2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, March 1990.

[Sha90] L. Sha, R. Rajkumar, J. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization", IEEE Transactions on Computers, Vol. 39, No. 9, September 1990.

[TuGu89] A. Tucker, A. Gupta, "Process Scheduling Issues for Multiprogrammed Shared Memory Multiprocessors", ACM Symposium on Operating System Principles, December 1989.

[VaZa91] R. Vaswani, J. Zahorjan, "The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors", ACM Symposium on Operating System Principles, October 1991.

[VaRo88] M. Vandervoorde, E. Roberts, "WorkCrews: An Abstraction for Controlling Parallelism", International Journal of Parallel Programming, August 1988.

[Wald94] C. Waldsburger, W. Weihl, "Lottery Scheduling: Flexible Proportional-Share Resource Management", USENIX Symposium on Operating Systems Design and Implementation, November 1994.

[Zaho90] J. Zahorjan, C. McCann, "Processor Scheduling in Shared Memory Multiprocessors", SIGMETRICS May 1990.

[Zaho91] J. Zahorjan et al., "The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Multiprocessors", IEEE Transactions on Distributed Systems, April 1991.

[Bach86] M. Bach, “The Design of the UNIX Operating System”, Prentice Hall, 1986.

[BaWa88] J. Barton, J. Wagner, “Beyond Threads: Resource Sharing in UNIX”, USENIX Winter Conference, February 1988.

[Blac90] D. Black, “Scheduling Support for Concurrency in the Mach Operating System”, IEEE Computer, May 1990.

[CaSm89] D. Callahan, B. Smith, “A Future-Based Parallel Language for a General Purpose Highly Parallel Computer”, Proceedings of the 2nd Workshop on Languages and Compilers for Parallel Computing, MIT Press, Cambridge, MA, 1989.

[Cray85] Cray Computer Systems, “Multitasking User Guide”, Technical Note SN-0222, January 1985.

[DiIy90] R.T. Dimpsey, R.K. Iyer, “Performance Degradation due to Multiprogramming and System Overheads in Real Workloads”, Proceedings of the 1990 ACM International Conference on Supercomputing, Amsterdam, Holland, June 1990.

[Drav91] R. Draves, et al., “Using Continuations to Implement Thread Management and Communication in Operating Systems”, ACM Symposium on Operating System Principles, October 1991.

[Edle88] Edler et al., “Process Management for Highly Parallel UNIX Systems”, USENIX Workshop on UNIX and Supercomputers, September 1988.

[EaZa93] D. Eager, J. Zahorjan, “Chores: Enhanced Run-time Support for Shared-Memory Parallel Computing”, ACM Transactions on Computer Systems, February 1993.

[GiPo92] M. Girkar, C.D. Polychronopoulos, “Functional Parallelism in Ordinary Programs”, IEEE Transactions on Parallel and Distributed Computing, March 1992.

[Leff89] S. Leffler, M.K. McKusick, M.J. Karels, J.S. Quarterman, “The Design and Implementation of the 4.3 BSD UNIX Operating System”, Addison-Wesley, 1989.

[LoGl87] S.P. Lo, V. Gligor, “A Comparative Analysis of Multiprocessor Scheduling Algorithms”, 7th International Conference on Distributed Computing Systems, September 1987.

[Poly89] C.D. Polychronopoulos, “Multiprocessing vs. Multiprogramming”, Proceedings of the 1989 International Conference on Parallel Processing, St. Charles, IL, August, 1989.

[Poly90] C.D. Polychronopoulos, “Control Flow and Data Flow Come Together”, Technical Report, Center for Supercomputing Research and Development, University of Illinois, November 1990.

[Poly94] C.D. Polychronopoulos, N. Bitar, S. Kleiman, “Nanothreads: A User-level Threads Architecture”, Technical Report #1295, Center for Supercomputing Research and Development, University of Illinois.

[Scot90] M. Scott et al., “Multi-model Parallel Programming in Psyche”,

provided user-level schedulers to schedule the process's threads. To facilitate this, a high speed two-way communication mechanism between the user-level scheduler and the kernel scheduler, similar to the PRDA, will be provided. This arena will allow the user-level scheduler to communicate its needs to the kernel and allow the kernel scheduler to indicate certain events such as processor allocation, processor preemption, and thread blocking to the user-level.

The new scheduler will also provide a new batch mechanism for scientific users. The existing batch policy of only scheduling a process when no real-time or timesharing band process is runnable will be maintained. A new band will allow users to utilize their machine in a fashion that provides greater importance to specific batch jobs. In particular, in exchange for information about how much cpu time, memory and I/O bandwidth a job requires and a desired completion time, the kernel scheduler will allocate processors to optimize job completion time rather than throughput, providing the requests are reasonable (possible). Requests that cannot be satisfied because they are unreasonable or conflict with previous requests will be rejected. Processes that exceed their specified user cpu time will be relegated to the timesharing band to fend for themselves. This mechanism allows scientific users to achieve a predictable number of runs of a large parallel application within a particular timeframe.

Finally, the new scheduler will provide a *memory affinity* scheduling mechanism to better support new machines with NUMA memory characteristics.

6. Concluding Remarks

The scheduling system described in this paper was introduced with Version 5 of the IRIX operating system, initially shipped as IRIX 5.0 and 5.0.1 on the Challenge line of multiprocessors in March of 1993. This was followed in August of 1993 by IRIX 5.1 on the Indy desktop workstation. As of the date of this writing, this scheduler is running on over 2000 Challenge multiprocessors and 10000 uniprocessor Indy workstations and runs on all existing MIPS-based Silicon Graphics computers including the PowerSeries multiprocessors. It has satisfied the needs of Silicon Graphics real-time, media, scientific, database, and general-purpose customers.

REFERENCES

[AnLa89] T. Anderson, E. Lazowska, H. Levy, "The Performance Implications of Thread Management Alternatives for Shared Memory Multiprocessors", IEEE Transactions on Computers, December 1989.

[AnBe91] T. Anderson, B. Bershad, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism", ACM Symposium on Operating System Principles, October 1991.

gang timeslice. On expiration of the gang timeslice the gang becomes *inactive* and gang members are captured and held by the gang squeue until shutdown is complete. When the inactive state is entered the gang control block is moved to the end of the gang queue allowing round-robin scheduling of gangs.

4.3.3 The *Inactive* State

In this state the gang squeue collects members of the gang. This happens quickly since the timeslice has been carefully controlled. Within the precision of the IRIX fast timeout mechanism (typically 1 ms), all gang members will be queued on the gang control block. Once all gang members have been captured the gang enters the *queued* state.

5. Future Work

The current IRIX Version 5 scheduler uses a decay-usage priority mechanism in the timesharing band in order to select the next process to dispatch. While priorities in the real-time band indicate a strict relative order of importance of processes, priorities in the timesharing band are intended to be an indication of how much cpu time processes acquire relative to each other. The priorities are, therefore, decayed with cpu time accumulation in order to approximate this behavior. This is an *ad hoc* approach at best. The next generation IRIX scheduler will treat the timesharing band differently. In particular, all priorities will be non-degrading and in the timesharing band the priorities will be a precise indication of how much cpu time processes will accumulate relative to each other. The scheduler will implement a *space-sharing earnings based* policy where the priority of a process will indicate the rate at which cpu time is accumulated [Zaho90][Poly94]. Processes with high earnings will be scheduled ahead of processes with lower earnings and processes will be charged for their cpu usage when it occurs. The earnings value is an accurate indication of what a particular process's *fair share* of the machine, denominated in cpu microseconds, is at any point in time. This value will feed into an additive function whose result will be the final scheduling criterion. This function will take into account cache affinity, interactive response, boosts for new jobs and other factors - the original earnings will remain unchanged pending cpu usage. This mechanism allows for flexibility in short term scheduling decisions in order to improve throughput or response time, while maintaining fairness over the long term.

In addition, the new scheduler will provide support for user-level scheduling of threads [AnBe91][Poly94]. The kernel scheduler has little understanding of an application's topology and cannot be all things to all processes. As such, the scheduler will be a resource arbiter, allocating processors to processes (effectively scheduling single-threaded or share group processes) and allowing multi-threaded processes with user- or compiler-

the gang squeue there is currently no way to change their settings. Future versions of IRIX may allow this.

Once enough processes from the share group are queued the gang can become *active* if the lowest priority member of the gang has a higher or equal priority to the highest priority process on the time-sharing squeue. This ensures that time-sharing processes are not starved by the gang squeue; the gang squeue will not starve itself since it ages the priority of the processes on its queue every second. The priority check is made using the time-sharing squeue pointer saved in the *Runq* data structure.

4.3.2 The *Active* State

When a gang becomes active the gang control block is moved to the front of the gang squeue so that it is checked first on each dispatch cycle. Gangs are not necessarily activated in order: for instance, if a gang has become *inactive* a successor gang on the list will be checked. As soon as the first gang enters the *queued* state it would preempt the processors from a previously activated gang, causing starvation. Upon activation, the gang squeue initializes a time slice counter to zero and dispatches the first process on the gang control block process queue. From this point until the timeslice expires the gang squeue will dispatch processes from the gang to any eligible processor. Queue ordering effectively boosts the gang priority for a short period of time, allowing parallel dispatch of the gang members as processors perform a dispatch cycle.

Since there is no knowledge of when a processor may request a dispatch cycle it may take up to the default timeslice (30 ms) before another processor scans the gang squeue once the gang has been activated. This can subvert the intended parallelism. One solution to this problem is to make the gang timeslice significantly larger than normal so that gang members actually run in parallel for a useful length of time. However, since it is important to preserve normal behavior of other processes on the system this is unacceptable. Another solution would check for active gangs on each UNIX clock tick on every eligible processor. The current process would be preempted and a dispatch cycle started immediately. This ameliorates the problem but up to 10 ms may still pass before the gang is running fully in parallel and thus the startup window must be made even tighter.

To that end, when the first gang member is dispatched the gang squeue scans the processors available to the gang squeue (up to the number of queued gang members) and each one that is running a process of equal or lower priority to the gang is interrupted to request a dispatch cycle. This closes the start-up window to a few tens of microseconds. While the gang is active, processes in the gang come and go from the gang control block queue continuously. For instance, it may be necessary to service a page fault which may cause one of the processes to block briefly. Once requeued it will be dispatched immediately. Each time a process is dispatched from the gang its timeslice is set to the amount of time left in the

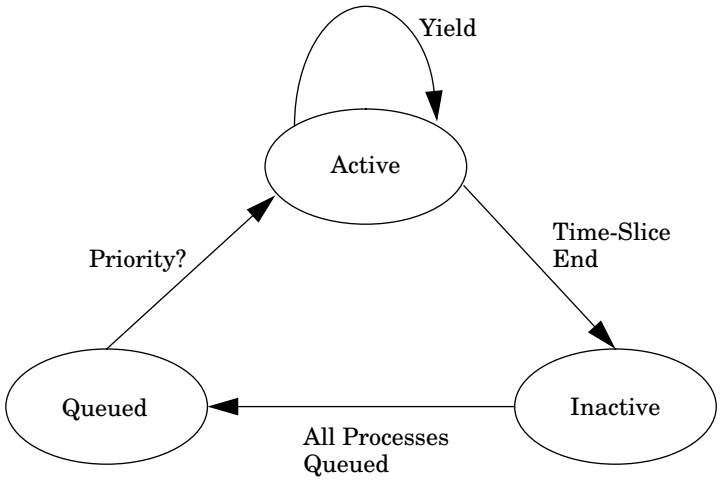

FIGURE 8: Gang State Diagram

The gang control block includes a pointer to the share group control block which is used for access to share group state. The gang squeue is actually formed from a list of these control blocks; each control block in turn acts as a queue for the processes in the gang. The gang squeue treats gang control blocks much as other queues treat individual processes.

4.3.1 The *Queued* State

In the *queued* state, the gang control block exists and processes which belong to the gang are captured after yielding and placed on the end of the queue associated with the corresponding control block. If the gang has returned to *free* mode then the gang squeue will reject gang members. The gang control block continues to exist for the life of the share group, however. The *equal* and *single* modes are handled in the squeue `_putq` routine. In *equal* mode, if a process from the gang is currently running, no other process from the gang will be dispatched; otherwise, the first queued member of the gang is dispatched, resulting in a round-robin effect. In *single* mode, if the master process is on the queue, then it is dispatched; otherwise the gang is ignored. If the gang is in *gang* mode then the more complex mechanisms of the squeue are invoked. Two parameters control process dispatch. The first indicates the minimum number of processes which must currently be queued on the gang control block before activating the gang and is set to one by default. The second indicates the maximum number of processes to run in parallel. This is set to the minimum of the number of processes in the share group and the number of processors available to the gang squeue by default. Although these variables are used by

4.3 The *Gang* Squeue

The gang squeue is used to schedule multiple related processes as if they were a single unit, that is, to schedule multiple processes to processors at the same time. A *gang* is a group of shared processes (a *share group)* derived from a common ancestor (the *master* process) for which *gang scheduling* has been requested. Given the fully distributed nature of the scheduler this is not as easy as it first appears. The various gangs that may be active must be round-robined and other work in the system must not be significantly disrupted by the presence of a gang.

Version 4 of IRIX included a different implementation of gang scheduling. This new version maintains the semantics and user-level interface of that version while contributing a completely new implementation. This was necessary for several reasons. IRIX Version 4 supported only the PowerSeries multiprocessor systems which scale to 8 processors. Users would run only a few (mostly one) gangs and little provision for impact on other work was taken into account. With the Challenge series of machines many gangs may be active at once in addition to typical timesharing, batch, deadline, and realtime processing. The Version 4 mechanism also relied strictly on probability for getting the members of the gang to run in parallel and this probability would decrease in a busy system.

There are four gang scheduling modes that may be requested by the process: *gang*, *free*, *single* and *equal*. By default, a share group is in the *free* mode. As long as the group remains in free mode the processes are ignored by the gang squeue. In *single* mode the master process of the group is dispatched normally while all other group members are blocked. In *equal* mode the members of the group are scheduled in a round-robin fashion with only one group member executing at any one time. In *gang* mode the processes in the group are dispatched in parallel.

A compiler generating parallel code breaks the execution up into a sequence of parallel and serial sections. For instance, the setup for a Fortran DO loop would be done serially by the master process, the proper number of threads would then be released to run in parallel to perform the work of the loop and finally cleanup and setup of the next loop would again be done by the master process. This means that the usual usage of the scheduler would be to flip between *gang* mode and *single* mode quite often. In fact, this is the preferred usage since it avoids having the other threads simply spin waiting for the serial section to complete - instead, they are held back by the scheduler until it is appropriate for them to run.

A gang exists in one of three states: *queued, active*, and *inactive*. Only if there are gangs in the *active* state will the squeue attempt to dispatch processes from the gangs. Figure 8 shows the simple state diagram.

The actions in each of these states are described below. The algorithms described rely on the information contained in a *gang control block*, which is automatically created when the share group leaves *free* mode for the first time.

size and added to the amount already allocated. If that amount exceeds the limits, the deadline is rejected. This algorithm treats the scheduler as if it had its own deadline and allocates time to processes from its own allocation.

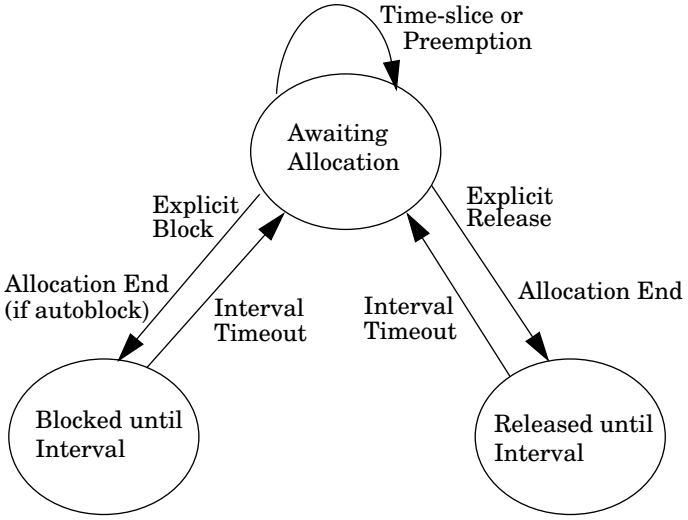

FIGURE 7: Deadline State Diagram

The squeue maintains the list of processes to run as a single queue, ordered on time-to-deadline. Dispatching of a process can then simply take the first on the queue. Each process is allowed a maximum time to execute limited by the allocation still undelivered and the default system time-slice interval meaning deadline tasks will execute in a round-robin fashion. The squeue will set a timeout for the undelivered allocation so that it regains control of the process when the allocation is finished. If there is still time left in the interval the process is released to the time-sharing squeue (or other squeues, as specified by the process) for ongoing execution. When a process is released in this fashion the squeue sets another timeout such that it can regain control of the process when a new interval begins.

The process may request certain variations to the above scenario. For example, it may request that when the allocation is finished that the process simply not run until the next interval begins. Such behavior crosses interval boundaries and persists until cancelled. Another alternative is for the process to request that it be blocked until the next interval or that any remaining allocation be released until the next interval begins. These requests only apply to the current interval. Figure 5 shows the state diagram that applies to a process in the deadline squeue.

another local queue quickly. Carefully constructed pathological cases can exact poor performance from this scheme, but in all cases examined so far this algorithm provides good scaling up to the 36 processors in the Challenge design.

4.2 The *Deadline* Squeue

The deadline squeue provides periodic deadline scheduling services to a process. A periodic deadline consists of two values: the interval (or period), which is the basic clock, and the allocation which is the guaranteed amount of execution time desired within each interval. The deadline squeue will ensure that the allocation is given to the process before the end of each interval. There is no assurance that the allocation will be delivered at any particular point in the interval nor that the allocation will be delivered as a single chunk of time. Delivery before the next interval begins is the only guarantee. Figure 7 illustrates this.

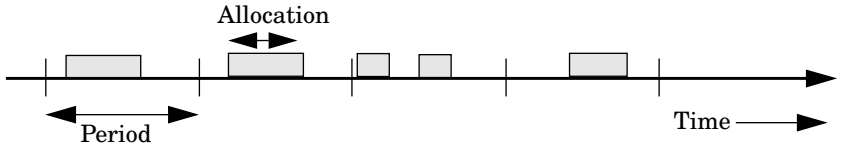

FIGURE 6: Deadline Scheduling

This style of deadline scheduling is useful for various continuous media handling applications such as constant-bit-rate delivery of video (e.g., MPEG at a 1.5Mbit/s rate), continuous audio and others. In essence, while the function of realtime scheduling is to manage latency and response time, the function of deadline scheduling is to manage throughput. Since these two goals tend to be opposed for any one process they cannot both be used at the same time (it is easy to see from the global scheduling policy that real-time processes have higher priority than deadline processes).

Since the deadline squeue guarantees an allocation, it must keep a notion of how much execution is available overall and enforce limits on allocation. Three limits are defined: `runq_dl_refframe`, `runq_dl_nonpriv`, and `runq_dl_maxuse`. The runq_dl_refframe variable specifies the largest period which can be requested, runq_dl_nonpriv specifies the maximum amount of processor time which can be allocated by non-privileged users and runq_dl_maxuse specifies the overall maximum which can be allocated. The amount of time available is calculated based on the reference frame size multiplied by the number of processors which can schedule deadline processes. When a request is made for a new deadline the requested interval must be less than or equal to the frame size. The allocation is scaled up by the ratio of the requested interval to the frame

20 millisecond range[1]. Given this, *every* process on the local queue is examined on every dispatch cycle. Since the affinity check is cheap, affinity is checked for every process and those not selected but whose affinity is expired are moved to the global queue immediately. Note that a processor only examines its own local queue; this implies that the queue data will tend to be exclusively owned by the processor cache and immediately available.

This solution implies, however, that the maximum time that a process may be blocked from running on a local queue will be the maximum time-slice that is allowed - typically 30 milliseconds in IRIX by default. With modern high-speed processors, 30 milliseconds is a very long time and can waste much of the system throughput. To address this issue the scheduler takes advantage of the well-known UNIX clock-tick mechanism which is 10 ms in IRIX. The scheduler has a global entry point which is called on every clock-tick and the ordered squeue supplies the `_timeq` method to be invoked when this occurs. When a clock-tick occurs for the processor it examines the local queue and moves any expired affinity jobs to the global queue making them visible to other processors for dispatching. This reduces the load-balancing penalty to the basic clock frequency as in other implementations of UNIX.

Unfortunately, it is possible for a processor to get limited to only examining its local queue. For example, consider a situation where several processes run for a short period, are briefly blocked, and then run again. A processor ends up running all these processes and, because of affinity, they all remain tied to that processor and its caches. This processor may never examine the global queue and higher priority processes may fail to dispatch while low priority work continues unabated.

To solve this problem the priority of the process that has been selected from the local queue is compared with the highest priority process on the global queue. If the global queue has a higher priority process it is selected instead without perturbing the local queue.

The architecture now provides a reasonable balance between single queue and distributed queue characteristics. Processes initially arrive on the global queue - some processor chooses the process and runs it giving the process affinity for that processor. If the process is CPU bound it will queue on the local queue of that processor pending dispatch. If it is I/O bound it is likely that the process affinity will expire and it will be queued on the global queue when it is unblocked. Essentially, CPU bound work will be distributed among various local queues while I/O bound work will be queued on the global queue and be immediately serviced with high probability. Load balancing will occur in parallel since expired processes will tend to move from one local queue, through the global queue, and onto

---

1. It is also assumed that it is atypical for a large number of processes to be restricted to a particular processor.

how often to examine other processors' queues will require walking a fine line between load balancing and queue interference. Furthermore, with large numbers of processors the question of which queues to look at becomes significant. In general, this type of solution will result in exactly the problem that the design was attempting to address.

The Ordered squeue addresses these problems by combining affinity management and distributed queueing. This algorithm is used only for ordered squeues which are non-realtime and will have heavy traffic (*Kernel, Time-Share, Batch*). Each processor is allocated a local queue on which processes with affinity to that processor will be queued[1]. All other processes are queued on the queue contained within the squeue object. Figure 6 illustrates this.

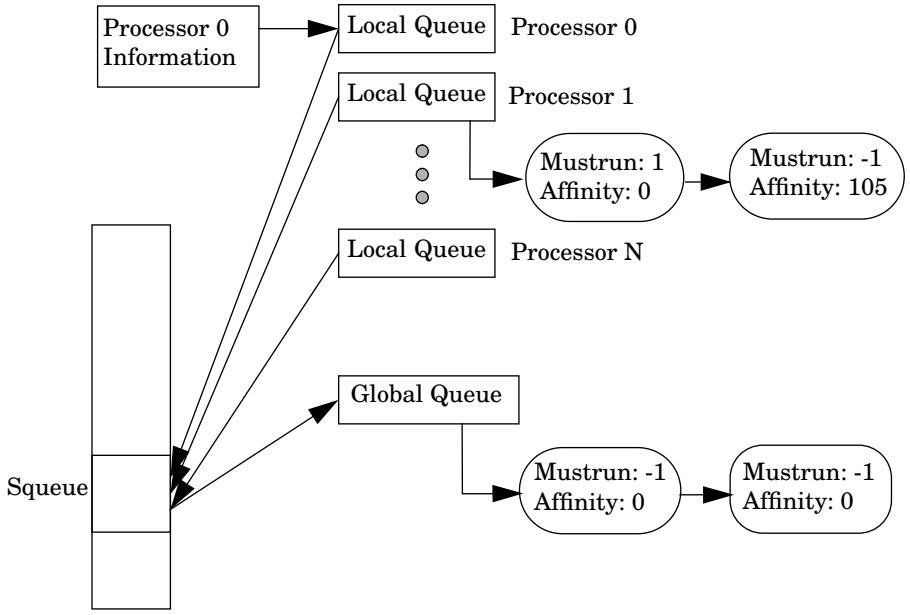

FIGURE 5: The Distributed Version of the Ordered Queue

When dispatching from an squeue the processor first examines the local queue; if no runnable processes are found then the global queue is examined.

It is assumed that the local queues never get very long. This is reasonable as the maximum affinity value for a process is typically in the 10-

1. Restricted processes are also queued on the local queue as there is no reason to have any other processor look at them. This allows optimization of performance in certain applications.

Gang: A gang squeue maintains a queue of gangs, which are groups of shared processes (*sproc(2)*) [BaWa88] descended from a common parent. In principle, the members of a gang share an address space and must physically run in parallel to achieve good performance. This squeue attempts to run the processes of a gang in parallel while maintaining fairness among gangs and with other processes on the system. *(Gang TS, Gang Batch)*.

4.1 The *Ordered* Squeue

The most heavily used squeue type is the *ordered* squeue. A modern shared-memory symmetric multiprocessor of the SGI Challenge class presents significant obstacles to efficient process dispatching. All communication between processors is handled by moving cache lines of data which are 128 bytes in length. Unlike the previous SGI PowerSeries multiprocessors there is no separate synchronization bus, so each synchronization transaction requires a complete memory bus cycle. Efficiency is improved due to the shared-read behavior of the cache system in that waiting on a spinlock is all contained within the processor-cache subsystem. However, when another processor invalidates the line by freeing the spinlock every processor waiting for that lock will immediately attempt to re-acquire the cache line causing a burst of bus traffic that degrades performance.

As the number of processors increases, this problem is exacerbated. In a busy system the scheduler is typically the most often executed piece of code and therefore its spinlocks are the most heavily used. If a single queue was used for all work, the multiprocessor performance of the system would degrade quickly. If cache affinity management is imposed on this structure the situation becomes much worse as a processor may have to examine many processes before finding a process which it can run and, in fact, some processors may not dispatch any process on a scan because of affinity for other processors. Empirical evidence shows that performance of the scheduler begins to *degrade* above eight processors when a simplistic queueing scheme is used even though the algorithms were designed to be independent of the number of processors.

The obvious solution to these problems is to use distributed queueing - allocating a queue for each processor. Distributed queueing has its own set of well-known problems especially in the area of load-balancing. For a general-purpose multiprocessor, automatic load-balancing is an important feature and distributed queuing makes this quite difficult. One solution is to decide the processor a process will run on at queue time. Since it is not known *a priori* how long a process will execute, many processors will invariably idle while others have long queues of work waiting. This situation can be ameliorated by allowing a processor with an empty queue to service other processor queues. Unfortunately a new set of problems are created since interference between processors will arise and the question of

4. Squeues

The *runqueue* is composed of a list of *squeues*. The scheduler includes a small internal table which describes the squeues to be created, their types, and other ancillary information. This table is the main point at which the global policy of the scheduler can be modified. It is read when the scheduler is first initialized and the information is used to build the runqueue and each of the squeues which comprises it. Conceptually, each squeue is an independent object with the following methods: `_putq`, `_getq`, `_timeq`, `_exitq`, `_forkq`, `_delq`, `_infoq`, `_endshaddrq`, `_joinq`, and `_leaveq`. Initialization consists of plugging the appropriate entry points and calling the `_joinq` method to allow the squeue to initialize itself. In a multiprocessor, additional processors may come and go arbitrarily (from the scheduler's point of view). Each new processor calls the `_joinq` method for each squeue which allows the squeue to update any internal state that is processor dependent. Conversely, a processor which is ceasing scheduling will call the `_leaveq` method for each squeue. Externally, actions such as enabling or disabling processors, restricting or isolating processors, or binding queues to processor sets will cause appropriate calls to the `_joinq` or `_leaveq` methods to allow state information to remain accurate.

For example, each squeue maintains a concept of the number of processors available to the squeue and the number which are implicitly available for process dispatching. If the implicitly available processors fall below a certain level it may be appropriate to provide an alternative scheduling behavior. In the gang scheduling squeue, a new gang will not be started if the number of available processors is less than the number of members of the gang.

For all the methods in an squeue, the global framework provides a single entry point called from other parts of the kernel. When invoked, this entry point in turn calls each squeue that has defined a method for that entry. The *fork* entry point, for example, first allocates the scheduler information data structure for the process and then calls each squeue `_fork` method in turn.

There are three basic types of squeue object defined. Each of these squeue types will be described in detail later in the paper. The seven configured squeues are derived from these three types. The types are:

Ordered: This is the simplest type, maintaining a single, priority ordered queue. On a multiprocessor this type applies affinity checks to each process. In addition, an ordered squeue may be configured as a distributed queue. *(Kernel, Real-Time, Time-Share, Batch).*

Deadline: A deadline squeue maintains a single, time-to-deadline ordered queue. *(Deadline).*

time has been spent running other tasks, then the process may be dispatched anywhere, otherwise the scheduler will only make it runnable for the processor it last ran on. This strategy makes affinity independent of how long a process was waiting or queued; on a moderately loaded multiprocessor it is quite possible that cached data may stay useful for many seconds.

Each scheduling squeue applies affinity as needed. The global framework provides a simple pair of routines for managing affinity: *set* the affinity value, which is called when a process yields the processor; and *check* the affinity value, which is called before every *runnable* check on the process. When checking affinity, this function will not return *OK* unless the affinity has expired or the checking processor is the last processor the process ran on. Cache affinity must be checked at this time because it is a constantly degrading value - the expiration of cache affinity should be noticed as quickly as possible to achieve good load-balancing.

Currently, affinity only lasts across a single trip to the runqueue. It is possible to carry affinity information across multiple trips, but that has not yet been implemented. It is not clear that the incremental gain in performance is worth the additional overhead, especially without hardware support for determining cache occupancy.

3.5 Runnable

There are a number of conditions which determine whether or not a process can be dispatched at any given instant in time. These conditions may change arbitrarily outside the control of the scheduler - thus, they must be checked each time a process becomes a candidate for dispatching. Each squeue typically requests the runnable state of several processes each time it is scanned.

Grouping of conditions is supported using *disciplines*. A discipline is defined to the scheduler as a callout routine whose return value specifies whether the process can run or not. The scheduler is set up to accept an arbitrary number of disciplines, however only a few are currently defined. In fact, only the *gfx* discipline, which checks whether or not a process can access the graphics head, is currently handled this way. Two other implicit disciplines are subsumed into the scheduler for optimal performance since the checks are almost always performed: normal UNIX handling (swapped processes, user areas, etc.) and processor set checking.

When a process is queued, the global framework initializes a bit vector indicating which disciplines are active for a process. In this case, the UNIX discipline is not included as it is always checked. For a normal UNIX process which is not bound to a processor set and is not a graphics process, this bit vector is zero. Each time the runnable check is performed, the heavier-weight checking of *gfx* and processor sets is only performed on exception (i.e., if the bit vector is non-zero), improving performance overall.

some point the amount of time it takes to re-fetch lost cache data will be less than the time to handle cache affinity.

The amount of good data associated with a process in a particular cache degrades as that processor performs other work. Depending on how much other work has been performed, it may or may not be advantageous to force a process to dispatch on the last processor it ran on. There is currently no hardware support for knowing the actual cache occupancy for a particular process, so it is estimated. The cache size for each processor and the state times for each process are known and thus the cache miss overhead can be calculated. Using these parameters and the running time, cache occupancy can be estimated. For simplicity, the scheduler assumes a linear relationship between running time and cache occupancy. Figure 4

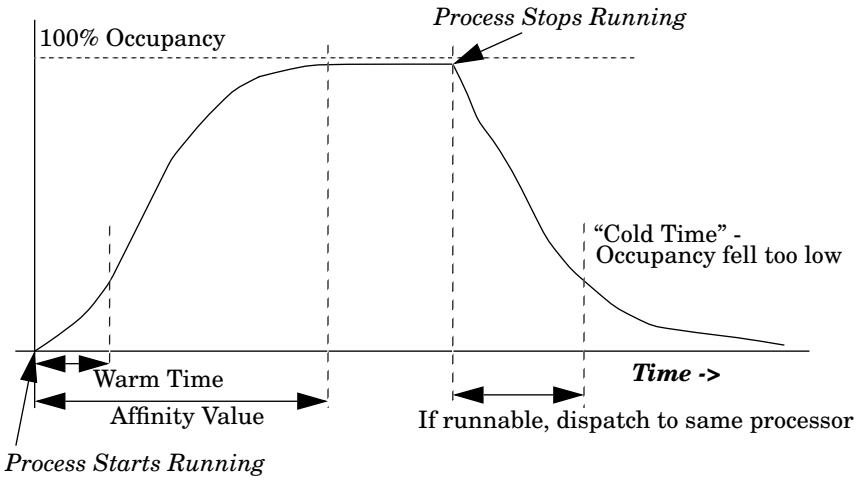

FIGURE 4: Occupancy Versus Time Assumption

gives a pictorial view of this assumption.

Affinity is captured when a process yields the processor; if the process has not run at least as long as the warm time, affinity handling is disabled for this dispatch cycle. The affinity of the process for the processor is capped at the affinity value and will range between the warm time and the affinity value based on the time spent last running.

To estimate the effect of running other processes on cache occupancy, each processor maintains a counter of the time spent running processes. When a process yields a processor, a snapshot of this counter is saved. The next dispatch decision to favor this process causes this snapshot to be compared with the current value of the counter for that processor. If enough

entered. The mechanism is very lightweight, assuming that a counter roll-over means "too long to worry about"; this is sensible as the scheduler typically deals in terms of 100's or 1000's of microseconds spent in various states. This finely-tracked time is the basis for proper operation of the scheduler and will be referred to as *run time, queue time*, and *wait time*.

Timeouts are sometimes used to control the time spent in a state. The typical usage is for managing time-slicing in sophisticated squeues such as the deadline or gang squeue. Internally, the scheduler manages timeouts in terms of the basic state timer and translates this to the units needed by the standard IRIX timeout mechanism. One interesting feature of the timeout algorithm is that it is processor-independent; that is, a time-slice-end timeout may fire on any processor but will cause an interrupt to be sent to the processor running the target process. This was necessitated by the IRIX timeout design which uses a master timing processor for fast timeouts.

The code is written to be independent of the clock frequency of the basic state timer; however, the use of the fastest timer available implies that the scheduler timing accuracy scales with the speed of the machine. For instance, a 100MHz R4000 class machine gives the scheduler a fundamental accuracy of 20 nanoseconds. Clearly, such accuracy is swamped by instruction overhead but realistic accuracy of a few microseconds is achieved. This accuracy is important to processor cache affinity calculations (see below), but is not available in general due to IRIX timeout granularity and system call interface limitations. As these limitations are removed the full accuracy of the scheduler can be exposed.

3.4 Processor Cache Affinity

Processor cache affinity is a mechanism which attempts to take advantage of data which a process may have fetched into a local processor cache last time it was running. This is important with modern caches; the largest caches currently supported by Silicon Graphics are four megabytes in size; sixteen megabyte caches will arrive soon. If processes are randomly assigned to processors on each dispatch cycle, there is a high probability that a process would spend much of its time-slice refetching data into the cache, rather than performing real work [VaZa91]. Affinity should not always be applied, however. For example, real-time or deadline processes are more interested in reducing latency and thus should always be dispatched immediately to any available processor.

The affinity mechanism implemented by the scheduler is based on the following model. It assumes that time spent in the *running* state accumulates good data in the cache, up to some limit defined by a combination between cache size and cache miss time. This limit is called the *affinity value*. There is also a minimum time, the *warm time*, which must be spent running to cause the affinity mechanism to activate. This minimum is necessary because affinity management creates additional overhead and at

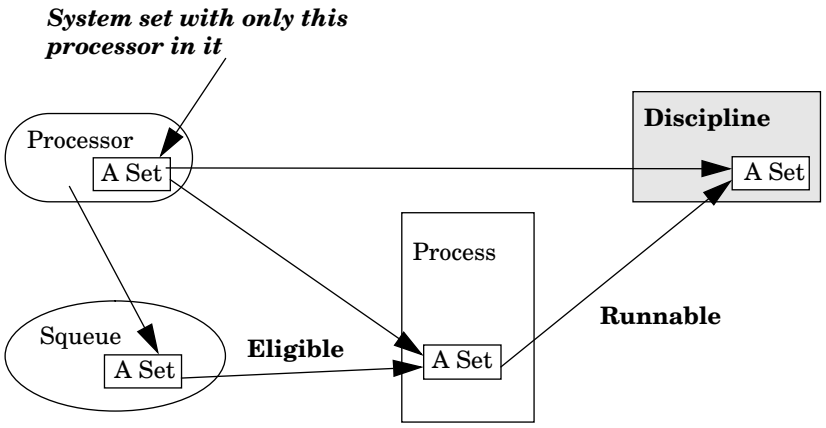

FIGURE 3: Applying Processor Sets

basic state timing and timeout management. In most traditional UNIX implementations very little tracking is done of the time spent in each of the basic process states. In general, a 100Hz counter is used to statistically track time spent running (split into user and kernel portions), with no tracking done of other states. Time slice management is performed using the same clock. In order to provide more precise timing for process control, visual simulation and other similar applications, the concept of high-accuracy timing has been introduced in previous versions of IRIX.

These timing mechanisms are based on keeping tight track of time based on the fastest available clock in a given system. For older IRIX systems without high resolution clocks, the basic kernel clock frequency is increased to 1000Hz or more, giving about one millisecond resolution. In Version 5 of IRIX, a new clock abstraction was introduced, allowing more accurate timing. On uniprocessor systems based on the R4000 architecture, the cycle counter provided by the microprocessor is used while on the Challenge multiprocessor system, the shared-memory bus clock is used. Both of these clocks provide accuracy on the order of 20 nanoseconds.

Higher level kernel mechanisms use this clock to perform accurate tracking of the total time a process spends in various states. Because of the long life of many processes, these mechanisms are somewhat heavyweight since they must carefully account for counter rollover and must keep the data in a form that performance monitoring tools can use. On the other hand, the IRIX scheduler needs to track a process at a much finer granularity - the amount of time a process *last* spent running or waiting is much more important. To this end, the scheduler tracks process state transitions directly. The time spent in the previous state is saved when a new state is

user area, current processor not in the processor set for a process, inability to access the graphics head, locked to a different processor, etc. When an squeue wishes to select a process, it must first check if it is runnable. A *discipline* is a conceptual grouping of such checks, for instance the *gfx* discipline performs checks necessary to determine if the process requires, and can access, the graphics head.

The following sections describe each of these utility functions in detail.

3.2 Processor Sets

A *processor set* is a named bit vector. The name is a 4-byte signed integer for storage and code efficiency. A user-level utility (*pset(1m)*) maintains a mapping of symbolic names to integer names known by the scheduler.

Like compiled languages, there is a distinct difference between *manipulating* a processor set and *binding* it to an object. A processor set is manipulated using a new command to the *sysmp(2)* system call, supporting actions such as creation, deletion, modification or query. A processor set is bound (or unbound) to an object using a new command to the *schedctl(2)* system call, supporting binding to processes, scheduling queues and disciplines.

Each processor set has a reference count, indicating the number of active bindings. A set cannot be deleted if the reference count is greater than zero. Certain processor sets may be marked as *system* sets, meaning that they are internally used by the scheduler and may not be modified or deleted. For example, each processor is assigned a processor set with only itself in it. This set can thus be used for fast set operations. Other system sets include all processors and all currently active processors.

Internally, the scheduler manipulates pointers to a descriptive structure for a processor set, rather than the sets themselves. This improves efficiency and modularity, while eliminating any size restrictions on the sets. A small set of processor set functions are provided, e.g., AND, OR, INTERSECTION, SET, CLEAR. Figure 3 shows the relationships among bound objects.

As described, there are three points at which processor sets may be bound: first, a squeue will not be examined for processes if the scanning processor is not in the squeue set; second, a process will not be examined unless the scanning processor is in its set; and finally, a process is not runnable if it requires a discipline and the scanning processor is not in the discipline set.

3.3 Timing

The timing services provided by the framework cover two areas:

```
do for each squeue {
   if (processor is not in squeue's processor set)
      try next squeue;
   if (squeue provides a process to run) then
      quit loop;
}
initialize scheduler state for running;
if (squeue did not setup time-slice-end)
   setup time-slice-end to defaults;
switch timing state for new process to "running";
return selected process;
End
```

The top level of the scheduler is conceptually quite simple. The actual queueing policy has been delegated to each squeue to determine. This top level algorithm provides some hooks that the various squeues may rely on. For instance, a squeue may specify that a particular routine (a "callout") is to be invoked when the process stops running. This can be used to maintain state information within a particular squeue.

Global routines also record all timing information and perform timing state changes. This information is then available to an squeue if needed. Also, an squeue may wish to control time-slice handling itself; if it does not, then the global routines will provide default time-slicing services.

Finally there are a number of utility functions that an squeue may call on to maintain state or make policy decisions. These are:

Processor Sets: The global framework provides a sophisticated mechanism for managing processor set information. An interface is provided for user-level programs to create, delete, and modify processor sets and their association with processes. No squeue currently deals explicitly with processor set information; instead, processor set checking is handled as part of the *runnable* utility.

Timing: The framework provides fundamental state timing services, as well as timeout management. These services are provided at the level of the fundamental hardware clock available, which is the R4000 counter register (external clock frequency on a uniprocessor) or the CC chip system clock (backplane clock frequency on a multiprocessor).

Processor Cache Affinity: The global framework provides a small set of routines for managing cache affinity.

Runnable: A process, even though queued, may not be runnable at any particular moment in time. Restrictions include a swapped

cessor running the lowest priority process is tracked. This processor may be immediately preempted if a real-time process is queued.

When a newly-created process is presented to the scheduler for the first time, the scheduler allocates a scheduling control block for the process. The control block holds all information needed to make scheduling decisions for a process, including state timing information. When a process exits, the scheduler is called to destroy this control block and free the allocated memory. The state timing information is sufficient to record the previous state, the current state, the time spent in the last state, a timestamp for the beginning of the current state, and a snapshot of the current UNIX seconds clock for rollover detection.

3.1 Control Framework

The main function of the scheduler is best described by the following pseudo-code for its two central functions: *put* a process on the queue and *get* the best process for the calling processor to run next.

```
Put a process on the scheduler queue
Begin
record time spent in previous state;
switch timing state for this process to “queued”;
if (callout on yield) then
   perform callout;
if (last state was running)
   cancel any pending timeouts;
record exception conditions;
do for each squeue {
   if (process pri within squeue priority limits)
then
      if (squeue accepts process) then
         quit loop;
}
assert(process was accepted by an squeue);
if (some processor is idle) then
   wake it up;
End

Get a process to run from the scheduler queue
Begin
if (calling process has yielded the processor) {
   set timing state of current process to “waiting”;
   if (callout on yield) then
      perform callout;
}
```

to obtain automatic load-balancing with a fully distributed algorithm.

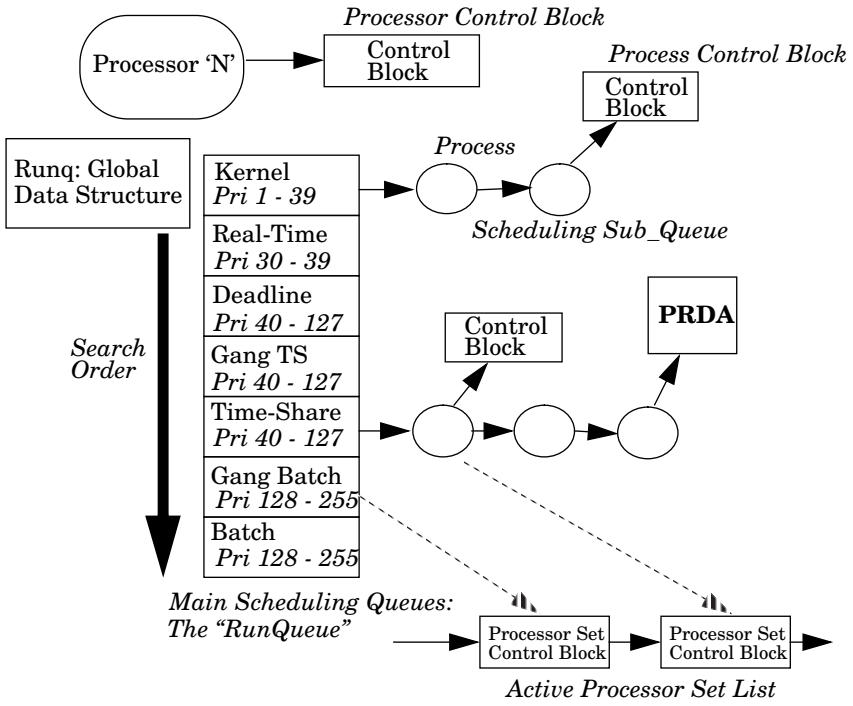

FIGURE 2: Overall Scheduler Architecture

The central data structure contains the main scheduling queues and is called the *runqueue*. The runqueue is arranged as a linked list of sub-queues, called *squeues*. Each squeue object is responsible for managing its own list of runnable processes, including addition, deletion, and selection of a candidate process to run and implementing a queueing policy most appropriate for the type of scheduling requested. The ordering of the squeues on the runqueue list determines the overall scheduling policy of the system. The time-share and batch squeues work as "catch-all" squeues for processes within their priority range.

The scheduler is anchored by a data structure called Runq, which contains all global information used by the scheduler. Separate pointers are kept to certain key squeues to aid other squeues in making local policy decisions. For instance, the gang scheduling squeue will not start a gang if there is a higher priority time-share process queued (see the section on gang scheduling). A list of pointers to the PDA of each processor is also maintained and allows the distributed scheduling algorithms to directly access key data for another processor, such as its current *running* time accumulation. To provide rapid scheduling of real-time processes, the pro-

cesses that are restricted to run on it. Restriction is used both by the kernel and by processes themselves. For instance, a single-threaded device driver may be allowed to run on only a single processor, while a real-time application may wish to dedicate processors for certain uses.

A *clock tick* is the basic heartbeat of a UNIX system. Its frequency depends on the underlying hardware; for all SGI systems, it is 100Hz. Each processor in the system has its own 100Hz clock, which is not necessarily synchronized with other processor clocks. When a clock tick occurs, the processor performs certain low-level maintenance actions, including executing any local *timeouts* that may have been queued. A timeout is a request to run a particular routine at some point in the future. In IRIX, one processor is termed the *master clock processor* and performs system-wide operations. This processor is responsible for maintaining the *fast clock*, which typically runs at 1000Hz, and the resulting *fast timeouts*, which have ten times the resolution of local timeouts. The resolution of the fast clock is configurable subject to the limitations of the underlying hardware.

The process *priority* is a small integer which indicates the importance of a process. Priority ranges from 1 to 255, with lower numbers indicating higher priority. IRIX attaches meaning to certain priority bands, indicating special dispatching features:

1 - 39	kernel-mode processes
30 - 39	user-mode real-time
40 - 127	general time-shared work
128 - 255	background processing

Note that in order to avoid the *priority inversion* problem IRIX implements the *basic priority inheritance protocol* as defined in [Sha90]. IRIX adds some additional non-priority mechanisms such as *gang scheduling* for distinguishing processes for special scheduling treatment.

Finally, the scheduler is mostly transparent to user-level processes. There are two distinct ways in which the process may request different scheduling behavior. The first of these is through the system call interface. The UNIX *nice(2)* call works as expected, while more sophisticated requests may be made through the *schedctl(2)* interface. The second method for communicating with the scheduler consists of a memory segment that is shared with the scheduler, called the *Private Data Area* (PRDA). Certain requests can be made simply by changing a variable in the PRDA.

3. Scheduler Architecture

A pictorial overview of the scheduler is given in Figure 2. Conceptually, this architecture provides the equivalent of a single, ordered list of processes waiting to run, with the first process on that list being the highest priority process in the system. This strategy makes it relatively simple

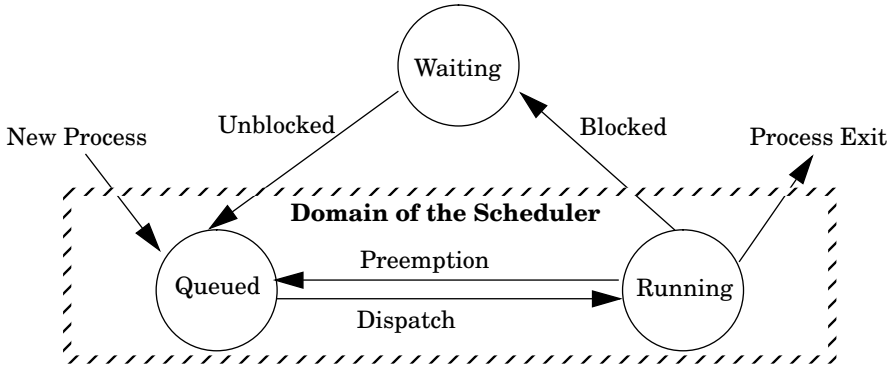

FIGURE 1: Process State Diagram

ple in a computer which includes processors with different instruction sets and tasks. Each processor has a data structure associated with it called the *private data area* (PDA) which records data that are local and unique to a processor, and includes data that are used to make scheduling decisions.

The lifetime of every process in the system can be described with the simple state diagram shown in Figure 1. The *queued* state indicates that the process is waiting to run on a processor. Processes in this state are kept in the *runqueue*. When a processor has finished with a previous process and is looking for more work to do, it queries the runqueue and picks the "best" process to run next. This process is then said to have been *dispatched* to a processor and it enters the *running* state. The *waiting* state indicates that the process has blocked waiting for some event to occur. A process will stay in this state until explicitly queued by some other process. The *running* state indicates that the process is currently in control of a processor. The process may either block and enter the waiting state, or can preempt itself (at time slice end, for instance) and enter the queued state. In either case, the process is said to *yield* the processor. A *dispatch cycle* occurs when one of the processors in the system initiates a scan of the runqueue to find a new process to run. During a dispatch cycle, each process on the runqueue will be checked to determine if it is *runnable* and the first *runnable* process found is dispatched to the processor. The *runnable* attribute is an instantaneous condition which indicates if the process can enter the running state. For example, access to the graphics head is a runnable condition which may change moment to moment and thus must be checked every time an attempt is made to dispatch a process.

A *real-time* process gets special attention from the scheduler as it must get the best response possible from the entire system, that is, the scheduling latency of such a process must meet certain worst-case criteria.

A *process* is said to be *restricted* if it can only run on a specific processor; alternately, a *processor* is said to be *restricted* if it can only run pro-

must operate have become much more complex. The operating system must support various forms of real-time (guaranteed response) applications, multiprocessors, massively parallel supercomputers, multi-media desktops, mainframe OLTP applications *and* departmental minicomputers. Aside from the increasing breadth of computing applications to support, the scheduler must also consider the major features of modern computing architectures: extensive cache hierarchies combined with large numbers of processors in a shared-memory environment.

There are two fundamental approaches to dealing with this complexity. One is to create new application-specific versions of the operating system for each environment. For example, there are several commercial "real-time" versions of UNIX for process control. Vendors of database systems typically optimize the scheduler for the bursty transaction-oriented nature of OLTP work. Vendors of UNIX-based supercomputers will make modifications to improve scheduling behavior for large numbers of parallel processors. Unfortunately, optimization is often achieved at the cost of poor performance in other domains.

The other approach is to design a single scheduling system which produces near-optimal results in a large number of different environments, potentially all active at the same time. Such a general-purpose system is more difficult to construct and can take additional memory and processor cycles to implement. The payoff for careful design is the development of a single body of efficient source code, a highly scalable operating system, and a very high degree of compatibility between machines in very different application areas.

The IRIX scheduler is an example of the latter approach. The remainder of this paper discusses its features, architecture and implementation.

2. Principles of IRIX Process and Processor Scheduling

An IRIX *process* is a collection of resources within the computer system. In general, a process is responsible for managing itself and its resources. This is accomplished by splitting each process into two pieces, the *kernel* portion and the *user* portion. While in the user portion, the process may only execute unprivileged code; to access a system resource the process executes a trap and enters the kernel portion of the process. This trap mechanism ensures that the process always executes kernel code as a trusted entity. The kernel portion of a process may access all parts of kernel memory.

A *processor* is the basic scheduling entity on the system and there may be any number of them present. The scheduler is fully distributed: that is, each processor is responsible for its own scheduling decisions based on information kept in a global information structure called the *runqueue*. All processors are assumed to be identical. The scheduler is designed, however, to allow future implementations where this is not the case, for exam-

A Scalable Multi-Discipline, Multiple-Processor Scheduling Framework for IRIX

James M. Barton
Nawaf Bitar

Silicon Graphics Computer Systems
Mountain View, California

ABSTRACT

This document describes the processor scheduling framework implemented in the Silicon Graphics IRIX Version 5 operating system. This framework provides the standard features and behavior expected of any UNIX time-sharing system, while adding support for four additional disciplines: a fast-response scheme for low-latency real-time computing, a time-based regime for throughput-oriented real-time computing, a multi-thread scheme for parallel computing applications, and several flavors of batch processing. In addition, the scheduling framework adds the notions of processor cache affinity, which attempts to take advantage of data already fetched into a particular processor cache, and processor sets, which allow an additional level of scheduling control on a per-processor basis. These features have been successfully deployed in production environments on machines ranging from single-processor desktop workstations to high-performance supercomputing multiprocessors.

1. Introduction

As a UNIX-based operating system, IRIX has a unique heritage. UNIX was originally designed as a reaction to large-scale mainframe operating systems which were perceived to be difficult to use, slow, and overly complex. The processor scheduling theory used was that of *time-sharing:* preemptive multi-tasking using multi-level feedback with aging; essentially, a priority-based scheduling regime where priorities degrade with increasing CPU utilization.

Such a scheme was sensible for the envisioned use of UNIX: departmental level minicomputers serving many users through low-speed terminals and peripherals. The number of runnable processes was never expected to be large, so a singly-linked list of runnable processes was used. New processes were inserted at the end of the list and the list was completely searched for the best candidate on every scheduling cycle. This method has actually worked quite well for twenty years and is still the basis of many modern UNIX schedulers [Bach86] [Leff89].

However, the environments in which a UNIX-based operating system