PMQS : Scalable Linux Scheduling for High End Servers

Hubertus Franke[†], Shailabh Nagar[†], Mike Kravetz[‡], Rajan Ravindran[†]

[†]*IBM Thomas J. Watson Research Center*
[‡]*IBM Linux Technology Center*

{frankeh,nagar,mkravetz,rajancr}@us.ibm.com
http://lse.sourceforge.net/scheduling

Abstract

The Linux kernel scheduler for large SMP and NUMA systems needs to address conflicting requirements of system througput and application response times. This paper presents a pooled multiqueue scheduler (PMQS) designed for such high-end SMP and NUMA systems. PMQS is based on MQS, a multiqueue scheduler that has previously been shown to significantly improve upon the scalability of the current Linux 2.4 scheduler. Load imbalances introduced by the pooling approach are reduced by a load balancing module. The paper presents different kinds of load balancers and compares their efficacy. The performance evaluation of PMQS and the load balancer is carried out using application and microbenchmarks with mixed results. Pooling is shown to have potential for further improving MQS' performance though more work needs to be done on the load balancers.

1 Introduction

Linux is becoming increasingly popular as a server operating system. It has already proven itself as a cost-effective solution for Web, file and print serving which typically run on systems with 1-4 CPUs. More demanding enterprise applications, such as database, e-business or departmental servers, tend to be deployed on larger symmetric multiprocessor (SMP) systems. To support such applications, Linux has to scale well as more CPUs are added to an SMP. Very large SMPs are increasingly built upon smaller SMP building blocks interconnected by a cache coherent interconnect. As this leads to non-uniform memory accesses, these systems are referred to as NUMA systems. Linux systems need to incorporate NUMA awareness into the base kernel.

With the increasing CPU count and application load, Linux SMP scalability has become one of the focal points for kernel development. Within that context, we have been looking at the scalability of the Linux kernel scheduler. The current Linux scheduler (2.4.x kernel) has two defining characteristics. First, there is a single unordered runqueue for all runnable tasks in the system, protected by a single spinlock. Second, during scheduling, every task on the runqueue is examined while the runqueue lock is held. These have a two-fold effect on scalability. As the number of CPUs increases, there is more potential for lock contention. As the number of runnable tasks increases, lock hold time increases due to the linear examination of the runqueue. Independent of the number of CPUs, increased lock hold time can also cause increased lock contention, depending on the frequency of scheduling decisions. For spinlocks, increased lock hold time and lock contention result in a direct increase in lock wait time which is a waste of CPU cycles. These observations are reinforced by recent studies. Measurements using Java benchmarks [2] show that the scheduler can consume up to 25% of the total system time for workloads with a large number of tasks. Another study [3] has observed run queue lock contention to be as high as 75% on a 32-way SMP.

To address these deficiencies, our previous work in [5] proposes and implements two kinds of kernel schedulers called Multi Queue Scheduler (MQS) and Priority Level Scheduler (PLS). PLS organizes the global runqueue using multiple priority levels resulting in fewer tasks examinations during scheduling. MQS is designed to address the issues of lock contention and lock hold time simultaneously. It does this by splitting up the global runqueue and its associated lock into per-CPU equivalents. The two schedulers have been evaluated on an 8-way SMP using a variety of relevant workloads. Our results show that lock contention is the greater of the two problems and that MQS outperforms both the default SMP scheduler (DSS) and PLS in most cases. Both PLS and MQS attempt to be functionally equivalent to DSS. As a result, MQS ends up doing an $O(N_{cpus})$ scan looking for the globally best candidate. While this has minimal impact on lock contention, it does leave MQS open to scalability concerns. A side-effect of making a global decision is an expected increase in the number of task migrations across CPUs leading to an increase in cache misses, and on NUMA systems, increased accesses to remote memory.

In this paper, we make an initial attempt to address these problems by using processor pooling. Processor pooling is implemented by an extension of MQS, called Pooled Multi Queue Scheduler (PMQS), which partitions the CPUs of an SMP into subsets for making scheduling decisions. This places an upper bound on the scope of the search for candidate CPUs and tasks and is expected to improve scheduler performance. It is also expected to improve system throughput by decreasing the probability of cache misses and remote memory accesses. However, processor pooling can create load imbalance problems due to runqueue partitioning. We look at two ways of balancing CPU loads. The initial placement (IP) scheme places newly created tasks on the least loaded CPU and does not interfere with the functioning of PMQS thereafter. The other way of load balancing is more aggressive. It runs periodically and explicitly balances CPU runqueues by moving runnable tasks between them.

Processor pooling is not a new idea. [7] has done a simulation-based study of processor pooling for parallel systems. Their results indicate that such pooling reduces the average job response time. The importance of cache-affinity in making scheduling decisions has been shown in [8, 9] using simulations and analytical models. The effect of initial placement as a load balancing mechanism has been simulated in [4]. In this paper, we take an implementation based approach to processor pooling in the context of Linux.

We examine the performance of processor pooling using two representative benchmarks, Mkbench and Chat, on a 16-way NUMA system and on an 8-way SMP. For Mkbench, on the NUMA system we find that PMQS shows substantial benefits over DSS and MQS. These benefits are even higher when no load balancing is done. However, for Chat, while PMQS does outperform DSS, it does substantially worse than MQS. Even though we feel that Mkbench is more representative of server workloads, the mixed performance makes it difficult to generalize the results of this paper and draw strong conclusions about the efficacy of pooling under Linux. These are only preliminary implementations and results of the pooling concept. Much work remains to be done to make stronger pronouncements.

The rest of the paper is organized as follows. In Sections 2 and 3 we describe the default SMP scheduler (DSS) and the Multi Queue Scheduler (MQS) respectively. The Pooled Multi Queue Scheduler (PMQS) is presented in Section 4. The load balancing is described in Section 6. Section 7 contains the performance results and Section 8 concludes with directions for future work.

2 Default SMP Scheduler (DSS)

The default SMP scheduler (DSS) in Linux 2.4.x treats processes and threads the same way, referring to them as tasks. Each task has a corresponding data structure which maintains state related to address space, memory management, signal management, open files and privileges. Traditional threading models and light-weight processes are supported through the `clone` system call.

For the purpose of scheduling, time is measured in architecture-dependent units called ticks. On x86 systems, timer ticks are generated at a 10ms resolution. Each task maintains a counter (`tsk->counter`) which expresses the time quantum for which it can execute before it can be preempted. By decrementing this counter on timer tick interrupts, DSS implements a priority-decay mechanism for non-realtime tasks. The priority of a task is determined by a `goodness()` value that depends on its remaining time quantum, `nice` value and the affinity towards the last CPU on which it ran. DSS supports preemption of tasks only when they run in user mode. Priority preemption can occur any time the scheduler runs.

The kernel scheduler consists of two primary functions :

1. `schedule(void) :` This function is called synchronously by a processor to select the next task to run e.g. at the end of `sleep()`, `wait_for_IO()` or `schedule_timeout()`. It is also called preemptively on the return path from an interrupt e.g. a reschedule-IPI (interprocessor interrupt) from another processor, I/O completion or system call.
2. `reschedule_idle(task_struct *tsk) :` This function is called in `wake_up_process()` to find a suitable processor on which the parameter task can be dispatched. `wake_up_process()` is called when a task is first created or when it has to be re-entered into the runqueue after an I/O or sleep operation. `reschedule_idle()` tries to find either an idle processor or one which is running a task with a lower goodness value. If successful, it sends an IPI to the target CPU, forcing it to invoke `schedule()` and preempt its currently running task.

Internally, the scheduler maintains a single runqueue protected by a spinlock. The queue is unordered, which allows tasks to be inserted and deleted efficiently. However, in order to select a new task to run, the scheduler has to lock and traverse the entire runqueue, comparing the goodness value of each schedulable task. A task is considered schedulable if it is not already running and it is enabled for dispatch on the target CPU. The goodness value, determined by the `goodness()` function, distinguishes between three types of tasks : realtime tasks (values 1000+), regular tasks (values between 0 and 1000) and tasks which have yielded the processor (value -1). For regular tasks, the goodness value consists of a static or non-affinity part and a dynamic or affinity part. The non-affinity goodness depends on the task's `counter` and `nice` values. The affinity part accounts for the anticipated overheads of cache misses and page table switches incurred as a result of migrating tasks across CPUs. If the invoking CPU is the same as the one the task last ran on, the goodness value is boosted by an architecture dependent value called `PROC_CHANGE_PENALTY`. If the memory management object (`tsk->mm`) is the same, goodness values are boosted by 1. The counter values of all tasks are recalculated when all schedulable tasks on the runqueue have expired their time quanta. Due to space limitations, we refer the reader to detailed descriptions of DSS in [6, 1].

The implications of the scheduling algorithm for large SMP and NUMA machines is discussed in Section 4.

3 Multi Queue Scheduler (MQS)

The Multi Queue Scheduler (MQS) is designed to address scalability by reducing lock contention and lock hold times while maintaining functional equivalence with DSS. It breaks up the global run-queue and global run-queue lock into corresponding per-CPU structures. Lock hold times are reduced by limiting the examination of tasks to those on the runqueue of the invoking CPU along with an intelligent examination of data corresponding to the non-local runqueues. More-over, the absence of a global lock allows multiple instances of the scheduler to be run in parallel, reducing lock wait time related to lock contention. Together these reduce the scheduler related lock contention seen by the system.

MQS defines per-CPU runqueues which are similar to the global runqueue of the DSS scheduler. Related information such as the number of runnable tasks on this runqueue is maintained and protected by a per-CPU runqueue lock.

The `schedule()` routine of MQS operates in two distinct phases. In the first phase, it examines the local runqueue of the invoking CPU and finds the best local task to run. In the second phase, it compares this local candidate with top candidates from remote runqueues and chooses the global best.

In more detail, the `schedule()` routine of MQS acquires the runqueue lock of the invoking CPU's runqueue and scans the latter looking for the schedulable task with the highest goodness value. To facilitate the global decision in the second phase, it also records the second highest non-affinity goodness value in the `max_na_goodness` field of the local runqueue. The non-affinity goodness (henceforth called na_goodness) is the goodness value of a task without any consideration for CPU or memory map affinity. The local candidate's goodness value (which includes appropriate affinity boosts) is compared with the `max_na_goodness` of all other runqueues to determine the best global candidate. If the global candidate is on a remote runqueue, `schedule()` tries to acquire the corresponding lock and move that candidate task over to its local runqueue. If it fails to acquire the lock or the remote task is no longer a candidate (its na_goodness value has changed), `schedule()` skips the corresponding runqueue and tries again with the next best global candidate. In these situations, MQS's decisions deviate slightly from those made by DSS e.g. the third best task of the skipped runqueue could also have been a candidate but is not considered as one by MQS.

The `reschedule_idle()` function attempts to find a CPU for a task which becomes runnable. It creates a list of candidate CPUs and the na_goodness values of tasks currently running on those CPUs. It chooses a target CPU in much the same way as the `schedule()` routine, trying to acquire a runqueue lock and verifying that the `na_goodness` value is still valid. Once a target CPU is determined, it moves the task denoted by its argument onto the target CPU's runqueue and sends an IPI to the target CPU to force a `schedule()`. `reschedule_idle()` maintains functional equivalence with DSS in other ways too. If a tasks' previous CPU is idle, it is chosen as the target. Amongst other idle CPUs, the one which has been idle the longest is chosen first.

MQS's treatment of realtime tasks takes into account the conflicting requirements of efficient dispatch and the need to support Round Robin and FIFO scheduling policies. Like DSS, it keeps runnable realtime tasks on a separate global runqueue and processes them the same way.

The implementation of MQS avoids unnecessary cache misses and false sharing. Runqueue data is allocated in per-CPU cache-aligned data structures. Implications of MQS for large SMP and NUMA systems is discussed in Section 4

4 Pooled Multi Queue Scheduler (PMQS)

Before describing the design of the pooled multiqueue scheduler, it is instructive to examine the implications of the scheduling algorithm followed by DSS and MQS. During `schedule()`, DSS selects the next best task to run, as determined by the `goodness()` function. MQ attempts to do the same though its examination of non-local tasks (tasks not on the runqueue of the invoking cpu) is not done under a lock. During `reschedule_idle()`, DSS tries to find the best CPU on which the newly woken/created task can be run. MQS does the same without holding a lock on the remote runqueues. Overall, whether it is choosing a task or a CPU, DSS looks at all viable candidates and choose the globally best one while MQ tries to achieve the same goal by an intelligent examination of fewer candidates. This has two implications for large SMP and NUMA machines :

1. **Scalability** : The `schedule()` function in DSS is $O(N_{tasks})$ whereas on MQ it is $O(\frac{N_{tasks}}{N_{cpus}} + N_{cpus})$ (on an average, the local runqueue in MQ has $\frac{1}{N_{cpus}}$ of the total tasks in the system). Even if N_{tasks} is a small, constant multiple of N_{cpus}, both schedulers approach $O(N_{cpus})$ and are susceptible to scalability problems. Similar conclusions can be drawn from their `reschedule_idle()` functions, which are both $O(N_{cpus})$, albeit with different locking strategies. Compared to DSS, MQS does increase scalability considerably by removal of a global lock. But it does not eliminate the potential scalability bottleneck.

2. **Locality** : When a runnable thread migrates from one CPU to another (either directly as a result of preemption or after a sleep/wakeup cycle), it runs the risk of losing the cache context accumalated on the previous CPU. For any SMP system, the probability of being able to take advantage of a previous run on the same CPU, is proportional to the number of alternate CPUs to which it could be migrated. For a NUMA system, migration could have even a greater penalty if a task runs on a node different from the one on which most of its memory is allocated. The effect of increased probability of cache misses and remote memory accesses is also highly dependent on the workload e.g. an application whose working set causes its cache context to get replaced periodically is not as affected by the cache miss effect and applications with high cache hit rates will not be severely affected by remote memory accesses.

To address these two problems, we propose a pooled multiqueue scheduler (PMQS) based on MQS. The central idea in PMQS is to partition the CPUs of an SMP/NUMA into pools and to limit the search for candidate CPUs and/or tasks to these pools. The size of the pools is a configurable parameter that can be dynamically changed to meet different criteria of performance, fairness etc. Restricting the scope of scheduling decisions to fixed-size pools improves the scalability of the scheduler by making it independent of NCPUS. It also increases the localilty of tasks with respect to the CPUs on which they run.

For simplicity, the implementation of PMQS is based on minor modifications to MQS. In general, DSS code which looks at all CPUs using

```
for (i = 0; i <= smp_num_cpus; i++)
```

is replaced by

```
for (i = first; i <= last; i++)
```

where first and last denote the corresponding limits of the invoking CPU's pool. The first phase of the `schedule()` routine of MQS is unchanged in PMQS and is used to find the best local candidate task. In the second phase of the routine, the only remote runqueues examined are those of CPUs within the same pool as the invoking CPU. The examination of remote runqueues continues to be done without holding a lock. In the `reschedule_idle()` function, only those CPUs are treated as candidates which lie within the same pool as the task's previous CPU. The only exception to this rule is when there are idle CPUs outside the pool. Leaving a CPU idle when other CPUs have runnable tasks leads to a complete waste of CPU cycles and is particularly undesirable in a server environment. Therefore, PMQS looks for idle CPUs systemwide without regard to pool boundaries. The hunt for idle CPUs uses a simple mechanism. A global bitmap, called *node_idle_bits*, records the currently idle CPUs in the system. The bitmap is maintained by the scheduler when it switches tasks on any CPU. During *schedule()*, *node_idle_bits* is first masked with a local pool mask to identify idle CPUs within the local pool. If none is found, it is masked with a remote pool mask to identify idle CPUs in remote pools. Consistent with the pooling philosophy, idle CPUs within a pool are preferentially chosen over those outside the pool. If an idle CPU is found at any stage, it is selected as the target CPU. Contrary to the behaviour of DSS and MQS, PMQS makes no attempt to select the longest idle CPU as the target. If no idle CPU is found, `reschedule_idle()` proceeds as normal, examining *max_na_goodness* values of CPUs within the pool.

5 Benchmarks

Before we proceed with the description of the different load balancers, we describe the two benchmarks (*Mkbench* and *Chat*) that are used in this study. These two benchmarks will be referenced in the load balancing section.

Mkbench consists of the multiple parallel compilation of the Linux kernel. This has traditionally been a popular benchmark in the Linux community, as it is often seen in development environments. It offers a good mix of I/O and CPU-bound phases especially when a parallel build is used. Parallel builds are issued by using the -j option of the make command. Our measurements of runqueue lengths during such compiles show that the number of runnable threads in the system roughly corresponds to the job size specified by the -j option.

The **Chat** benchmark, which can be found at `http://lbs.sourceforge.net/`, simulates chatrooms with multiple users exchanging messages using TCP sockets. The benchmark is based on the Volano Java benchmark, which was used in some of the first reports of scalability limitations with the default SMP scheduler (DSS) of Linux [2]. Each chatroom consists of 20 users with each user broadcasting a variable number of 100 byte messages to every other user in the room. A user is represented by two pairs of threads (one each for send and receive) on the client and server side, resulting in 4 threads per user and 80 threads per room. Each message is sent from the client_send to its server_receive which then broadcasts it to all other client_receive threads in the room. 100 messages sent by each user translate to 20*100*19=38,000 messages being sent and received per room. Each receive is a blocking read and the interleaving of numerous reads and writes causes the scheduler code to be invoked frequently. The characteristic parameters of the Chat benchmark are the number of rooms and the number of messages per user. From a scheduler perspective, the former controls the number of threads created and the latter controls the number of times threads sleep and awaken via blocking reads. At the end of a benchmark run, the client side reports the throughput in number of messages per second. A higher throughput indicates a more efficient kernel scheduler.

6 Load Balancing

Dividing the CPUs of an SMP into pools for scheduling decisions, has the welcome effect of improving the scalability of the scheduling algorithms. However, this kind of limiting introduces the problem of load imbalance between the CPUs in different pools. We focus on two kinds of imbalances depending on what is defined as the load :

- **Runqueue length imbalance** : The number of tasks on the CPU runqueues varies with application being run by each of those tasks. Runqueues with more I/O bound tasks are likely to have fewer runnable tasks on an average. The sum of runqueue lengths for the CPUs in a pool could differ significantly from the sum for another pool leading to a runqueue length imbalance. The performance impact of such an imbalance is felt in two ways. First, there is an increased probability of CPUs being idle in one pool while another pool has runnable tasks. This has been taken care off in the *schedule()* functions through the global *idle_mask*. Second, the tasks in each pool have a longer average waiting time in the runqueue before they get to run.
- **Priority inversion** : The priority of the tasks running on CPUs of a pool is determined by the distribution of priority values amongst the pools. For strictly

pooled schedulers, a task gets to run on a CPU based on it relative priority to currently running tasks in the same pool. Hence it is possible that a runnable high priority task waits for a CPU in one pool while a lower priority task is running on a CPU in a different pool. This kind of priority inversion stems directly from the conflict between a global priority value and a local (limited to a pool) search for candidate tasks/CPUs. Priority inversion is primarily a fairness issue though it could also have performance implications for multithreaded or interdependent processes.

The load imbalances mentioned here only pertain to non-real time (SCHED_OTHER) tasks as MQS and PMQS maintain real time tasks on a separate global runqueue.

The existence of load imbalances does not necessarily call for corrective measures. For high end systems where system throughput is generally more important than job response times, isolating CPU pools from each other might be desirable. In such cases, priority inversion is not an issue and it is sufficient to ensure that all CPUs have enough tasks and that the initial placement of tasks (amongst pools) is balanced.

In this paper we examine various load balancing mechanisms under PMQS and compare their efficacy and their overall performance impact on various workloads as compared to DSS and MQS. These mechanisms are LBOFF, IP, LBC and LBP and are described below. To establish their efficacy we use the two distinct workloads, namely Mkbench and Chat, to monitor runqueue length per CPU for a 4-way SMP system. PMQS is configured with a poolsize of 1. We show for these benchmarks and configurations the deviation from the mean runqueue length at one second intervals. Mkbench was run with 2 simultaneous kernel builds with "-j 16" yielding an average load of 32 or 8 per CPU. Chat was run with 10 rooms and 900 messages, which yielded an average of 207 runnable tasks or 52 per run queue.

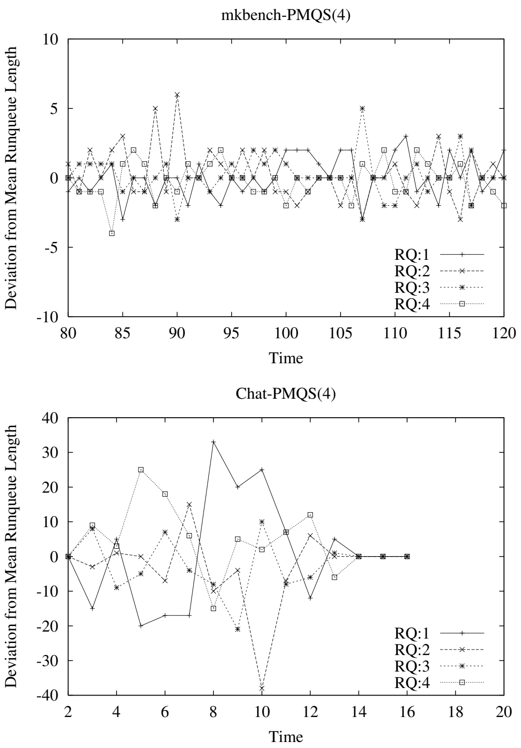

Figure 1: Deviation from mean of runqueue lengths for a 4-way SMP executing PMQS(4)

Figure 1 shows the load imbalances that are achieved under Mkbench and Chat when running PMQS with poolsize=4, which is basically equivalent to MQS. For Mkbench the individual runqueue length of the various CPUs falls into a narrow range of ± 2.

LBOFF simply utilizes CPU pooling without any attempt to balance load among the pools or CPUs. With poolsize=1, the runqueues are isolated from each other. The only means for a process to migrate from one CPU to another is during reschedule_idle() invocations when there exist idle processors on remote pools. Figure 2 shows the load imbalances that are achieved under Mkbench and Chat. For both, the per-CPU load can deviate quite substantially. We use this graph as a reference point for evaluating the load balancers below.

In *initial placement* (IP), a task is moved to the least loaded CPU as defined by the CPU's runqueue length, when a new program is launched, i.e. at *sys_execv()* time. Figure 3

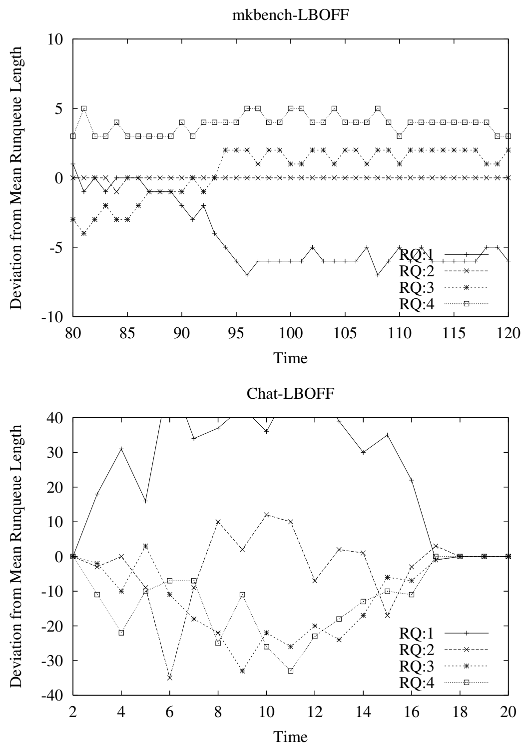

Figure 2: Deviation from mean of runqueue lengths for a 4-way SMP executing LBOFF

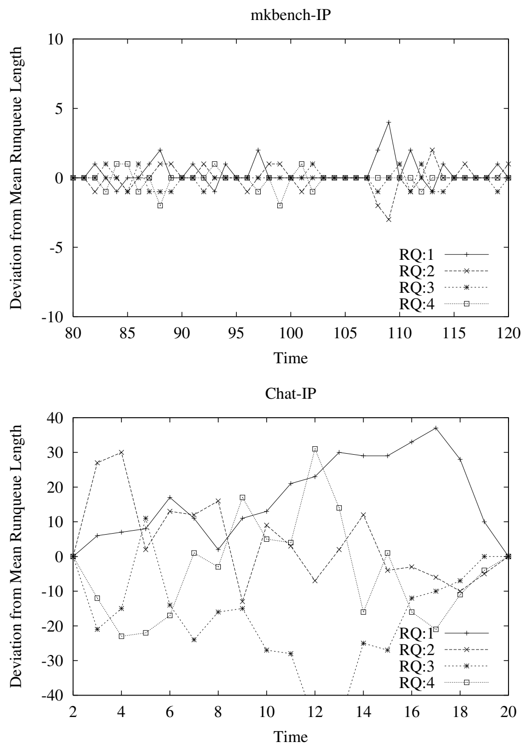

Figure 3: Deviation from mean of runqueue lengths for a 4-way SMP executing IP

shows that IP is very effective in equalizing the runqueue lengths in an environment of short lived processes. On the other hand, in environments with long lived processes IP is ineffective.

We compare this static approach to the problem of load imbalance with a flexible and dynamic load balancing mechanism. This flexible mechanism allows a system administrator to choose between the extremes of isolating CPUs (LBOFF) and treating them as one entity for scheduling (as is done by MQS). To do this, we provide an external load balancer module (LB) which balances the runqueues of CPUs belonging to different pools based on user-specified parameters and and a load function determining the CPU-load weight of each runqueue towards the overall load. So far we have considered two load functions. The `runqueue_length` load function is simply the length of the CPU runqueue and the `runqueue_na_goodness` load function is computed by summing the non-affinity goodness values of each task on a runqueue. The results presented in this paper are based on the `runqueue_length` function.

The LB module is invoked periodically through a timer function. The frequency of invocation is controlled through a user-specified parameter which can be dynamically altered through a /proc interface. Unless otherwise noted, the LB is invoked every 600 milliseconds. On each invocation, LB first records the load on each CPU runqueue. Once individual runqueue loads have been determined, LB computes the average load across the system. Runqueues are marked as having a “surplus” or “deficit” load.

From here we have experimented with two different version of load balancers, called LBC and LBP. LBC (**L**oad **B**alancing across all **Cpus**), tries to equalize all runqueues within the system tightly. For that LBC first performs intra-pool balancing by transferring tasks from surplus to deficit runqueues within each pool until runqueue loads are equal the average. In the second stage, tasks are transferred between the remaining surplus to deficit runqueues system wide, reflecting an

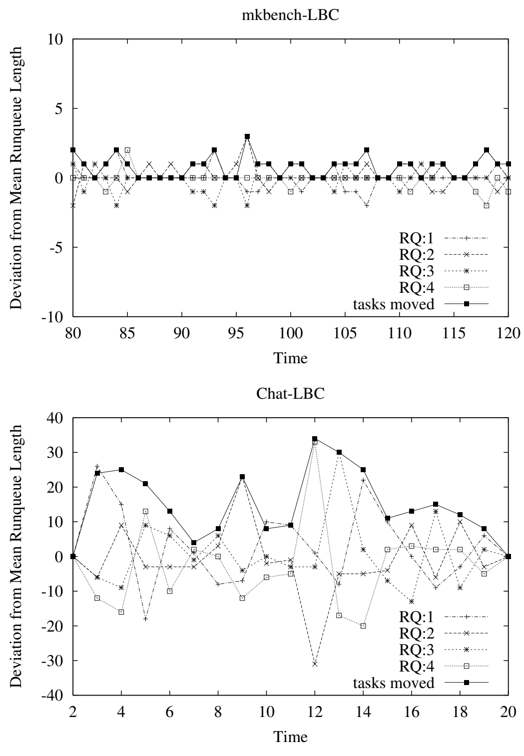

Figure 4: Deviation from mean of runqueue lengths for a 4-way SMP executing LBC

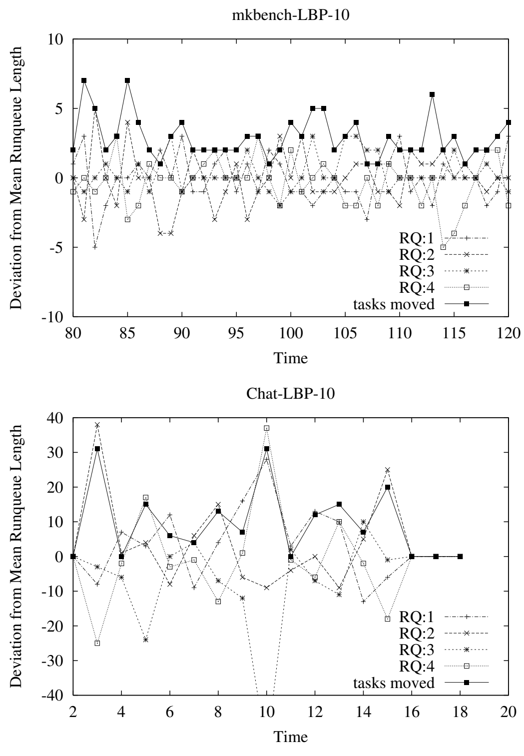

Figure 5: Deviation from mean of runqueue lengths for a 4-way SMP executing LBP-10

inter-pool balancing. In the following figures we also plot the total number of tasks moved during the LB phase to indicate the corrective actions taken by the LB after the observed state. Figure 4 shows that for Mkbench, LBC controls the runqueue length very tightly and needs to typically move only one or two tasks. For Chat, however, Figure 4 shows that LBC's tight balancing act leads to over correction as is clearly seen in states t=5,13,14. The statistics are summarized in Table 1. For LBC an average of 11.5% of tasks are moved every LB invocation with a maximum of 23.1%.

We, therefore, developed LBP (**L**oad **B**alancing across all **P**ools), which differs from LBC in two aspects. First, it does not perform any intra-pool balancing on the assumption that `schedule()` and `reschedule_idle()` do a good enough job of load balancing within a pool as shown in Figure 1. Second, it defines a user-specified error tolerance factor to avoid over aggressive corrections leading to the oscillation seen in Figure 4. A runqueue is considered balanced, if its load is within the error tolerance of the system average load. Tasks from surplus queues are transferred to deficit queues only if they are in distinct pools. The error tolerance factor is dynamically configurable through the /proc interface. Since poolsize=1 was selected for this evaluation, the following figures simply demonstrate the effects of the error tolerance.

Figure 5 shows the profile for LBP-10, i.e. LBP with a error tolerance of 10%. Combined with the statistics in Table 1, we observe that average number of tasks moved has been reduced to 5.6%. The combined average runqueue length is 179, which amounts to 45 per CPU. Only if the individual runqueue length differs more than 5 from the mean does LBP-10 try to balance that queue. The oscillations observed under LBC are significantly reduced.

Figure 6 shows the profile for LBP-45, i.e. LBP with a error tolerance of 45%. Combined with the statistics in Table 1, we observe that average number of tasks moved has

	LBC		LBP-10		LBP-45	
Σ Runqueue Lengths: mean (max)	150	(244)	179	(308)	178	(347)
Moved Tasks: mean (max)	17	(34)	10	(31)	2	(15)
% Moved Tasks: mean (max)	11.5%	(23.1%)	5.6%	(24.4%)	0.9%	(5.5%)

Table 1: Σ Runqueue Lengths and Tasks Moved Statistics for a 4-way SMP.

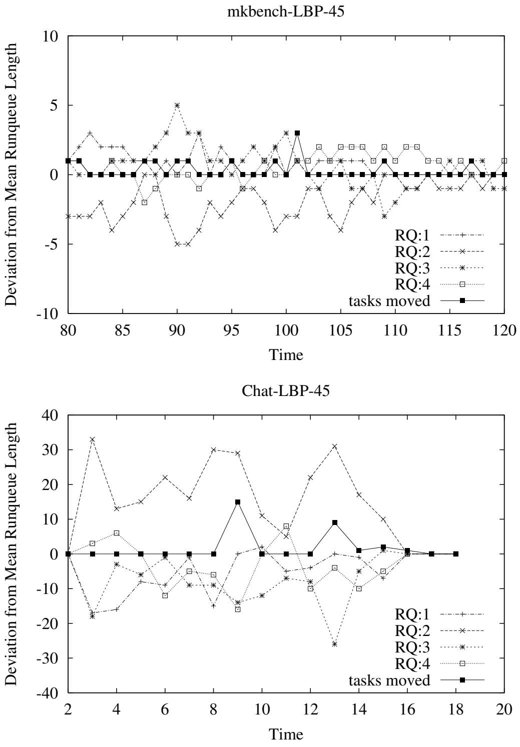

Figure 6: Deviation from mean of runqueue lengths for a 4-way SMP executing LBP-45

been reduced to 0.9%. The combined average runqueue length is 178, which amounts to 45 per CPU. Only if the individual runqueue length differs more than 20 from the mean does LBP-45 try to balance that queue. The oscillations observed under LBC and LBP-10 are virtually eliminated.

For Mkbench, Figure 5 actually shows worse oscillating behavior for LBP-10 as compared to LBC, requiring more aggressive task movements. In contrast, LBP-45 initiates a significantly smaller number of task moves, resulting in a smoother profile. However neither LBP is capable too obtain the close balanced achieved in IP and LBC.

To summarize, in this section we introduced various load balancing mechanisms. We observed that these mechanisms in general are effective in balancing the lengths of the individual runqueues. However, we also observed that the runqueue length are sensitive to the nature of the workload and that tight load balancing methods can have adverse effects. In the next section, we will evaluate the overall performance effects of load balancing techniques.

7 Performance Evaluation

To evaluate the efficacy of PMQS and the various load balancing techniques we ran Mkbench and Chat on two different systems, (a) a 16-way NUMA system and (b) an 8-way SMP system. The 16-way NUMA system consists 4 quad building blocks. Each quad consisted of 4x450 MHz PIII processors with 512 KB caches and 1GB main memory. We utilized the NUMA patch of Martin Bligh (IBM LTC) for the Linux 2.4.5 kernel to ensure proper interrupt and I/O routing between quads. This particular NUMA patch does not provide NUMA memory abstraction and treats the memory of all nodes as a flat physical memory space. As such all kernel data structures will be located on the first node.The 8-way SMP system is an IBM Netfinity 8500R with 700MHZ PIII processors, 2MB caches and 2.5GB of main memory. The kernel version used on this machine was 2.4.7.

We study the effect of *load*, *poolsize*, and *workload nature* on the overall performance for the five different load balancing methods LBOFF, IP, LBC, LBP-10 and LBP-45. Since, as seen in section 6 the workload nature is one of the overriding differentiators, we show results for each type of workload separately.

7.1 Mkbench Evaluation

To compare the performance and scalability of DSS, MQS and PMQS, we ran a series of kernel builds with varying job sizes on the 16-way NUMA machine. The load on the system is determined by the number of simultaneous kernel builds and the job size of each build.

For a 4x4 NUMA system, a poolsize of 4 is a natural selection for PMQS, as it assigns a single pool to each node. This limits scheduling and data lookup within local nodes and only migrates processes across nodes during the LB phases.

Scheduler	B=2	B=4	B=8
	(32)	(64)	(128)
DSS	-3.93	-3.25	-3.47
LBOFF	5.27	16.30	23.28
IP	5.05	12.90	21.29
LBP-45	2.01	4.19	3.91
LBP-10	2.09	3.55	2.22
LBC	5.89	8.03	7.44

Table 2: PMQS (poolsize=4) as compared to MQS,DSS for Mkbench configurations with varying number (B) of kernel builds on a 4x4-way NUMA system.

Table 2 shows the results for poolsize=4, the job size of 16 and LB invocation frequency of 600 milliseconds for 2, 4 and 8 parallel kernel builds (B=2,4,8). In this setup, B responds to the per CPU load and to an average system wide load of 32, 64 and 128 runnable tasks respectively. First, Table 2 shows that the DSS scheduler under performs the MQS scheduler consistently between 3.25% and 3.93% for B=2,4,8. This corresponds to the results published earlier in [5]. Overall PMQS consistently outperforms MQS across all considered loads and configurations.

In general, LBOFF performs best. The reason for this is that parallel kernel builds are throughput oriented parallel applications. Kernel compiles create a dependency graph, and upon finishing the compilation of an individual file, the next one is started. Hence, the rate of progress for individual compiles does not hinder overall completion time. MQS and DSS both are schedulers that take global priorities into account and hence tend to migrate tasks to ensure the best global scheduling decisions. On NUMA machines crossing node boundaries increases the negative cache effects. Among the dynamic load balancers, LBC, the most aggressive load balancer, performs better than the LBP-45 and LBP-10. This seems somewhat surprising, as one might expect that tighter load balancing of a parallel, mostly independent throughput oriented application creates unnecessary overhead. We believe that this can be attributed to the fact that LBC creates less overhead than LBP for poolsize=1.

To study the effect of overhead associated with load balancing we varied the invocation frequency for LBC from 200 milliseconds to 2 seconds for poolsize=4 and 4 kernel builds. The results, shown in Table 3, show the general trend that less frequent LB invocation, thus lowering the overhead associated with LB, in general increases performance.

Overall, PMQS had a maximum performance advantage of 23.28% for high load (B=8) and LBOFF and a minimum performance increase by 2.01% for low load (B=2) and LBP-45 when compared to MQS.

Analyzing the impact of increased load (B=2,4,8), it is shown that for non-periodic load balancers (LBOFF and IP), the %-improvement over MQS increases with the load. This is again explained by the fact that MQS tends to make global decisions, thus forcing more task migration and hence loss of cache state. Periodic load balancers do not show such dramatic %-improvements over MQS and actually peak at medium load.

We also studied the effect of changing poolsizes. Table 4 shows the results presented in Table 2 for poolsize=8. The trend for poolsize=8 are very similar to those for poolsize=4. However, the performance improvements are not as significant. The reason for is that during scheduling intra-pool balancing as performed by the basic PMQS scheduler results in more task migrations. For LBC we also measured the performance for poolsize=16 and we actually see relative perfor-

	200	400	600	800	1000	1200	1400	1600	1800	2000	LBOFF
poolsize=4	7.45	8.00	8.03	7.67	8.93	8.86	8.35	8.51	9.35	9.64	16.30
poolsize=8	2.55	2.52	2.56	3.01	2.69	3.72	3.26	3.99	3.79	4.67	6.02

Table 3: Impact of LB invocation frequency (in msecs) for LBC as compared to MQS for poolsizes 4 and 8 and Mkbench(B=4).

Scheduler	B=2	B=4	B=8
	(32)	(64)	(128)
DSS	-3.93	-3.25	-3.47
LBOFF	3.02	6.02	16.85
IP	2.09	13.02	14.26
LBP-45	0.57	1.62	1.19
LBP-10	-0.13	0.62	-0.13
LBC	1.98	2.56	2.18

Table 4: PMQS (poolsize=8) as compared to MQS,DSS for Mkbench configurations with varying number (B) of kernel builds on a 4x4-way NUMA system.

mance degrations as compared to MQS of 2.27%, 1.31% and 1.01% for B=2,4,8 respectively.

Having evaluated the efficacy of PMQS for NUMA based systems for kernel compiles, and having estabilished that a poolsize equal the number of cpus per node provides the greatest benefit, we now turn our attention to whether providing smaller poolsize bears any effect. For that we executed the parallel kernel builds (B=1,2,4,8) on an 8-way Netfinity SMP system. The "-j" factor was chosen as 8 to again provide the same load per CPU as in the NUMA system. We varied the poolsize from 1 to 8. The results are presented in Table 5 and are relative to MQS performance. First we observe that the DSS and MQS have only marginal differences in their performances across all loads B.

LBOFF and IP are extremely sensitive to low load situations (B=1,2) and small poolsizes and substantially underperform MQS. Both show good improvements only for B=4 and poolsize=2,4. In general, LBP-45, LBP-10 and LBC demonstrate small overall performance improvements throughout the configuration space of poolsizes and buildfactors considered. Furthermore, for high loads (B=8) no meaningful difference with respect to MQS can be established independent from the poolsize. Though no definite selection can be made, in general LBs outperform LBOFF and IP. In particular LBP-45 seem to show the best overall performance while running Mkbench. We note that LBOFF with poolsize=8 is effectively an MQS scheduler with the changes made to idle process identification.

Overall for NUMA system we have shown that PMQS provides increasingly better performance as compared to MQS when the load increases and when the poolsize is equal the number of cpus in the system. LBOFF and IP provided by far the largest benefits. In contrast, we have shown for a single SMP, that LBOFF and IP have the opposite effect when the poolsize is decreased.

This evaluation suggest that Mkbench on a NUMA system might benefit from a mixed load balancing approach wherein intra node balancing is performed based on LBs and internode balancing is performed using no load-balancing or IP.

7.2 Chat Evaluation

The Chat benchmark was run for three different configurations ranging from 10 rooms, 100 messages per user to 30 rooms, 300 messages. For brevity, these configurations are labelled (10,100), (20,200) and (30,300) where the first number refers to the number of rooms and the second one refers to the number of messages.

The results for the 4x4-way NUMA system with poolsize 4, 8 and 16 are shown in Table 6. MQS improves over DSS between 277% and 482% for this scheduler intensive benchmark. For PMQS, in general we observe the almost inverse behavior when comparing

PoolSize	B=1	B=2	B=4	B=8
DSS				
N/A	0.48	0.95	-0.42	-1.37
LBOFF				
1	-34.10	-9.55	3.18	0.25
2	-17.57	-3.96	10.88	3.12
4	-12.72	4.05	13.59	-0.21
8	2.20	3.37	3.11	0.47
IP				
1	-19.71	-10.51	-0.43	-1.02
2	-9.00	-6.60	15.02	-0.83
4	-10.88	-2.89	12.98	-8.35
8	-7.58	-10.91	-11.99	-1.09
LBP-45				
1	2.82	3.61	2.75	0.30
2	2.28	-4.03	5.70	4.26
4	0.16	1.07	8.34	0.52
8	5.69	3.68	3.19	0.23
LBP-10				
1	5.47	4.37	3.07	0.39
2	-1.77	-2.53	4.73	0.62
4	3.30	4.07	4.33	-0.25
8	4.11	3.09	3.01	0.24
LBC				
1	2.22	-2.49	2.98	-0.18
2	0.23	3.58	3.23	0.39
4	4.49	0.53	3.32	-0.06
8	0.91	2.37	2.20	-0.13

Table 5: PMQS as compared to MQS,DSS for Mkbench configurations with varying number (B) of kernel builds on an 8-way SMP system.

Chat and Mkbench. Here, all PMQS versions either substantially underperform MQS or break even. The higher the load, the worse the performance degradation. Increasing the poolsize also increases the performance for all PMQS but LBP-45. The reason for this is that Chat has, due to the send/recv interactions between threads, rapidly changing runqueue lengths, which then trigger load balancing as observed in Figures 4-6. It is likely that the frequent load balancing leads to continuously moving tasks, often before they get a chance to run. Investigating this aspect is part of our future work.

We now turn our attention to the SMP case to answer the question whether pooling within an SMP provides benefits. The results are shown in Table 7. With the exception

Scheduler	(10,100)	(20,200)	(30,300)
DSS	-63.92	-74.95	-79.26
Poolsize=4			
LBOFF	-27.59	-50.29	-53.70
IP	-39.05	-53.62	-59.29
LBP-45	0.65	-15.26	-25.34
LBP-10	-13.92	-18.38	-19.25
LBC	-1.14	-9.54	-4.41
Poolsize=8			
LBOFF	-7.55	-40.64	-51.32
IP	-27.38	-40.98	-47.15
LBP-45	-16.32	-30.18	-25.74
LBP-10	2.76	-14.85	-14.27
LBC	-3.27	-0.66	-0.23
Poolsize=16			
LBC	-1.42	-2.44	-4.62

Table 6: PMQS as compared to MQS,DSS for Chat on a 4x4-way NUMA system.

of LBC at very high load, PMQS substantially underperforms MQS. PMQS still outperforms DSS. Two general trends can be observed within each category. Performance increases for any given load and LB algorithm with increasing the poolsize and performance increases with load.

8 Conclusions and Future work

In our previous work, we addressed the scalability limitations of the default Linux scheduler (DSS). We proposed a Multi Queue Scheduler (MQS) which used per-CPU runqueues instead of a single global runqueue. However, to maintain strict functional equivalence with DSS, MQS continued to examine all runqueues, albeit intelligently, to make global scheduling decisions. In this paper, we take the work one step further and present a Pooled Multi Queue Scheduler (PMQS) based on MQS. The processors of an SMP are divided into pools for the purpose of scheduling decisions, reducing the number of remote CPU runqueues that need to be examined. As this can lead to load imbalances, we have complemented PMQS with a number of load balancers.

We evaluated the performance of PMQS

PoolSize	(10,100)	(20,200)	(30,300)
DSS			
N/A	-44.03	-73.90	-81.97
LBOFF			
1	-50.54	-49.98	-47.17
2	-34.43	-42.67	-44.68
4	-24.98	-10.63	-11.50
8	-16.17	-5.65	-6.90
IP			
1	-49.19	-53.84	-48.43
2	-34.59	-41.88	-43.84
4	-17.90	-15.09	-10.91
8	-16.80	-9.01	-8.22
LBP-45			
1	-39.51	-25.90	-13.98
2	-28.31	-14.98	-2.88
4	-12.38	-3.66	-2.53
8	-14.79	-6.20	-5.98
LBP-10			
1	-35.17	-11.18	-14.73
2	-30.70	-8.64	-1.29
4	-20.34	-5.88	-1.09
8	-12.85	-5.09	-7.75
LBC			
1	-31.08	-13.90	-0.52
2	-29.06	-7.06	3.36
4	-19.95	-4.45	2.83
8	-16.00	4.22	8.31

Table 7: PMQS as compared to MQS,DSS for Chat on an 8-way SMP system.

and the different load balancers against MQS and DSS on a 4x4-way NUMA system and on an 8-way SMP using two benchmarks. The Mkbench benchmark, which is throughput oriented, benefitted overall from PMQS while the Chat benchmark did not. We believe that Mkbench is more representative of server workloads as it consists of largely unrelated tasks running for short time periods. Chat is more of a microbenchmark with very strong interactions between a large number of tasks leading to a very high rate of scheduling decisions. Different conclusions were also drawn about the relative performance of the load balancers.

The pooling scheduler and load balancers chosen for study are preliminary implementations of the general concept of subdividing processors into pools and regulating load across them. The choice of these implementations was dictated by simplicity and a desire to make incremental changes to MQS.

The performance evaluation has some lessons for future work. First, pooling does show benefits over and above those seen by multi queue schedulers alone. Aggressive load balancing is generally a bad idea as it tends to overcorrect for load imbalances and leads to excessive task migrations.

As part of our future work, we will look at load balancing algorithms which try to balance loads asymptotically. We will also take a fresh look at the approach of running a load balancing module periodically. It might be better to integrate load balancing *functionality* into the scheduler code.

Overall we believe that PMQS is a very flexible extension to MQS and both have shown small to significant performance improvements over DSS. The pooling approach has shown promise and merits further investigation.

9 Acknowledgments

We would like to thank the many people on the `lse-tech@lists.sourceforge.net` mailing list who provided us with valuable comments and suggestions during the development of these alternative scheduler implementations. Christine Moore of the Open Source Development Lab has been been very helpful in providing access to 16-way NUMA systems. This work was developed as part of the Linux Scalability Effort on SourceForge (`lse.sourceforge.net`). Here you can find more detailed descriptions of our scheduler implementations as well as the latest source code.

References

[1] Daniel P. Bovet and Marco Cesati. Understanding the Linux Kernel. OŔeilly Associates.

[2] R. Bryant and B. Hartner. Java Technology, Threads, and Scheduling in Linux. *Java Technology Update*, 4(1), Jan 2000.

[3] R. Bryant and J. Hawkes. Lockmeter: Highly-Informative Instrumentation for Spin Locks in the Linux Kernel. In *Proc. Fourth Annual Linux Showcase and Conference, Atlanta*, Oct 2000.

[4] S. Curran and M. Stumm. A Comparison of basic CPU Scheduling Algorithms for Multiprocessor Unix. *Computer Systems*, 3(4):551–579, October 1990.

[5] M. Kravetz, H. Franke, S. Nagar, and R. Ravindran. Enhancing Linux Scheduler Scalability. In *Proceedings of the Ottawa Linux Symposium, Ottawa, CA*, July 2001.

[6] S. Molloy and P. Honeyman. Scalable Linux Scheduling. In *Usenix Annual Technical Conference (Freenix Track)*, June 2001. To appear.

[7] M. S. Squillante and Edward A. Lazowska. Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling . *Transactions on Parallel and Distributed Systems*, 4(2):131–143, February 1993.

[8] M. S. Squillante and Edward A. Lazowska. Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling . *Transactions on Parallel and Distributed Systems*, 4(2):131–143, February 1993.

[9] J. Torellas, A. Tucker, and A. Gupta. Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 24(2):139–151, February 1995.