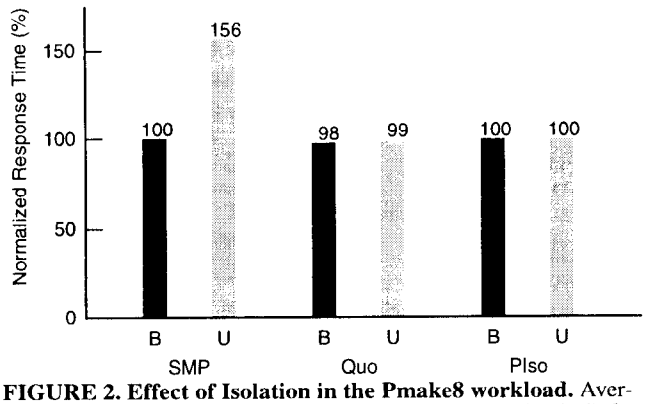

FIGURE 2. Effect of Isolation in the Pmake8 workload. Average response time for jobs in the lightly-loaded SPUs (1–4) for the balanced (B) and unbalanced (U) configurations normalized to the SMP time in the balanced configuration.

isolation.

In contrast, the regular SMP kernel (SMP) is unable to provide any isolation between jobs. The response time for the jobs in SPUs 1 – 4 increases by 56% when going from the balanced configuration with 8 jobs to the unbalanced configuration with 12 jobs. This kernel does not differentiate between the jobs, and gives all jobs approximately the same share of resources. Therefore, there is an increase in the response time of all jobs including those of the lightly-loaded SPUs (1 – 4).

4.2.2 Resource Sharing

We consider the performance of the heavily-loaded SPUs (5 – 8) in the unbalanced configuration to study the controlled-sharing aspect of performance isolation. The performance of the jobs in these SPUs is shown in Figure 3. The average response time is shown for each of SMP, Quo, and PIso normalized to the SMP performance in the balanced configuration.

The SMP kernel represents the best case for sharing and throughput. The jobs in SPUs 5 – 8 do well under SMP because they are able to take up more than their "fair share" of resources. This scheme treats all jobs equally, and gives them all the same level of resources. As a result the response

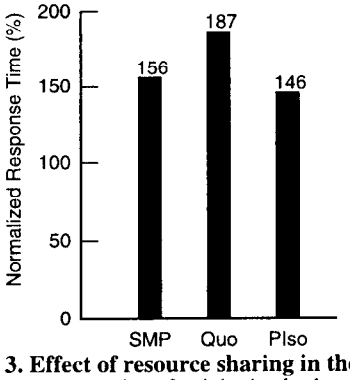

FIGURE 3. Effect of resource sharing in the Pmake8 workload. Response time for jobs in the heavily-loaded SPUs (5–8) for the unbalanced (12 jobs) configuration normalized to the SMP time in the balanced configuration.

SPU 1	SPU 2
4 process OCEAN	3 VCS & # Flashlite
Half the machine	Half the machine
(4 processors)	(4 processors)

FIGURE 4. SPU configurations for the CPU Isolation workload. The CPU isolation workload has two SPUs each of which gets half the machine. SPU 1 runs a four processor parallel Ocean application. SPU 2 runs three copies of VCS and three copies of Flashlite. SPU 2 has more CPU load than SPU 1.

time of the jobs in these SPUs increases by only 56% even though their resource requirements double.

Fixed quotas (Quo) are unable to do resource sharing. There are resources idle in SPUs 1 - 4, but they cannot be used by SPUs 5 - 8 because of the static fixed quotas. Therefore, Quo increases the response time for these jobs by 87%, performing much worse than the SMP case.

In contrast, performance isolation is able to provide controlled sharing of resources in addition to isolation between SPUs. The performance of these jobs with PIso is as good as that with SMP[1]. Performance isolation achieves this by carefully allowing these heavily-loaded SPUs to utilize resources that are idle in the lightly-loaded SPUs (1 – 4). From the isolation numbers for SPUs 1 – 4 in Figure 2, we know that this sharing is achieved without breaking isolation for the lightly-loaded SPUs.

4.3 Experiments Using the CPU Isolation Workload

The CPU isolation workload consists of compute-intensive scientific and engineering jobs with kernel time only at the start-up phase. The structure of the workload is shown in Figure 4. The workload has a total of ten processes on eight processors, and it will be used to demonstrate CPU isolation. There is adequate memory for all applications and so memory is not an issue for performance. For the performance isolation runs there are two SPUs corresponding to two users. Each SPU is allocated four CPUs. One SPU runs the four process Ocean application, and the other SPU runs the three Flashlite and three VCS jobs. Figure 5 shows the results for this workload. Response time numbers are averages of all the jobs of a type normalized to the SMP case.

For isolation we focus on the performance of Ocean, which runs in the SPU with a lighter load as it has four processors for four processes. It should benefit from isolation. For the

[1] Actually, the response time for PIso is a little better than that of SMP. From a pure CPU-scheduling viewpoint, they should have performed about the same. The difference is a result of the effect of different amounts of memory available during the run. This happens because the light-load SPUs finish early, and they release memory in addition to CPUs. This memory then becomes available to the heavy-load SPUs. In the SMP case all the jobs are equal, and finish at about the same time, using their share of memory till the end.

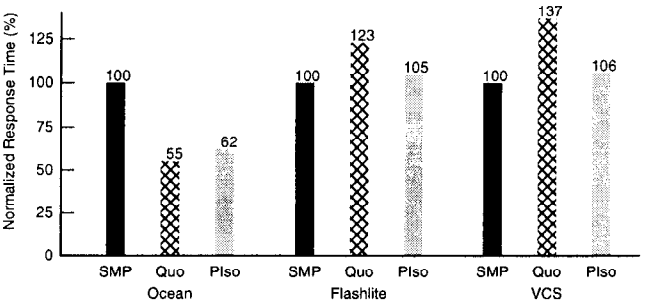

FIGURE 5. Response times for a compute intensive workload. For performance isolation, Ocean runs in one SPU (four processes on four processors) and all the Flashlite and VCS jobs in another (six processes on 4 processors). The response time (latency) shown is the average for all jobs of a type, and is normalized to that for the SMP case.

Ocean processes, performance isolation (PIso) is able to improve the response time compared to SMP. PIso does this by isolating the processes within an SPU, preventing interference from the other applications. In the SMP configuration, the Ocean processes run slower because all the processes are treated equally. Therefore Ocean gets less than its "fair share" of CPU time, and sees interference from the other processes. Fixed quotas (Quo) the ideal case for isolation does a little better than PIso.

For sharing we focus on the Flashlite and VCS processes that are running in the SPU with heavier load as there is six processes running on four processors. SMP is the ideal for sharing of resources. The VCS and Flashlite processes perform significantly better with Performance Isolation (PIso) than with Fixed quotas (Quo). Performance Isolation achieves this by carefully utilizing CPU resources that would have been idle in the Ocean SPU, in such a way as to not affect isolation for the Ocean processes. Performance isolation (PIso) is also able to keep the performance of the VCS and Flashlite processes comparable to that of the SMP configuration in this case. Because the workloads are dissimilar this last result will not generally be the case, and will be dependent on the relative durations of the applications.

4.4 Experiments Using the Memory Isolation Workload

The memory isolation workload will highlight performance isolation for main memory. This workload and the experiments are similar to that of the Pmake8 workload, but with a focus on memory. The structure of this workload is shown in Figure 6. There are two SPUs on a four processor system. The total memory size is deliberately made small (16 Mbytes). This memory is enough to run one job in each SPU, but leads to memory pressure in a SPU with two jobs. The results highlighting isolation and sharing are shown in Figure 7. The graphs are similar to the ones for the Pmake8 workload, and can be interpreted similarly.

The effect of providing isolation is illustrated by the lower graph that shows the performance of the job in SPU1 in the balanced and unbalanced configurations. Performance

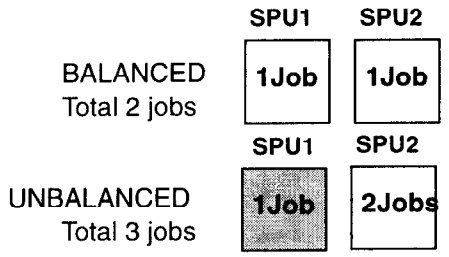

FIGURE 6. SPU configurations for the memory-isolation workload. The figure shows the distribution of jobs to SPUs in the balanced and unbalanced configurations for the memory-isolation workload.

isolation is able to provide isolation to maintain performance as the background system load increases. Only a 13% decrease in performance compared to the SMP case of 45% decrease. SMP treats all processes the same, resulting in less resources and lower performance for the processes of SPU1 as system load increases.

The effect of sharing is illustrated by the upper graph that shows the performance of the jobs in SPU2 in the unbalanced configuration. The loss in performance with fixed quotas is large, 145% decrease in performance compared to the balanced configuration. A 100% reduction in performance is accounted for by CPU time because there are two jobs instead of one. The additional 45% is because of the memory limitation when running two jobs in one SPU without sharing. Performance isolation through the careful sharing of resources — memory and CPU in this case — delivers significantly better performance, close to the SMP case.

4.5 Experiments Using the Disk Bandwidth Isolation Workload

We demonstrate the effect of performance isolation on disk bandwidth using two different I/O intensive workloads. For these runs we use the disk model based on the HP97560 disk. To reduce the length of the simulation runs we use a scaling factor of two for the disk model, i.e., the model has half the seek latency of the regular disk. We also make sure that the file buffer caches are cold for these experiments. To keep the experiment simple, the machine is a two-way multiprocessor.

The first workload (pmake-copy) has two SPUs, one running a pmake job, and the other a process copying a large file (20Mbytes). A single disk contains both the source and results of the pmake, and the source and destination file of the copy. The pmake makes a total of 300 requests to the disk, and these are not all contiguous as they access multiple files and have many repeated writes of meta-data to a single sector. The copy makes a total of 1050 requests to the disk. These are mostly contiguous sectors as they are reading and writing large files. There are multiple outstanding reads because of read-ahead by the kernel. The buffer cache fills up causing writes to the disk.

The disk-request scheduling algorithm in IRIX 5.3 is optimized for throughput as was described in Section 3.3.

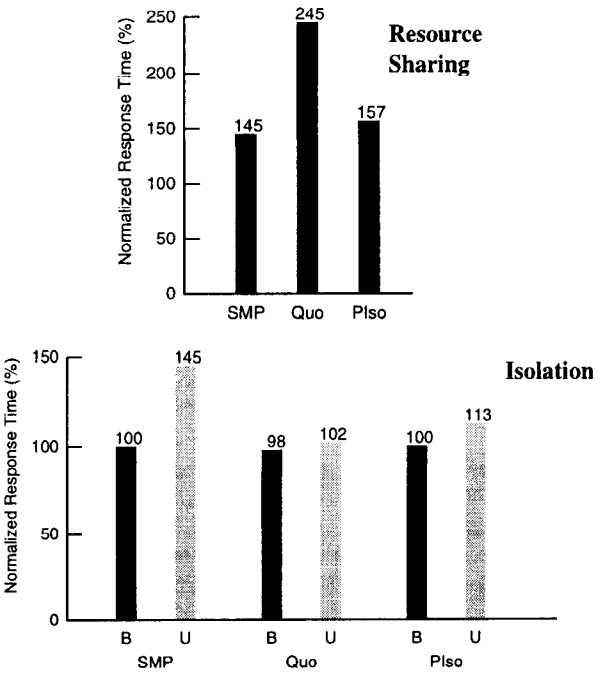

FIGURE 7. Performance Isolation for a memory-limited workload. The graph at the bottom shows the effects of providing isolation for SPU1 that runs 1 job in both the balanced (B) and the unbalanced (U) configuration. The graph on the top shows the effect of resource sharing for SPU2 in the unbalanced configuration running two jobs. The response time (latency) for all jobs are shown normalized to that of the balanced SMP case.

Its primary consideration is to reduce disk seek latency. Therefore, it considers only the current head position and the sector number of the requests when scheduling. As a result, the reads and writes of the copy that are to contiguous sectors, can lock out the more random requests from the pmake. This locking out of other requests is the same behavior that users sometimes observer when a large core file is dumped to disk.

To show the effect of all the issues and how performance isolation is able to deal with them, we consider three different policies for scheduling disk requests for the experimental runs.

1. **Pos:** The standard head-position based scheduling, currently in IRIX.
2. **Iso:** This a blind performance isolation policy. This policy ignores head position, and only strives to provide fairness for disk bandwidth to the SPUs.
3. **PIso:** The performance isolation policy described in Section 2.2.3. This policy gives weight to both isolation and the head position when scheduling requests. The goal is to balance fairness of bandwidth and effective throughput to the disk.

The results for the pmake-copy workload for the three cases are shown in Table 3. The performance isolation policy (PIso), by incorporating fairness, significantly reduces the response time for the pmake job (39%). In the regular IRIX case (Pos), the copy job was able to lock out the pmake

Conf	Response time (sec)		Avg. Wait Time (ms)		Avg. Latency (ms)
	Pmk	Cpy	Pmk	Cpy	
Pos	15.7	12.8	94.3	38.4	10.6
Iso	9.2	15.8	23.4	63.5	11.6
PIso	9.6	15.6	23.0	75.4	11.4

TABLE 3. The effect of performance isolation on a disk-limited workload. The pmake-copy workload consists of two SPUs, one a pmake process (Pmk) and the other a large (20 Mbyte) file copy (Cpy) to the same disk as the pmake. The response time and the average wait time per request for each job is given, along with the average disk latency.

Conf	Response time (sec)		Avg.Wait Time (ms)		Avg. Latency (ms)
	Small	Big	Small	Big	
Pos	0.93	0.81	155.8	12.1	6.4
Iso	0.56	1.22	68.9	23.7	8.2
PIso	0.28	0.96	31.9	16.6	6.6

TABLE 4. The advantage of considering both head-position and fairness. The big-and-small-copy workload consists of two processes copying files, one a small 500Kbyte file (Small) and the other a larger 5 Mbyte file (Big). The response time and the average wait time per request for each job is given, along with the average disk latency.

requests from the disk, significantly slowing down the pmake job. The fairness provided by PIso can be seen in the significantly lower average wait time for the requests from pmake. These requests now do not have to wait for all copy requests to be processed. For the pmake job, the average time a request spends waiting in the disk queue decreases by 76%. Performance isolation, by incorporating head-position information in the scheduling decision, does not significantly change the average seek latency for the disk. The copy job, as expected, does see a reduction in performance (23%).

The blind performance isolation policy (Iso) that ignores head position is also able to improve fairness, and consequently the response time for the pmake. In this workload its performance is similar to the performance isolation policy because the pmake makes fairly irregular requests, therefore ignoring disk-head position does not result in a large penalty. However this is not always true, and completely ignoring disk-head scheduling could lead to reduced performance.

The second workload (big-and small-copy) will illustrate the importance of maintaining disk-head position as a factor in the scheduling decision. In this case also there are two SPUs, one with a process copying a small file (500 Kbytes), and the other with a process copying a larger file (5 Mbytes). Both jobs in this workload can benefit from disk-head position scheduling because they are both accessing contiguous sectors on disk in a regular manner. The results of the experiment are shown in Table 4.

The big difference between the two workloads is that in this workload, the smart performance isolation policy (PIso) is able to significantly outperform the blind one (Iso). In IRIX (Pos), the larger copy by happening to issue requests to the disk earlier than the smaller copy, is able to lock out the requests of the smaller copy. Both the PIso and Iso policies provide fairness, improving the response time of the small copy and allowing it to finish sooner than the larger one. However, the PIso policy provides better response times for both processes as compared to the Iso policy because it incorporates head-position information also. The average seek latency per request for the PIso policy is about the same as the IRIX position-only scheduling policy (Pos). The Iso policy pays almost a 30% increase in average seek latency. The average time a request spends waiting in the disk queue decreases from Iso to Piso for both the small and the large copy, 54% for the former and 30% for the latter.

5. Related Work

The closest work to performance isolation is in the IBM mainframe space. The Workload Manager (WLM) functionality [AEE+97] of the IBM OS390 operating system is extremely sophisticated, and allows the specification of high-level performance goals and an importance value for these goals. These goals can be of the form of desired response times for tasks or transactions or speed of execution (velocity) for batch jobs. The system continually monitors resource usage and application performance, and uses this information to readjust resource allocation to meet the specified goals. WLM also works across a cluster of machines and in client-server environments. However, to be successful, such a system requires fairly close coupling with applications to recognize entities such as transactions, clients and their corresponding servers, etc. It also requires a good apriori knowledge of the applications to be controlled, so that acceptable goals can be specified. These conditions are probably normal in the mainframe world.

Our idea of performance isolation has a different "philosophy" with more modest goals, addresses a more chaotic environment, and is simpler to implement. It only guarantees isolation not performance, i.e., a minimum level of resources which can be used by an SPU. The task load placed on an SPU decides the resulting performance. Performance isolation requires only minimal static configuration and is targeted at general-purpose servers where the tasks a user may run could be unknown. It should be noted that the underlying controls in the OS390 systems seem to be sufficient to implement performance isolation should it be desired.

Other than the above mentioned OS390 work, performance isolation has not been really studied for general-purpose servers. The SPU kernel abstraction that explicitly assigns the machine resources to groups of processes and enables

different sharing policies, makes performance a first-class kernel citizen similar to address-space protection as provided by the virtual memory system. Previous work in resource allocation has taken a piecemeal approach, focussing only on allocating individual resources (mainly CPU time) or concerned only with individual processes. What has been lacking in these piece-meal solutions is a comprehensive solution that encompasses all resources that can impact application performance, and is able to deal with arbitrary groups of processes. While most other work has been for uniprocessors only, we specifically target shared-memory multiprocessors, where the problems from interference are more acute.

We now describe some of the other techniques that have been proposed for the allocation of individual resources. Waldspurger [Wal95] demonstrates stride scheduling, a technique for providing proportional-share resource management for a variety of computing resources, including CPU, memory, disk and network bandwidth. They do not consider an unified solution that accommodates all the resources, and their solutions are only proposed for uniprocessors, not multiprocessors. Their work is the only one to attempt at fairness for disk bandwidth allocation. Using simulations they show that for certain limited workloads their "funding delay cost" model for scheduling disk requests can achieve fairness. Our implementation that balances head position and fairness is different and more generally applicable. They provide a real implementation only for the CPU time resource, and the analysis for other resources is done using simple simulations. An important contribution of our work is a real implementation of all the mechanisms and policies described.

A number of studies have considered fairness when allocating a single resource. Most of these studies have concentrated on CPU scheduling [Hen84][KaL88]. An extension to [KaL88], the SHAREII resource management tool [Sof96] also assigns *fixed quotas* for virtual memory and other non-performance related resources such as disk space. A few proposals consider fairness for memory allocation. [Cus93] describes the scheme used to allocate memory to processes in Windows NT. This scheme consists of assigning shares of pages to *individual processes*, changing these shares dynamically based on page-fault rates, and a local page replacement policy. They operate at process level, and provide no support for grouping processes. [HaC] have a proposal for allocating memory and paging bandwidth to disk using a market approach. They assume that there are enough processors, so CPU time is not an issue they consider. Their unit of fairness is again *individual processes*.

Though we do not discuss performance isolation for network bandwidth, the implementation would be similar to that of disk bandwidth, without the complication of head position. Stride scheduling is used in [Wal95] to study fairness for network bandwidth by changing the order of service from FCFS. Also, Druschel and Banga [DrB96] implement a scheme called lazy receiver processing (LRP) to provide fairness for network bandwidth.

The Stealth Distributed Scheduler [KrC91] implements isolation goals similar to ours in a limited sense, in the context of distributed systems when scheduling foreign processes on a user's workstation. They solve the simpler problem of preserving the performance of a single class of higher priority "local" processes by pre-empting resources from "foreign" processes.

Our work is also different from the whole class of real-time systems because these systems primarily use resource specification and admission control as a means to provide hard guarantees to jobs. Performance isolation does not require per-application resource specification, and does not use admission control because it only guarantees a certain level of resources, not response times or deadlines. At a high level, our SPU concept is similar to that of resource reserves used in the real-time system described in [Mer97].

6. Concluding Remarks

The tight coupling of processors, memory, and I/O in shared-memory multiprocessors enables SMP operating systems to efficiently share resources. There has however been a popular perception that unlike workstations, SMP kernels (such as UNIX or Windows NT) on commodity shared-memory multiprocessors cannot isolate the performance of a user or a group of processes from the load placed on the system by others. This work demonstrates that with better resource allocation policies and mechanisms a shared-memory multiprocessor server can provide workstation-like isolation in a heavily loaded system and maintain the benefits of resource sharing of SMPs.

Performance isolation replaces the process-only CPU-centric control over resource sharing found in current SMP operating systems, and gives users or tasks significantly better control over the performance they can expect when utilizing a shared machine. Groups of processes are isolated from the background load on the system, and are guaranteed a fixed share of the machine's resources based on pre-configured contracts or agreements. Performance isolation also maintains good throughput by carefully reallocating under-utilized resources to processes that might need them.

To implement performance isolation, we introduce a kernel abstraction called the Software Performance Unit (SPU). The SPU associates computing resources on the system with groups of processes that are entitled to these resources, and restricts the use of these resources to the owning processes. The SPU is the unit of isolation, and provides a powerful mechanism to enable different contracts between users or services for sharing a larger machine. Each SPU has associated with it a sharing policy that decides how and when resources belonging to the SPU may be shared with other SPUs.

We implement performance isolation in the IRIX5.3 kernel from Silicon Graphics for the CPU, memory and disk bandwidth resources of the system. Running a diverse set of workloads on this kernel using SimOS, we demonstrate that

performance isolation is feasible and robust across a range of workloads and resources. The results show that performance isolation is successful at providing workstation-like isolation under heavy load, SMP-like latencies under light load, and SMP-like throughput. Given the benefits and robustness of the results, we believe performance isolation should be seriously considered for implementation for all SMP server operating systems.

References

[ACP+94] T. Anderson, D. Culler, D. Patterson. A Case for NOW (Networks of Workstations). In *IEEE Micro,* February 1995.

[ABL+91] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 95-109, October 1991.

[AEE+97] J. Aman, C. Eilert, D. Emmes, P. Yocom, and D. Dillenberger. Adaptive algorithms for managing a distributed data processing workload. In *IBM Systems Journal*, vol. 36, no. 2, pages 242-283, 1997.

[AOG91] D. Anderson, Y. Osawa, and R. Govindan. Real-time disk storage and retrieval of digital audio/video data. In *Technical Report CSD-91-646*, Computer Science Department, U. of California, Berkeley, August, 1991

[CDV+94] R. Chandra, S Devine, B Verghese, A Gupta, and M. Rosenblum. Scheduling and Page Migration for Multiprocessor Compute Servers. In *Proceedings, Architectural Support for Programming Languages and Operating Systems*, 12-24, October 1994.

[Cus93] H.Custer. Inside Windows NT. Microsoft Press, 1993.

[DrB96] P. Druschel and G. Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 261-275, October 1996.

[HaC] K. Harty and D. Cheriton. A Market Approach to Operating System Memory Allocation. Stanford TR, *http://www-dsg.stanford.edu/Publications.html.*

[Hen84] G.J. Henry. The Fair Share Scheduler. AT & T Bell Laboratories Technical Journal, October, 1984.

[Her98] S. Herrod. Using Complete Machine Simulation to Understand Computer System Behavior. Technical Report: STAN-CS-TR-98-1603, Computer Science Department, Stanford University, February 1998.

[Hyd94] E. Hyden. Operating System Support for Quality of Service. Ph.D. Thesis, Wolfson College, University of Cambridge, February, 1994.

[KaL88] J.Kay and P.Lauder. A Fair Share Scheduler. Communications of the ACM, January, 1988.

[KrC91] P. Krueger and R. Chawla. The Stealth Distributed Scheduler. In 11th International Conference on Distributed Computing Systems, May, 1991.

[KTR94] D. Kotz, S. Toh, and S. Radhakrishnan. A Detailed Simulation Model of the HP 97560 Disk Drive. Dartmouth PCS-TR94-220, July, 1994.

[LeL82] H. Levy and P. Lippman. Virtual Memory Management in the VAX/VMS Operating System, IEEE Computer, March, 1982

[Mer97] C. Mercer. Operating system resource reservation for real-time multimedia applications. *Ph.D. Thesis, School of Computer Science, Carnegie Mellon University,* CMU-CS-97-155, June 1997.

[Ous82] J.K. Ousterhout. Scheduling Techniques for Concurrent Systems, In 3rd International Conference on Distributed Computing Systems, 1982

[RHW+95] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete Computer Simulation: the SimOS approach. In *IEEE Parallel and Distributed Technology*, Fall 1995.

[Sof96] SHAREII: A resource management tool. SHAREII data sheet, Softway Pty. Ltd., Sydney, Australia.

[SRL90] L. Sha, R. Rajkumar, and J. Lehozcky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, pages 1175-1185, September 1990.

[Teo72] T. Teory. Properties of Disk Scheduling Policies in Multiprogrammed Computer Systems. In *Proceedings of AFIPS Fall Joint Conference*, pages 1-11, 1972.

[TuG91] A. Tucker and A. Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In P*roceedings of the Twelfth ACM Symposium on Operating Systems Principles,* pages 159-166, December 1991.

[Wal95] C.A. Waldspurger. Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management. *Ph.D. Thesis, Massachusetts Institute of Technology*, September 1995.

[WaW95] C.A. Waldspurger and W.E. Weihl. Stride Scheduling: Deterministic Proportional-Share Resource Management. Technical Memorandum MIT/LCS/TM-528, June, 1995.

[WOT+95] S. Woo, M. Ohara, E. Torrie, J.P. Singh, A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In Proceedings 22nd Annual International Symposium on Computer Architecture, June, 1995.