Performance Isolation and Resource Sharing on Shared-Memory Multiprocessors

Ben Verghese
Anoop Gupta
Mendel Rosenblum

Technical Report: CSL-TR-97-730

July 1997

This work was supported in part by ARPA contract DABT63-94-C-0054.
Mendel Rosenblum is partially supported by a National Science Foundation Young Investigator award.
Anoop Gupta is partially supported by a National Science Foundation Presidential Young Investigator award.

Performance Isolation and Resource Sharing on Shared-Memory Multiprocessors

Ben Verghese, Anoop Gupta, and Mendel Rosenblum

Technical Report: CSL-TR-97-730

July 1997

Computer Systems Laboratory
Department of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305-2140
{pubs}@cs.Stanford.EDU

Abstract

Shared-memory multiprocessors are attractive as general-purpose compute servers. On the software side, they present programmers with the same programming paradigm as uniprocessors, and they can run unmodified uniprocessor binaries. On the hardware side, the tight coupling of multiple processors, memory, and I/O enables efficient fine-grain sharing of resources on these systems. This fine-grain sharing is important in compute servers because it allows idle resources to be easily utilized by active jobs leading to better system throughput. However, current SMP operating systems do not provide an important feature that users of workstations enjoy, namely the lack of interference from the jobs of unrelated users.

We show that this lack of isolation is caused by the resource allocation model carried over from single-user workstations, which is inappropriate for multi-user multiprocessor systems. We propose "performance isolation", a new resource allocation model for multi-user multiprocessor compute servers. This model allows the isolation of the performance of groups of processes from the load on the rest of the system, provides performance comparable to a smaller system that corresponds to the resources used, and allows the sharing of idle resources for throughput comparable to a SMP OS. We implement the performance isolation model in the IRIX5.3 operating system for three important system resources: CPU time, memory, and disk bandwidth. Our implementation of fairness for disk bandwidth is novel. Running a number of workloads we show that this model is very successful at providing workstation-like latencies under heavy load and SMP-like latencies under light load.

Key Words and Phrases: SMP, resource allocation, performance isolation, fairness, resource sharing.

1.0 Introduction

Shared-memory multiprocessors are being increasingly used as general-purpose compute servers. These systems are attractive for a number of reasons. On the software side, they present programmers with the same programming paradigm as uniprocessors, and they can run unmodified uniprocessor binaries. On the hardware side, the tight coupling of multiple processors, memory, and I/O allows the aggregation of enormous computing power in a single system, and the efficient fine-grain sharing of these resources between processes.

The fine-grain sharing enables flexible and automatic reallocation of the resources to accommodate the disparate resource requirements of processes in a typical compute-server workload. 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. With unconstrained sharing and contention for resources, jobs belonging to one grouping can severely impact the performance of jobs belonging to other groupings. Shared-memory systems represent one end of the spectrum for clustering computing resources, as seen in Figure 1. These systems seem to favor overall throughput at the cost of response time for individual jobs. This perception has led to many detractors of these systems. This paper will show that overall throughput does not have to come at the cost of response time.

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. A user's jobs running on a workstation are isolated from the impact of jobs on other workstations. However 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.

1.1 The Problem

We start with an example to illustrate the effect of the interference between jobs caused by the lack of isolation on a shared-memory multiprocessor. Consider a user A running a job on a uniprocessor system (the job in this case is a parallel make with two simultaneous active compiles). The user now participates in buying a multiprocessor with three others. This is a four-way multiprocessor that has four times the memory and disks as the uniprocessor. User A's share of the computing resources of the multiprocessor is equivalent to that of the uniprocessor system. Therefore, user A may expect that the observed performance on the multiprocessor should at worst be comparable to the uniprocessor performance.

Figure 2 shows the performance of user A's pmake job on the uniprocessor and the multiprocessor, normalized to the uniprocessor case (UP), under various system load conditions. In the first configuration (MP1), user A is the only one using the system. User A's job is able to finish quicker (35% improvement) by using the idle processors and additional memory. More available resources under low-load conditions is the primary attraction of moving to a shared-memory multiprocessor for compute-server type workloads. Idle resources are easily exploitable unlike a workstation solution.

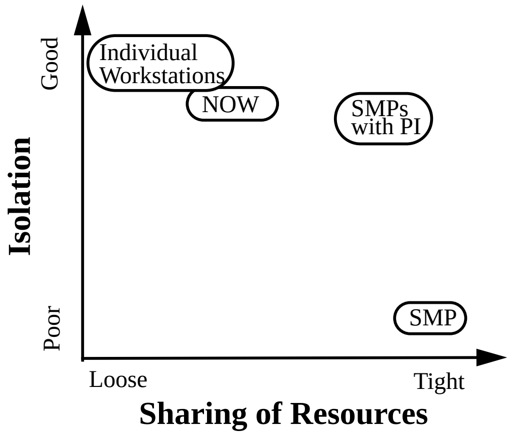


FIGURE 1. Different methods of clustering computing resources. The two axes here are sharing of resources and isolation of performance. Individual workstations provide good isolation of performance, but poor sharing of resources. The situation is the opposite for current SMPs. With Performance Isolation on shared-memory multiprocessors (SMPs with PI), we get most of the benefit on both axes, workstation response times under heavy load, and SMP response times under light load.

As many users of MP systems have observed, when the other users increase the load on the system, the performance of User A's job steadily gets worse even though its resource requirements do not change. MP6 represents such a case, where the other three users have started a total of 5 similar jobs on the system for a total of six jobs. Response time for user A, whose job has had constant resource requirements, has increased to 236% of the uniprocessor case. Another scenario familiar to multiprocessor users is four users each running a job, and one of the other users causes a core dump (MP4+D) to the disk used by user A. User A's response time is now over three times what it was on the uniprocessor due to the writes caused by the core dump competing for disk bandwidth with the pmake. This example clearly shows that the performance seen by an individual user on these systems is completely dependent on the background system load and the activities of other users. Clearly, enough incentive to retreat to individual workstations.

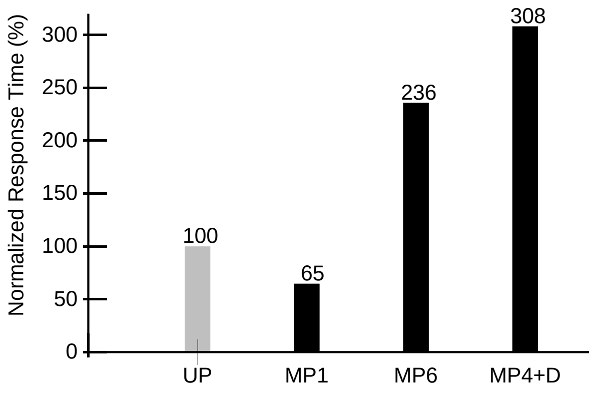


FIGURE 2. Performance of a single Pmake on a four-way MP. Response time for the Pmake job on an MP is shown for three different system workloads relative to the uniprocessor case (UP). MP1 is just the single Pmake; MP6 is six simultaneous Pmake jobs accessing different disks; and MP4+D is 4 simultaneous Pmake jobs, with one large file copy or core dump happening to the disk of one of the Pmakes.

Our assertion is that this degradation of performance is not inherent to the fine-grain sharing provided by the SMP operating systems on these machines. The observed degradation of performance is an artifact of the existing CPU-centric resource allocation model that might have been appropriate on single-user workstations, where access to the CPU implied access to all resources on the system. In this model, CPU allocation was done to maintain fairness by process, memory allocation was somewhat ad-hoc with static quotas per process at best, and allocation of I/O bandwidth was non-existent. Our example above shows that this model is not appropriate for modern multiprocessors (or uniprocessors) where CPUs, memory and I/O bandwidth are *all* determinants of overall performance. Resource allocation needs to be done to provide fairness not just between competing processes, rather it must accommodate competing groups of tasks or users.

One method to preserve latency is to enforce fixed quotas per user for the different resources. However, this method would not allow the sharing of idle resources, and significantly reduce the throughput and response time seen on these systems under light-load conditions. This paper presents "*performance isolation*", a new resource allocation model for shared-memory multiprocessors. This model isolates groups of tasks from the resource requirements of others, and preserves the inherent sharing ability of these machines. Performance isolation seeks to take advantage of the tight coupling and flexibility of shared-memory machines to provide both workstation-like latency under heavy load and SMP-like latency and throughput under light load.

2.0 A Framework for Performance Isolation

We will now develop and define the performance isolation model for resource allocation, and then provide a general framework for implementing this model in the operating system.

2.1 Performance Isolation Model

On large compute-servers there is usually an implicit contract between users. For example, all users have equal priority and expect equal performance, or project A owns a third of the machine and project B owns two thirds. However, current SMP operating systems do not provide a facility to enable or enforce such agreements, and so cannot give users any expectation of the performance that they will see.

We propose an abstraction called a *Software Performance isolation Unit* (SPU), to provide such a facility to flexibly partition the machine from a performance viewpoint. There are three parts to the SPU. The first is a criterion for assigning processes to an SPU. The desired basis for the grouping of processes can vary greatly, and is dependent on the environment of the particular machine. 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.

The second part is the share of resources assigned to the SPU. We are primarily interested in the computing resources that directly affect user performance: CPU time, memory, disk bandwidth, and network bandwidth. There are many possible ways of partitioning resources between SPUs. For example, it could be a fixed fraction of all the hardware resources, or it could be arbitrarily complex, such as 2 CPUs, 128 Mbytes of memory, half the bandwidth to all disks and the network.

The third part is a sharing policy. This policy decides when and to whom resources belonging to an SPU will be allocated when these resources are idle. One example of a sharing policy is to not give up any idle resources; this is approximately what an individual workstation provides. Another example is to share all resources with everyone all the time without consideration for when they are idle or not. This approximates the current SMP systems. We are interested in a sharing policy that falls between these two, and gives up idle resources in such a way as not to affect the performance of the SPU giving up the resources.

We propose a new resource allocation model for shared-memory multiprocessors based on the SPU, the performance isolation model, to get the benefits of both sharing and isolation. The performance isolation model essentially partitions the computational resources of the multiprocessor into multiple units, and from a resource allocation viewpoint the multiprocessor now looks like a collection of smaller machines. The resources corresponding to each of these smaller machines is assigned to a SPU. The distribution of resources to SPUs can be based on a previously configured contract.

The performance isolation model is defined by the following three goals:

1. If the resource requirements of a 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. The SPUs whose resource requirements exceed their configured fraction of machine resources should be able to easily use any idle resources to improve their response time and throughput.
3. The level of performance of the SPU under this model should be comparable to that of the SPU's processes running on a smaller machine with the same resources as the SPU. That is to say that the performance cost/overhead of providing isolation should be minimal.

To exploit the capabilities and resources of shared-memory systems, a dynamic and flexible solution is needed that is enforced in the kernel software. Only SPUs with active processes need to be allocated resources, and temporary reallocation of idle resources should be allowed. There are two parts to the solution for providing performance. First, we need to provide isolation between SPUs by implementing mechanisms in the kernel to restrict SPU resource usage to allocated limits. This will guarantee that even when the system is heavily loaded an SPU will be able to utilize all the resources it is entitled to, and that other SPUs will not be able to do so unless specifically allowed. Second, based on the sharing policy we need to reallocate idle resources to SPUs that may need them. This allows sharing of idle resources when the system is lightly loaded, providing for better throughput and response time.

2.2 Providing Isolation

Most current SMP operating systems are structured to allow unconstrained sharing of resources among processes. The resource allocation mechanisms are completely unaware of any higher abstraction, such as the user that owns the process. In order to provide isolation between SPUs two new aspects of functionality were needed in the kernel. First, the utilization of resources by individual SPUs needed to be tracked. For example, every time a memory page is allocated, the kernel needs to know which SPU is getting the page and increment its page usage counter. Second, mechanisms are needed to limit the usage of resources by a SPU to allocated levels. For example, currently in IRIX a request for a page of memory will only fail if there is no free memory in the system. With isolation, a request may need to be denied if a SPU has used its allocation of pages, even if there is still memory available in the system.

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 user SPU. Examples of the former are pages of memory accessed simultaneously by multiple SPUs such as shared library pages or code, and disk write requests that often contain pages from multiple SPUs. Examples of the latter are kernel processes, such as the pager and swapper daemons.

For such problems our strategy has been to choose the simplest solutions that seem reasonable. More sophisticated solutions are easily considered in the future, as we encounter instances where these proposed solutions clearly do not work. For the sharing 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 are allocated to the appropriate user SPUs. The kernel SPU has unrestricted access to all resources.

2.3 Policies for Sharing

The policies for sharing are a critical feature of the performance isolation model. The isolation mechanisms can restrict resource usage by a SPU to the limits set by the policy module. Setting fixed quotas would be enough to provide isolation between the SPUs. However, this would prevent the sharing of idle resources leading to poor throughput and response time when the system is lightly loaded. For performance isolation, the policy module enables sharing between SPUs by checking for idle resources, and changing the resource limits according to the sharing policy to allow reallocation of these resources.

If an SPU enables sharing, transferring all idle resources from underloaded SPUs to overloaded SPUs would result in the best system throughput. However, the resources that are currently idle in an SPU may be needed in the future. The key factor in making the decision to transfer resources is the cost of revoking these resources when they are needed again by the loaning SPU. The base performance of an SPU will be impacted if the resources that are loaned are unavailable when they are needed. 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. When making sharing decisions, the policy module needs to ensure that the cost of revocation does not impact the performance of the loaning SPU and break isolation. Based on the revocation cost, it is possible to choose the desired trade-off between isolation and sharing for each resource, anywhere from complete isolation (fixed quotas) to complete sharing (SMP).

TABLE 1. Summary of the performance isolation implementation. For each resource we show the metrics that need to be kept per SPU, the mechanism to isolate the performance of an SPU and the basic sharing policy.

Resource	Unit of allocation	SPU metrics tracked	Isolation Mechanism	Sharing Policy
CPU	Full CPU unit	CPUs allocated, CPUs idle	SPU id of process and processor must match while scheduling	SPU id ignored if CPU is idle
Memory	Page (4K)	Pages allocated, Pages currently used	Page granted only if used count < allocated count for SPU.	Increase allocation if SPU has memory pressure and total idle > reserved threshold
Disk BW	Sectors per second	Per disk - Sectors transferred, Pending requests	SPU skipped if transferred > (avg. of SPUs + BW difference threshold)	Head-position scheduling unless SPU is to be skipped

Beyond the basic goals outlined, the sharing policy space can be much more sophisticated. For example, SPUs could be hierarchical, with sharing happening at lower levels of the hierarchy before the higher-levels. Also, individual SPUs could choose different trade-offs between isolation and sharing. There could be a temporal element to the sharing policy, i.e., giving up a fraction of the idle resources after certain periods. This policy space is vast, and is a rich area for future work.

3.0 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 a prototype implementation of the model, giving details of how this model is actually implemented in the SGI IRIX5.3 kernel. 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). For this prototype, we make two simplifying assumptions. All resources are to be divided equally among all the active SPUs, though it will be obvious from the implementations that different shares can easily be supported. The sharing policy for all the SPUs is to share any idle resources equally with all other active SPUs.

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 in IRIX, and how the sharing policy enables sharing by reallocating idle resources. A high-level summary of the implementation for each resource is given in Table 1. For our implementation we picked reasonable policies and mechanisms that allow us to clearly demonstrate the effectiveness of performance isolation. Other mechanisms may chosen for each of the resources, and other sharing policies can be supported as appropriate.

3.1 CPU Time

In IRIX as in most UNIX systems, CPU time is allocated in time slices to processes. A priority-based scheduling scheme is used in which the priority of a process drops as it uses CPU time. This scheme maintains fairness at a process level. Isolation requires a mechanism to provide fairness at the SPU level. CPU time on a multiprocessor can be allocated either through time multiplexing or space partitioning of the CPUs. We chose the latter as it fits better with our model of partitioning the machine, and our assumption that there will be fewer active SPUs than CPUs. There are two issues with our choice of a pure space-partitioning policy for CPUs. If our assumption of more CPUs than SPUs does not hold it will be beneficial to switch to time partitioning the CPUs. Also, parallel applications that use a space partitioning policy [ABL+91][TuG91] can be easily accommodated in our current scheme. Accommodating gang-scheduled [Ous82] applications would require a hybrid space and time partitioning policy.

Each SPU is allocated an integral number of CPUs depending on its entitlement. The SPU to which a CPU is allocated is its home SPU. If in the division, fractions of CPUs need to be allocated to SPUs, this is implemented by periodically reassigning CPUs among SPUs. The period between such reassignments is kept fairly long to reduce the overhead of reassignment (currently 250 milliseconds).

In IRIX, a CPU normally picks the runnable process with the highest priority when scheduling a new process. To provide isolation this behavior is modified by having CPUs only select processes 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.

If a SPU is lightly loaded, one or more processors belonging to this SPU may be idle. Sharing is implemented by relaxing the SPU restriction when a processor becomes idle. If a processor cannot find a process from its home SPU, it is allowed to schedule processes from another SPU. The SPU getting the idle processor is not explicitly chosen, but because the process from the head of the priority queue is picked, it 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 non-home SPUs are tracked. If a process from the home SPU now becomes runnable, and there are no allocated processors 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-CPU 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 policy could consider not loaning every idle CPU, or not loaning a CPU if it is being revoked frequently.

3.2 Memory

The IRIX5.3 kernel supports per-process limits 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. These limits are per-process, and cannot provide the strict isolation that our model requires. In certain circumstances, they may inhibit sharing of idle resources in the system.

Isolation for physical memory between SPUs is enforced by setting a maximum limit on pages an SPU may use, and not allowing a SPU to use more pages than it has been assigned. Memory pages are conceptually space-partitioned among the SPUs, and the maximum limit represents the share of memory that an SPU is entitled to. The number of pages used by each SPU is counted. Page allocation is augmented to record the SPU id of the process requesting the page. 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.

As mentioned in Section 2.2 pages might be accessed by multiple SPUs. When a page is first accessed, it is marked with the SPU id of the accessing process. A subsequent access to the page by a different SPU will turn it into a shared page. The id is reset when the page is freed. The cost of these pages is assigned to the shared SPU. Similarly the cost of pages used by the kernel is assigned to the kernel SPU. The allocation of pages to SPUs is periodically updated to account for changes in the usage of the shared and kernel SPUs.

A request for pages beyond the allocated limit of the SPU will fail, and the process will be descheduled pending the freeing of pages. Corresponding changes are made to the paging and swapping functions to account for per-SPU memory limits. The pager process is woken up when a SPU is low on memory, and selectively frees pages from such SPUs. The original paging mechanism in IRIX5.3 only triggers when the system as a whole is low on memory, and it reclaims memory from all processes. The swapping of processes is modified to take into account the memory situation of SPUs when picking processes to swap in or out.

Sharing of idle memory is implemented by changing the allocation of memory to 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. The memory re-allocation is temporary, and is changed 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. 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 is configurable, and we chose 8% of the total memory. This value is what IRIX uses to decide if it is running low on memory. The reserve threshold reduces the chance of running out of pages on the loaning 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. In the C-SCAN algorithm the outstanding disk requests are serviced in the order encountered as the disk head sweeps from the first to the last sector on the disk. Stated in a different way, the request that is ahead of and closest to the current head position is chosen. When the head reaches the request closest to the end of the disk, it then goes back to the first sector and starts again. This technique reduces the disk-head seek component of latency, and prevents starvation. However, the process requesting the disk operation is not a factor in the algorithm, and this could significantly affect isolation of 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 while all requests of one SPU are serviced before requests from other SPUs are scheduled.

To provide isolation we need to account for disk bandwidth usage by SPUs, and incorporate this information when 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.

Second, frequently the writes to disks are done by system daemons that are freeing pages, such as bdflush (flushes buffer cache data) and pdflush (flushes page frame data). Therefore these write requests contain pages belonging to multiple SPUs. These requests are scheduled as part of the shared SPU, which is given the lowest priority. Once the write request is done, the bandwidth used to write each individual page is charged to the appropriate user SPU.

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 unit of allocation is sectors per second. 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. To accommodate SPU shares of bandwidth that are not equal, we would decay the count at a rate proportional to the share. This count of sectors transferred represents the bandwidth used by each SPU, and is kept for each disk.

Implementing performance isolation requires moving to a roundrobin-type scheduling of requests by SPU based on bandwidth shares of each of the SPUs. However, completely abandoning the current disk-head position based scheduling would result in poor throughput because of excessive delays caused by the extra seek time (see results in Section 4.5). Therefore, the policy used is a compromise that incorporates both disk-head-position-based scheduling and a fairness criteria.

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 (current count of sectors) exceeds the average bandwidth usage by a threshold (the *BW difference threshold*). Once a 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. The fairness criteria is checked for the requesting SPU after each disk request. The choice of the BW difference threshold allows a trade-off. Smaller values imply better isolation, and a choice of zero resulting in round-robin scheduling. Larger values imply smaller seek times, and a very large value resulting in the normal disk-head-position scheduling.

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 a SPU with high disk utilization can affect the performance of another SPU using the same disk. However, we will show that performance isolation can considerably reduce the impact of such shared access.

3.4 Kernel Resources

In addition to the physical resources discussed above, there are shared kernel structures in SMP kernels that need to be considered in the implementation of the performance isolation model. These shared structures can be accessed and updated from any processor, and contention for these resources is a potential source of problems. The impact of these problems scales with the number of processors and the kernel activity of the workload. Although they can get in the way of providing good performance isolation, most of these problems are not specific to performance isolation, and need to be addressed when designing scalable SMP operating systems. We outline the sources of these problems and our solutions to some that we encountered.

First, updates to shared structures need mutual exclusion, and this is provided through spinlocks and semaphores. Excessive contention for these structures will lead to reduction in performance as compared to a smaller system. Contention for semaphores results in blocking of processes. Contention for spinlocks results in additional kernel time spent spinning on a location. Both these will increase the response time of a task.

In our implementation of performance isolation, we encountered and fixed several of these problems. For example, mutual exclusion for access to file system inodes was provided through a semaphore per inode. There is considerable contention for the inode semaphore at the higher levels of the file system. This allows interference between unrelated processes for the root of the filesystem, completely breaking performance isolation even on a four processor system for a filesystem intensive workload like parallel make. The dominant operation on these high-level inodes a lookup. This is essentially a read-only operation which takes a directory inode and a string representing the next part of the path and returns an inode that corresponds to the child. Our solution was to modify the inode lock to be a multiple readers single write lock, thus allowing multiple lookups to proceed in parallel. This removed the contention problem for the inode lock. As would be expected this improved the performance of the basic SMP kernel too. In general, I/O to a shared file system seems to have the potential to break performance isolation, and care should be taken when designing such systems.

Another example is the page_insert_lock semaphore protecting the mapping from file vnode/offset to pages of memory. This was a single semaphore for the entire hash table of mappings, a much larger granularity than necessary. By splitting this semaphore using a hash function on the vnode/offset, we were able to reduce the contention on this semaphore greatly. There are other semaphores that we noted, but did not address because their impact on performance was not significant in our workloads. The above two changes were required to provide performance isolation, and as would be expected also improved the response time of the base IRIX system by 20-30% on a four processor system for some workloads.

An example of a spinlock with high contention is memlock, which protects many of the important shared VM structures in the kernel. We did not address this problem, as it would require pervasive changes in the VM system. We will quantify the performance impact of memlock contention in Section 4.2.3. The problem of memlock contention has been fixed in newer versions of IRIX.

Second, 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

TABLE 2. Description of 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.

Workload	System Parameters	Applications	SPU Configuration
Kernel Intensive	8 CPUs, 44 Mbytes memory, 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, 64 Mbytes memory, separate fast disks	Ocean (4 processors), 3 Flashlite, 3 VCS	2 SPUs: 1 SPU Ocean, 1SPU all Flashlite and VCS
Memory Isolation	4 CPUs, 16 Mbytes memory, 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 memory, shared HP97560 disk	Pmake and File copy	1 SPU pmake, 1 SPU file copy

the kernel to provide isolation between SPUs. This problem is similar to the well-studied priority inversion problem [SRL90]. The solution requires that a process blocking on a semaphore transfer its resources to the process holding the semaphore until the semaphore is released. Though we developed an implementation for this problem, it was not actually implemented because the priority inversion problem did not have a significant impact on performance in any of our workloads.

Third, updates from a processor to shared kernel structures will invalidate the data in all the other processors caches if it is there. This requires that the data be loaded from memory or another cache when it is needed again by the processors, leading to additional memory stall time that would not be seen in a system with a smaller configuration, especially in a uniprocessor system. Again, this and the previously described problems were noticed as a result of our attempts to provide performance isolation, but actually need to be addressed in the design of scalable operating systems for shared-memory multiprocessors.

4.0 Performance Results

To demonstrate the capabilities and the effectiveness of the performance isolation resource allocation model we run a number of different workloads on our implementation in the IRIX5.3 kernel. The workloads are summarized in Table 2. Each workload is run with three different multiprocessor resource allocation schemes as shown in Table 3. We compare the results of the performance isolation scheme with that of two others, a scheme with fixed quotas for each SPU and the regular SMP kernel with unconstrained sharing and no SPUs.The SMP kernel used for these experiments has been modified to include the semaphore fixes described in Section 3.4, and therefore has better performance than the standard IRIX5.3 kernel.

TABLE 3. The three different resource allocation schemes compared in the workload runs.

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

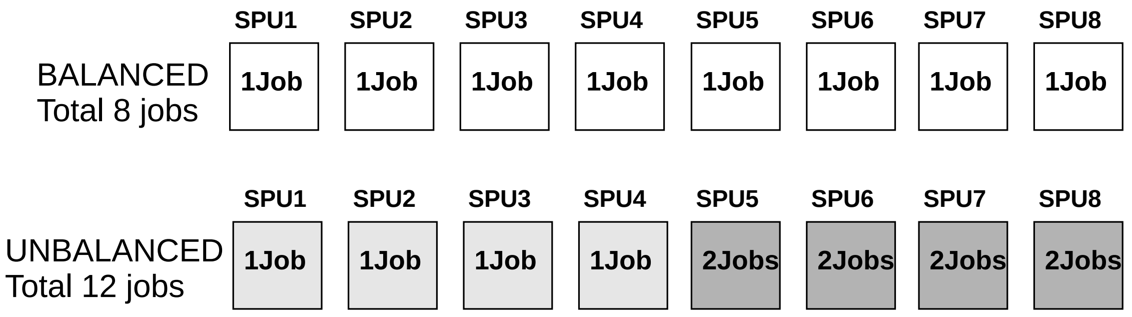


FIGURE 3. Balanced and Unbalanced configurations for PMAKE. The figure shows the distribution of jobs to SPUs in the balanced and unbalanced configurations for the PMAKE workload. The **BALANCED** configuration has eight jobs, one per SPU. The **UNBALANCED** configuration has 12 jobs. The lightly-loaded SPUs (1-4) have 1 job each, and the heavily loaded SPUs (5-8) have 2 jobs each.

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 [RHW+95], loosely modelled on 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.

4.2 Kernel Intensive Workload

The first workload consists of a number of pmake jobs as described in Table 2. 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. For the distribution of processes to SPUs we consider two different scenarios as shown in Figure 3. The first is a balanced configuration running eight jobs, one per SPU for performance isolation. The second is an unbalanced configuration with four SPUs (1-4) running one job each, and the other four SPUs (5-8) run two jobs each.

This workload will be used to demonstrate performance isolation for both processor and memory resources, and the effect of contention for kernel resources. We will show that performance isolation is able to both isolate SPUs from background system load, and allow the sharing of idle resources to improve the performance of SPUs under heavy load.

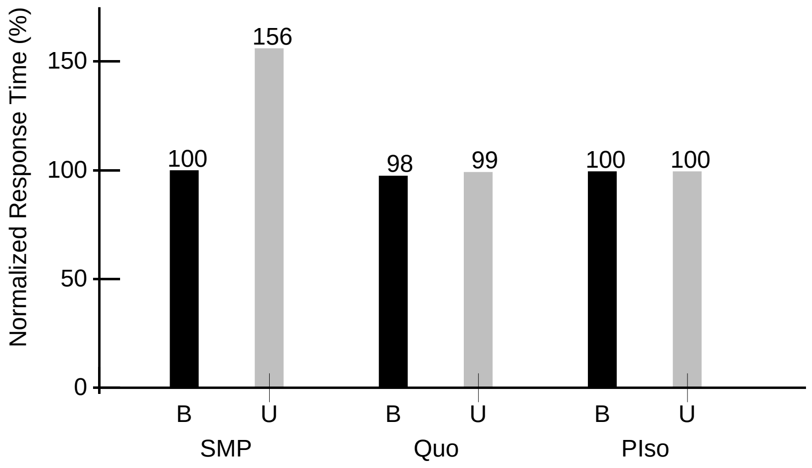


FIGURE 4. Effect of Isolation in PMAKE. Average response time for jobs in the lightly-loaded SPUs (1-4) for the balanced (B) and unbalanced (U) configurations normalized to the SMP time in the balanced configuration. Performance Isolation (PIso) and Fixed Quotas (Quo) are able to maintain performance as system load increases, whereas SMP is unable to do so.

4.2.1 Isolation

The Pmake workload is very kernel intensive, and has a lot of process creation and filesystem activity. The high kernel activity makes performance isolation difficult for this workload. We will first demonstrate how performance isolation is able to isolate SPUs from changes in background system load, by considering the performance of the jobs in SPUs one to four. These SPUs have only one job, and do not change their resource requirements when going from the balanced to the unbalanced configurations. Figure 4 shows the average response time for these jobs in the balanced and unbalanced configurations, normalized to the case of SMP in the balanced configuration.

In the regular SMP kernel the response time for the jobs 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 about the same share of resources. Therefore the SPUs that introduce two jobs (SPUs 5 - 8) increase the load on the system, and increase the response time of all jobs including those of the lightly-loaded SPUs (1 - 4). Both, Performance Isolation (PIso) and Fixed Quotas (Quo) by isolating SPUs from each other and allocating resources based on SPUs, are able to effectively maintain the performance of jobs in the lightly-loaded SPUs despite the increase in overall system load.

4.2.2 Resource Sharing

The next issue is what happens to the performance of the heavily-loaded SPUs (5-8) in the unbalanced configuration. We know that these jobs do well under SMP because they are able to take up more than their “fair share” of resources. The performance of the jobs in these SPUs is shown in Figure 5. The average response time is shown for each of SMP, Quo, and PIso normalized to the SMP performance in the balanced configuration.

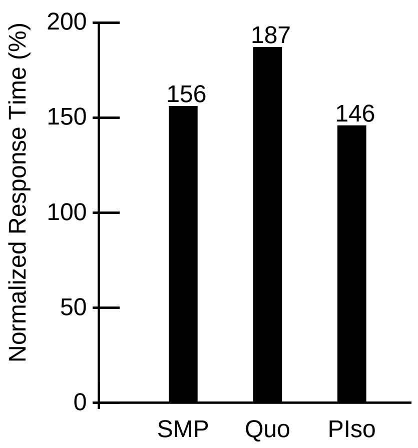


FIGURE 5. Effect of Resource Sharing in PMAKE. Response time for jobs in the heavily-loaded SPUs (5-8) for the unbalanced (12 jobs) configuration normalized to the SMP time in the balanced configuration. Performance Isolation (PIso) allows the heavily-loaded SPUs to use the idle resources in the system to improve performance. Fixed Quotas (Quo) does not allow this and so performance suffers.

The performance in Quo is worse than that of SMP. By imposing fixed quotas, Quo was able to isolate the lightly-loaded SPUs, but leaves resources idle after the jobs in these SPUs finish. The jobs in the heavily-loaded SPUs are unable to use these resource because of the fixed quotas. The performance isolation case in contrast is able to carefully share these idle resources, thus providing response time for the heavily-loaded SPUs that is comparable to the SMP case. Actually PIso does a little better because when the light-load SPUs finish early, they release memory that 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 memory till the end

4.2.3 Comparison with a Single Workstation

One of the goals of performance isolation was to not introduce much extra overhead, and to be roughly comparable in performance to a smaller machine corresponding to the resources allocated. For this PMAKE workload, the equivalent smaller machine would be a uniprocessor workstation. The equivalent single workstation numbers are not really done on a single workstation configuration because of the difficulty of specifying an equivalent memory configuration. Specifying an eighth of the memory would be wrong because the kernel takes up a fixed number of pages. These runs are done by setting up eight user SPUs, with only one SPU running the workload and the other SPUs idle. This configuration could be considered somewhat optimistic.

Figure 6 compares the performance of the single workstation (Ref) and the MP with performance isolation for the PMAKE workload. The performance of SPUs 1-4 is compared with running one job on the equivalent workstation, and that of SPUs 5-8 in the unbalanced case with two jobs on a workstation. Performance isolation is able to keep the performance of the individual jobs somewhat comparable to that of the equivalent individual workstation in the one job case, and performs better in the two job case by using idle resources on the system. The increase in response time in the one job case is 31%. As mentioned in Section 3.4, there are a number of

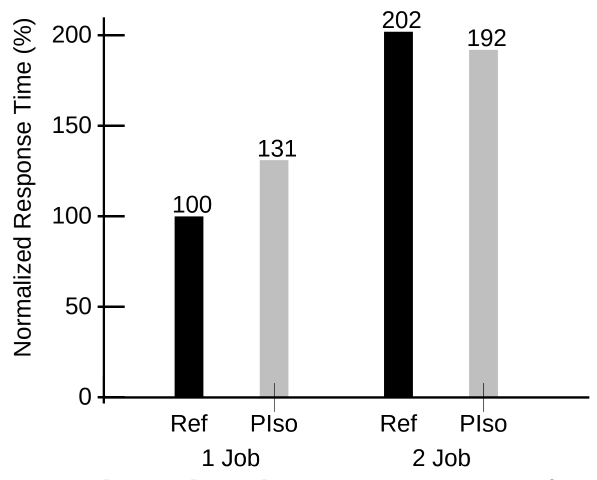


FIGURE 6. Comparison with a single workstation. Response time for the PMAKE workload with performance isolation (PIso) and an equivalent single workstation (Ref). The numbers for both 1 Job and 2 Jobs in an SPU and on a workstation are shown. The response times are normalized to that of 1 Job on a workstation. In the two job case PIso does better by using idle resources in the MP system.

reasons related to contention for kernel resources that can cause this overhead. It should also be noted that this workload is kernel intensive and so presents the worst side of this problem.

Figure 7 shows where the time is spent for all four configurations for the one job case. The first observation is that the overheads in the performance isolation case are the same as for the SMP case. No significant additional overhead is introduced by the mechanisms that provide performance isolation. Second, the increase in response time from Figure 6 is matched almost exactly by the increase in computation time here. No significant additional idle time is spent waiting for semaphores because our implementation was successful in removing the main causes of semaphore waits. Third, the increase in overhead from the single workstation to the multiprocessor is mostly because of two causes as discussed in Section 3.4, synchronization time (about 60%) and additional kernel stall time because of inter-processor communication (40%).

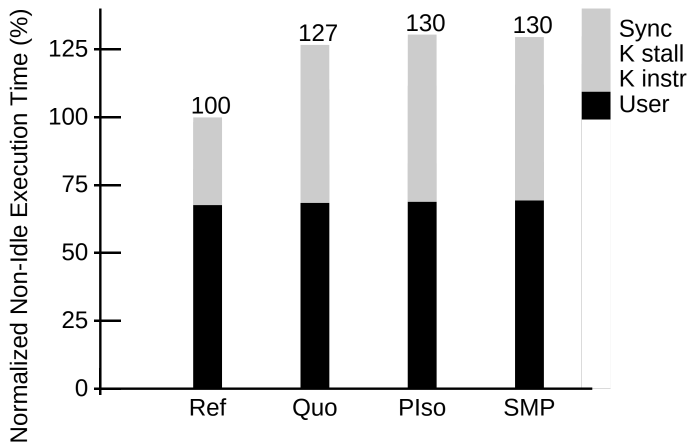


FIGURE 7. Non-idle execution times for PMAKE. The times for all cases are shown normalized to that of the appropriate equivalent single workstation reference case (Ref). Each bar shows User, Kernel instruction, Kernel stall and Synchronization time for that entire run. There is additional kernel stall time and synchronization time for all the MP schemes (PIso, Quo, and SMP) compared to the Ref, and this is not because of providing Performance Isolation.

Analyzing the synchronization time in more detail, we found that more than 80% of this time is spent contending for a single lock, memlock. This is a coarse lock that protects most of the important structures and flags in the VM system — the lists of free pages, the hash table that maps vnodes and file offsets to memory pages, the flags for each physical page frame descriptors, and a number of other smaller structures. A more scalable design would replace this lock with a set of finer grain locks.

We next analyzed the kernel stall time in detail by function, comparing the eight processor runs with the equivalent single workstation ones. About half the extra stall time is attributable to VM structures and functions, and about a third is because of filesystem and buffer-cache related structures and functions. This is not surprising because the Pmake workload, with many short-lived processes and much file activity, is very memory system and filesystem intensive. For the memory system, the contended structures are the heads of free page lists and hash tables. It is possible to restructure the VM to have less contention. This becomes critical when shared-memory systems move to a CC-NUMA memory architecture. It is somewhat harder to make a simple case for the filesystem structures as they truly need to be shared by processors. These synchronization and stall issues are general scalability issues, and are not particular to performance isolation. Solving the scalability problem, will also improve the MP performance with SPUs, and bring it much closer to that of the equivalent individual workstation. Some of the VM bottlenecks have been addressed in newer versions of IRIX.

4.3 CPU Isolation Workload

This workload described in Table 2 consists of compute-intensive scientific and engineering jobs with kernel time only at the start-up phase. The workload has a total of ten processes on eight processors, and will be used to demonstrate CPU isolation. There is adequate memory for all the application and so that is not an issue. For the performance isolation runs there are two SPUs corresponding to two users. One runs the four process Ocean, and the other three Flashlite and three VCS jobs. Each SPU is allocated four CPUs. The sharing policy is to share idle resources with the other SPU.

Figure 8 shows the results for this workload. Response time numbers are averages of all the jobs of a type normalized to the SMP case. In the SMP configuration all the processes are treated equally, therefore Ocean gets less than its “fair share” of CPU time and runs slower than in the other cases. Both fixed quotas (Quo) and performance isolation (PIso) are able to improve the response time of the Ocean processes greatly by isolating it within a SPU, preventing interference from the other applications.

The Flashlite and VCS processes are running in the heavily-loaded partition; six jobs on four processor. Fixed quotas (Quo) causes the performance of these applications to get worse. Performance isolation (PIso) by sharing idle resources is able to keep the response time of these

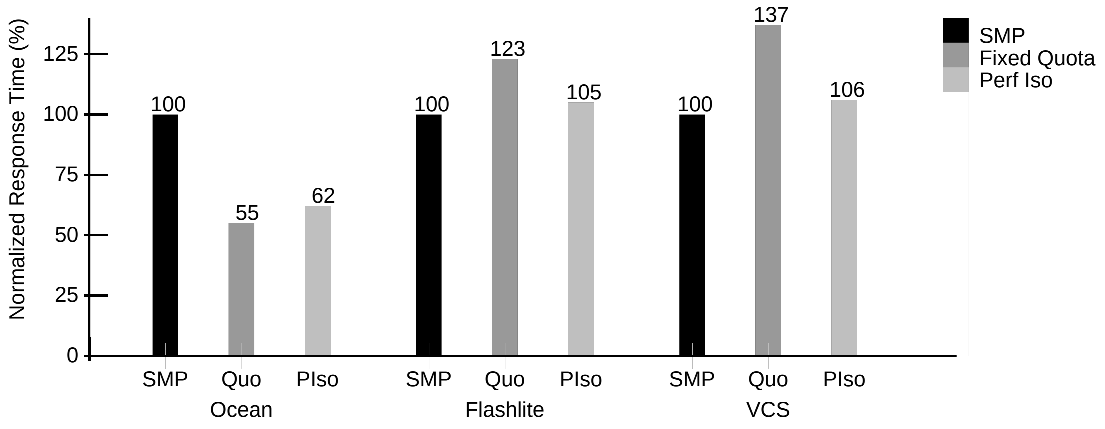


FIGURE 8. Response times for a compute intensive workload. The workload consists of a four-process parallel OCEAN, three Flashlite, and three VCS jobs. The system has eight processors and plenty of memory. For performance isolation, Ocean runs in one SPU (four processes on four processors) and all the 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.

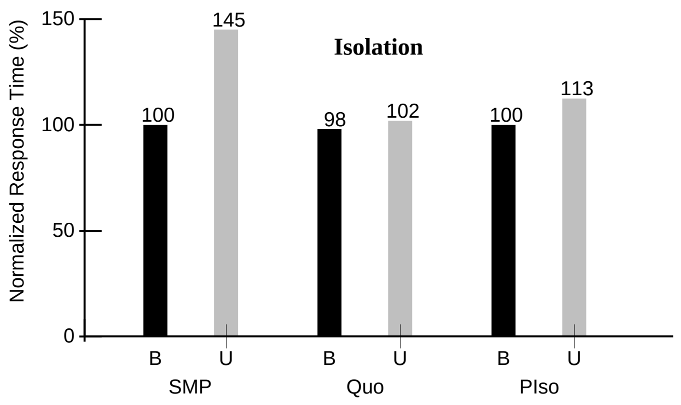
application comparable to the SMP case, while isolating the OCEAN processes from interference.

4.4 Memory Isolation Workload

This workload and the experiments are similar to that of the previous pmake workload, and is described in Table 2. In this case, the memory size is deliberately made small to clearly show performance isolation for memory. The memory is enough to run one job in each SPU, but leads to memory pressure in a SPU with two jobs. The balanced configuration has one job in each SPU, and the unbalanced configuration has two jobs in SPU2 and only one job in SPU1.

Figure 9 shows the performance of the different configurations. The figures are similar to the previous pmake workload. On the left we show the effect of providing isolation for SPU1 in the balanced and unbalanced configurations. Both performance isolation and fixed quotas are able to provide isolation to maintain performance as the background system load increases. SMP treats all processes the same, resulting in lower performance for the processes of SPU1. Looking at the processes of SPU2 in the unbalanced configuration, we see that fixed quotas results in poor performance. The loss in performance with fixed quotas is large because of the memory limitation. Performance isolation through the sharing of idle resource — memory and CPU in this case — delivers significantly better performance, close to the SMP case under heavy load.

One difference that is not visible in Figure 9 is that the response time for the light-load SPU with performance isolation is only 4% larger than the equivalent smaller machine, unlike 31% in the



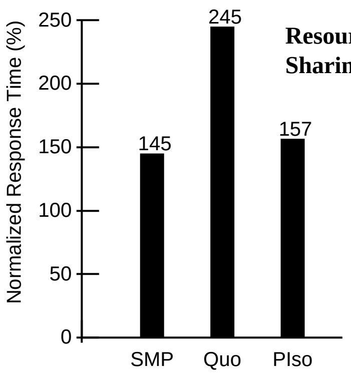


FIGURE 9. Performance Isolation for a memory-limited workload. The graph on the left shows the effects of providing isolation for SPU1 that runs 1 job in both the balanced and the unbalanced configuration. The graph on the right shows the effect of resource sharing for SPU2 in the unbalanced configuration running two jobs. The response time (latency) for all jobs are shown normalized to that of the balanced SMP case. Both performance isolation (PIso) and fixed quotas (Quo) are able to provide isolation against background system load. Only performance isolation is able to help the heavily-loaded SPU2, by sharing idle resources.

previous pmake workload. The effect of contention is much smaller because this is only a four processor MP, and the equivalent smaller machine is a two processor MP.

4.5 Disk Bandwidth Isolation Workload

Implementing performance isolation for disk bandwidth is more difficult than for processors and memory. Consider two possible ways that disks might be used by unrelated tasks. The first possibility is that the different tasks access different disks that are on the same SCSI controller, or maybe even different controllers. In this case the tasks would interfere only minimally by competing for bandwidth on the SCSI bus, but not for the same disk. This was the case for the previous three workloads.

The second possibility is that the tasks just share access to a larger disk instead of accessing different disks. This case is more difficult from the viewpoint of performance isolation. The additional seek time to move the disk head between two potentially different sets of disk sectors becomes an issue. Also a SPU that is using the disk less frequently has a greater likelihood of finding a request from other SPUs currently being serviced by the disk. The isolation mechanism can let this new request bypass currently queued requests for bandwidth fairness, but it still has to wait for the currently active request to complete.

To demonstrate the effect of performance isolation on disk bandwidth we do two sets of runs. To keep the experiment simple, the machine is a two way MP and the HP97560 disk model is used. To reduce the length of the simulation runs we use a scaling factor of two for the disk, i.e., the disk has half the latency of the regular model. The first set of runs 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. We consider three different policies for scheduling disk requests. The first is the standard head-position based scheduling as is currently in IRIX. The second is a strict performance isolation policy that ignores head position, and only strives to provide equal disk bandwidth to the SPUs. The third is the performance isolation policy described in Section 3.3 that gives weight to both isolation and the

TABLE 4. The effect of performance isolation on a disk-limited workload. The workload consists of two SPUs, one a pmake process and the other a large (20 Mbyte) file copy to the same disk as the pmake. There are three runs. First with the standard head-position based scheduling, next with isolation-only scheduling, and finally with the disk bandwidth performance isolation policy that takes into account both head-position and isolation. The response time and the average queue time per request for each job is given, along with the average queue length and average disk latency.

	Response time (sec)		**Avg. Wait Time (ms)**		**Avg. Queue Length**	**Avg. Latency (ms)**
Configuration	**Pmake**	**Copy**	**Pmake**	**Copy**		
Position only	15.7	12.8	94.3	38.4	5.1	10.6
Isolation only	9.2	15.8	23.4	63.5	3.7	11.6
Position & Isolation	9.6	15.6	23.0	75.4	4.3	11.4

head position. This policy tries to balance fairness of bandwidth and effective throughput to the disk. The results of the three cases are shown in Table 4.

The pmake makes a total of 300 requests to the disk, and these are not all contiguous as they represent multiple files. Also some of these are repeated writes of meta-data to a single sector. The large copy makes a total of 1050 requests to the disk. These are mostly contiguous sectors as it is reading and writing large files. There are multiple outstanding reads because of read-ahead by the kernel. There are writes because the buffer cache fills up. As a result of these contiguous reads and writes, the position-only schedule, locks out the pmake from the disk for long periods of time, resulting in a large response time for the pmake. This is the same result that users see when a large core file is dumped to a disk.

The performance isolation policy preserves fairness by considering both isolation and the head position when making a decision. The result can be seen in the significantly lower average wait time for the requests from pmake, as these requests are now not waiting for all copy requests to be processed. The response time for the pmake job is significantly improved by incorporating fairness into the scheduling policy. The copy job, as expected, does see a reduction in performance.

The strict isolation only policy that ignores head position is also able to improves 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 in general true, and completely ignoring disk-head scheduling could lead to reduced performance.

We use a different workload to 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 5. Once again both the isolation-only policy and the performance isolation policy improve the response time of the smaller copy by incorporating fairness into the scheduling decision. However for this workload, the latter policy by also incorporating head-position information is able to provide better response times for both processes as compared to the former policy. The

TABLE 5. The advantage of considering both head-position and fairness. This workload consists of two processes copying files, one a small 500Kbyte file and the other a larger 5 Mbyte file. There are three runs, first with the standard head-position based scheduling, next with isolation-only scheduling, and finally with our disk bandwidth performance isolation policy. The response time and the average queue time per request for each job is given, along with the average queue length and average disk latency. The performance isolation policy performs significantly better than the Isolation only policy in this case.

	Response time (sec)		**Avg. Wait Time (ms)**		**Avg. Queue Length**	**Avg. Latency (ms)**
Configuration	**Small**	**Big**	**Small**	**Big**		
Position only	0.93	0.81	155.8	12.1	5.17	6.4
Isolation only	0.56	1.22	68.9	23.7	3.26	8.2
Position & Isolation	0.28	0.96	31.9	16.6	3.1	6.62

average latency per request for the latter is about the same as the position-only scheduling policy, whereas by ignoring head-position scheduling the former pays almost a 30% increase in average latency.

4.6 Motivation Numbers Revisited

Having demonstrated the effectiveness of performance isolation for a number of workloads and different scenarios, we now revisit the experiments done in Section 1.1 that defined the problem and motivated the performance isolation model. Figure 10 compares the performance of the regular SMP kernel (SMP) with the performance isolation model (PIso). It is clear that the model has been successful at achieving its stated goals.

In the MP1 case, PIso achieves about the same benefit as SMP by being able to utilize idle resources from other SPUs. In the MP6, and MP4+D cases we see excellent isolation of performance, improving the response time by over 50% compared with the SMP kernel. The actual response time for the MP4+D shows an increase of 47% over the uniprocessor case, but in this case we have an increased load on the disk resource (accesses by the copy program), without any additional bandwidth resources.

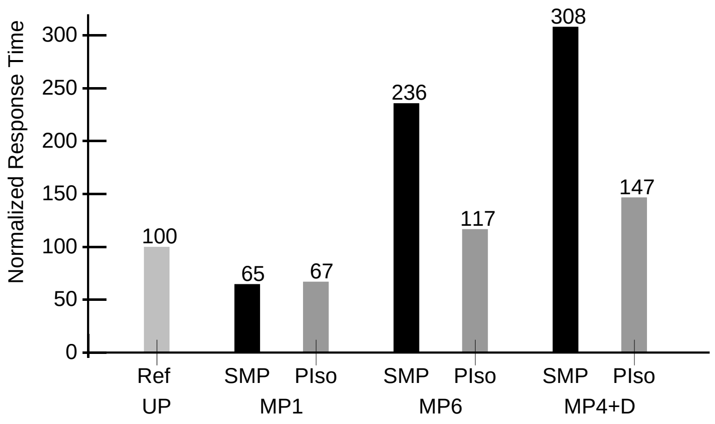


FIGURE 10. Motivation workload with and without performance isolation. The runs are the same as the ones described in Section 1.1. For each run there are two bars; with the performance isolation model (PIso) and with the regular SMP kernel (SMP).

5.0 Discussion

The main differences between our work and other related work is that our work specifically targets shared-memory multiprocessors while most other work has been for uniprocessors, we propose a formal framework to balance the twin goals of throughput and latency rather than just provide long-term fairness, and our work includes all the system resources that could affect performance including disk bandwidth.

[Wal95] shows different techniques for providing proportional-share resource management for a variety of computing resources, including CPU, memory, disk and network bandwidth. Their work analyzes the different techniques carefully for each resource, but does not consider a comprehensive model for multiprocessors that accommodates all the resources. They also only provide a real implementation for CPU time, and the analysis for other resources is done using simulations and not with real workloads. However, their techniques are interesting and could be used in place of the specific implementations that we chose for some of the resources.

There have been numerous proposals that implement a part of our model primarily for uniprocessors, providing fairness to individual users. Most of these only consider CPU scheduling [Hen84][KaL88][WaW94]. An extension to [KaL88], the SHAREII resource management tool [Sof96] also assigns quotas for 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 of dynamic allocation of pages to processes (not users), and a local page replacement policy is conceptually similar to our implementation for memory isolation in IRIX. [HaC] have a proposal for allocating memory and paging bandwidth to disk based on a market approach. They assume that there are enough processors, so that is not an issue, and their unit of fairness is individual processes. [Wal95] presents the only other work on fairness for disk bandwidth allocation. They discusses their funding delay cost model for scheduling disk requests, and through simulations show that it can achieve fairness for certain limited workloads. We consider our implementation that balances head position and fairness to be novel. 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 implement fairness for network bandwidth by changing the order of service from FCFS.

Our model is probably more similar to resource allocation schemes on mainframe (MVS) and minicomputer (VMS) [LeL82] multi-user systems than the current crop of UNIX based SMP operating systems. The Stealth Distributed Scheduler [KrC91] implements isolation goals similar to ours in the context of distributed systems, when scheduling foreign processes on a user’s workstation. They attempt to prevent the foreign processes from affecting the performance of the workstation owner’s processes for all the resources; CPU, memory, paging bandwidth, etc. They therefore concentrate only on isolation for the owner’s processes, and not on isolation and sharing for all SPUs as we do.

As processors get faster, shared-memory multiprocessors are moving from bus-based architectures to the CC-NUMA architecture. The performance isolation model will continue to be applicable to these multiprocessors. There are issues related to the implementation of the model that could impact data locality in the NUMA architecture. The space partitioning policy that we selected for CPUs will potentially provide better cache and memory locality than a time partitioning policy, because it explicitly restricts the processes to a subset of the processors. If a time partitioning policy is used, strong cache and memory affinity will be important to improve data locality [CDV+94]. On a bus-based shared-memory multiprocessor all pages are conceptually the same, and so we only need to maintain a count of pages used by a SPU. When moving to a CC-NUMA architecture, pages on each memory node are conceptually different. Closer co-ordination between the allocation of CPUs and the allocation of memory pages would be necessary to maintain good data locality.

6.0 Conclusions

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 belief that unlike workstations, SMP kernels on multi-user shared-memory multiprocessors cannot provide isolation of performance for a user. This had the implication that moving from workstations to compute servers was like taking a step backward to the old mainframe days. This work proves otherwise by showing that with an appropriate resource allocation model, a shared-memory compute server can provide workstation-like isolation in a heavily loaded system and SMP-like latencies in a lightly-loaded system.

We propose a new resource allocation model for multiprocessor compute servers to get the best of both worlds: the performance isolation of workstations, and the resource sharing of SMPs. We provide a framework to analyze the basic trade-offs in implementing this model. We show that the implementation complexity of this model is manageable by describing our prototype implementation in the IRIX5.3 kernel from Silicon Graphics. Our implementation manages the CPU, memory and disk bandwidth resources of the system. Running a diverse set of workloads on this kernel implementation, we demonstrate that the model is robust across a range of workloads and resources, and is extremely successful in providing workstation-like latencies by isolating performance under heavy load, and SMP-like latencies under light load. Given the results that we demonstrate, this resource allocation model should be implemented in the operating systems of shared-memory multiprocessor compute servers to provide users of such machines with the performance that they expect.

References

[ACP+94] T. Anderson, D. Culler, D. Patterson. A Case for NOW (Networks of Workstations). Presented at *Principles of Distributed Computing*, August 1994.

[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.

[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.

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

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

[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, In *IEEE Computer*, March, 1982

[Ous82] J.K. Ousterhout. Scheduling Techniques for Concurrent Systems, In *The 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, J.P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. In *IEEE Transactions on Computers*, pages 1175-85, September 1990.

[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.

[WaW94] C.A. Waldspurger and W.E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the first symposium on Operating Systems Design and Implementation*, November, 1994.

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