Enhancing Linux Scheduler Scalability

Mike Kravetz
IBM Linux Technology Center

Hubertus Franke, Shailabh Nagar, Rajan Ravindran
IBM Thomas J. Watson Research Center
{mkravetz,frankeh,nagar,rajancr}@us.ibm.com
http://lse.sourceforge.net

Abstract

This paper examines the scalability of the Linux 2.4.x scheduler as the load and number of CPUs increases. We show that the current scheduler design involving a single runqueue and lock can suffer from lock contention problems which limits its scalability. We present alternate designs using multiple runqueues and priority levels that can reduce lock contention while maintaining the same functional behavior as the current scheduler. These implementations demonstrate better overall scheduling performance over a wide spectrum of loads and system configurations.

1 Introduction

Linux has seen tremendous growth as a server operating system and has been successfully deployed in enterprise environments for Web, file and print serving. Often, the increased demand in such environments can be met by horizontally scaling the system with clustering. For such applications, the operating system needs to efficiently support SMPs consisting of only a small number of CPUs.

More demanding applications, such as database, e-business or departmental servers, tend to be deployed on larger SMP systems. To support such applications, Linux must scale well *vertically* as more CPUs are added to an SMP. It must also scale with the increased number of processes and threads that such SMPs are expected to handle. In both these situations, the scheduler can be a key factor in achieving or limiting operating system scalability. 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.

Lock contention problems can generally be addressed in two ways. First, the protected data structure can be reorganized so that it can be traversed faster with a corresponding decrease in the average lock hold time. Second, the data structure can be broken up or partitioned into smaller parts, each protected by its own separate lock. This reduces the probability of lock contention overall. Additionally, it allows multiple examinations of the subparts to proceed in parallel, reducing lock wait time for the data structure as a whole.

The main contribution of this paper is the design, implementation and evaluation of two new Linux schedulers which improve scalability using these two approaches. The priority level scheduler (PLS) aims at reducing lock hold time by maintaining runnable tasks in priority lists. The multiqueue scheduler (MQ) reduces lock contention by maintaining per-cpu runqueues. Both of these solutions are deployed on commercial operating systems but have not been seriously considered for Linux. Though priority level schedulers have been implemented for Linux [5] and have shown improvements over the vanilla scheduler, we show here that the reduction in lock hold time by such methods only improves scalability with an increased number of tasks. However, it is not sufficient to improve scalability with increasing CPU counts. In particular, though our PLS also does better than the current scheduler at moderate to high task counts, MQ outperforms the current scheduler *and* PLS over a wide range of workloads. More importantly, these improvements are obtained while maintaining functional equivalence with the current scheduler, leaving room for further improvements.

The rest of the paper is organized as follows. Section 2 presents a description of the implementation of the current scheduler. The parts which define the functionality (and need to be retained) are identified along with the bottlenecks. Section 3 presents the priority queue scheduler implementation. The main contribution of this paper, the multiqueue scheduler, is described in Section 4. Results using microbenchmarks and a decision support workload are shown in Section 5. Section 6 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. The responsiveness of lengthy kernel code can be increased by checking for scheduling requirements at appropriate locations. 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. In such cases, the `schedule()` function is called if the `need_resched` field of the current task is set.
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 [5, 1].

3 Priority Level Scheduler (PLS)

The priority level scheduler (PLS) seeks to reduce the number of tasks examined during a scheduling decision. It reorganizes the single runqueue of the default SMP scheduler (DSS) into an array of lists indexed by the non-affinity goodness of tasks. The indices of the currently running task, and the highest schedulable task together with an affinity boost, determine the range of lists to be searched for the next candidate. The priority lists are still protected by a single runqueue lock as they conceptually provide a single runqueue.

In our implementation, we coalesce all realtime tasks into a single list at the highest index. This method of enqueueing tasks results in 61 lists for the x86 platform and up to 335 lists for other architectures. A task's goodness value can change during its execution, e.g. during `fork, timer, exit`, and `recalculate`, requiring it to be reassigned to a different priority list. To avoid frequent requeueing, yielding tasks are enqueued according to their non-yield goodness values and handled appropriately while walking the lists. The implementation ensures that yielding tasks do not execute before any other runnable task.

At `schedule()` time, the currently running task is the default candidate to run next, and if it is not yielding, also establishes the lowest list to be scanned (as no task on a lower list can receive an affinity boost which results in a priority higher than that of the currently running task.) If the task stopped executing, e.g. due to I/O wait, the `idle-task` becomes the default candidate and all lists need to be searched by default. Tasks with expired counters fall into the lowest list and are never inspected.

The determined range of lists is now scanned in top-down priority order for non-yielding schedulable tasks and if one is found and its goodness value is better then the default candidates, it becomes the default candidate. Further search can be limited to lists whose priority lie within `PROC_CHANGE_PENALTY` of the default candidate's list, as no list below that can have a higher goodness value even after getting an affinity boost. Even within this range, the search can be terminated as soon as a task is found that last ran on the invoking CPU. As a further optimization, we maintain a bitmap of non-empty list indices that allows us to efficiently skip empty lists, using the `find_first_zero()` function. We disregard the `tsk->mm` boost, which essentially provides a tie-breaker between two task of equal priority, as it would require a complete scan of the last reached list and the one below it. We have also implemented versions of priority level schedulers that account for the `tsk->mm` boost but only observed infrequent differences in scheduling behavior compared to DSS, while suffering from degraded performance. We chose to present the best performing PLS implementation to highlight the need for reducing lock contention as done in MQ. We have also implemented versions that limit the number of lists, by utilizing a different hash function, but did not observe performance improvements. Since PLS keeps running tasks on the runqueue (i.e. in their list) and therefore inspects these tasks during scheduling, we can expect that for low task counts ($\approx$ #CPUs), PLS will introduce additional overhead compared to DSS. However, with the increase in the number of tasks, the probability of finding a task that ran last on the invoking CPU increases as does the benefit of limiting the number of tasks that need to be traversed. Together, we expect an reduced average lock hold time.

4 Multi-Queue Scheduler (MQ)

The multi-queue scheduler (MQ) 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. Moreover, 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.

MQ 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 MQ 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 next. Schedulers incorporating only this phase exist [4], but can lead to problems of priority inversion and load imbalances amongst the runqueues. The load imbalance problem is illustrated in Fig 1 which shows the deviations from the mean runqueue length over time for 4-way SMP executing a kernel build and using such a restricted multi-queue scheduler. MQ directly addresses priority inversion in the second phase by comparing the local candidate with the top candidates from remote runqueues before making the final selection. This also has a load balancing effect.

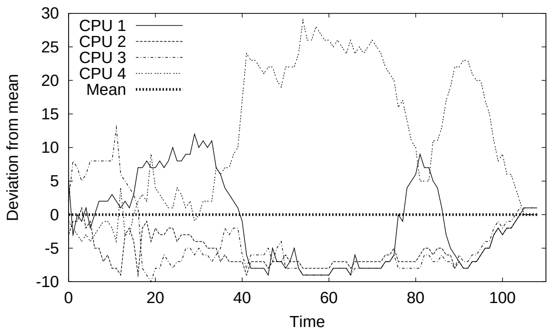

Figure 1: Deviation from mean of runqueue lengths for a 4-way SMP during a kernel build and running a scheduler which only looks at the local runqueue

In more detail, the `schedule()` routine of MQ 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 best 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 the 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, MQ'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 MQ.

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.

MQ'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.

An important aspect of MQ's implementation is the care taken to avoid unnecessary cache misses and false sharing. Runqueue data is allocated in per-CPU cache-aligned data structures.

#CPUs	#invocations of `schedule`	Run queue length	
		Mean	Maximum
2-way	241817	4.93	18
4-way	308396	7.25	23
8-way	816135	8.21	35

Table 1: Runqueue lengths for TPC-H on DSS

5 Performance Evaluation

To assess the scalability of the various schedulers presented, we show the performance impact of increasing the number of CPUs and number of runnable tasks. We first show the scope of the problem through statistics collected using the industry standard TPC-H benchmark. Then we use two microbenchmarks to control an increase in load and evaluate the performance of the three scheduler designs.

All benchmarks were run on an 8-way IBM Netfinity 8500R with 700MHZ PIII processors, 2MB caches and 2.5GB of main memory. We varied the CPU numbers via the `maxcpus` boot parameter and increased the offered load through benchmark parameters. All tests were run using the 2.4.3 distribution of the Linux kernel.

5.1 TPC-H

TPC-H is an industry standard decision support benchmark consisting of ad-hoc database queries. For a detailed description of TPC-H please see `http://www.tpc.org`. We chose this benchmark to represent a real world workload often serviced by large SMP systems. Due to the intricacies of publishing actual TPC-H results, we focussed our attention on the lock contention analysis rather than the normally reported metrics. Hence, we spent very little time tuning either the system or the database for optimal performance. The intent here is to motivate our work by showing the extent of the lock contention problem in a realistic workload. The benchmark is run with a sufficiently small database to minimize disk I/O.

Table 1 shows the length of the runqueue and the calls to the `schedule()` function for DSS, as the number of CPUs is increased from 2 to 8. The size of the database is kept constant but its degree of parallelism is increased in proportion to the number of CPUs. As Table 1 shows, the system is fully loaded with the average number of runnable tasks exceeding the number of CPUs.

	2-Way	4-Way	8-Way
DSS			
Contention	2.4%	9.6%	47.2%
Mean Hold Time	1.5us	2.2us	3.9us
Mean Wait Time	2.8us	3.9us	10us
PLS			
Contention	2.0%	13.6%	53.7%
Mean Hold Time	1.7us	2.6us	4.4us
Mean Wait Time	3.1us	4.2us	11us
MQ			
NF Contention	1.9%	8.1%	11.4%
Contention	3.3%	9.6%	14.2%
Mean Hold Time	1.7us	2.0us	2.8us
Mean Wait Time	2.0us	3.2us	3.0us

Table 2: Lock statistics for TPC-H

Table 2 shows statistics for the lock contention for the runqueue lock. The results were collected by running kernels instrumented with Lockmeter [3]. We clearly see that lock contention is a significant problem as the number of CPUs increases, with the resulting lock wait times rising from 2.8us on a 2-way to 10us on the 8-way for DSS and and similar increases for PLS. PLS does exhibit higher lock contention and lock hold times then DSS. This reflects that PLS's overhead at the low mean runqueue length outweighs the expected gains at the high thread count. MQ does much better at controlling the lock contention. On an 8-way system, the lock contention is only 11.4% as compared to the 47.2% in DSS and 53.7% in PLS. The lock wait times show a corresponding decrease with MQ doing 31% better than DSS on an 8-way.

For MQ we report another lock contention measure called NF(Non-failure) Contention. By default, failed `spin_trylock` attempts contribute to the total contention for a lock as reported by the lockstat tool. However, such failed attempts are relatively inexpensive and do not result in wasted CPU cycles caused by spinning. The MQ scheduler makes use of `spin_trylock` in attempting to modify data on non-local runqueues. In contrast, the DSS and PLS schedulers never make use of `spin_trylock`. The measurement NF Contention is lock contention computed without including calls to `spin_trylock`.

For the 2-way and 4-way systems, we observe the same overall performance amongst the three schedulers. However, on the 8-way system MQ does 6% better than the other two which agrees with the marked difference in lock contention numbers reported here.

5.2 Chat

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.

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 number of rooms and the second one refers to the number of messages. To better understand the load seen by the scheduler in these configurations, Fig 5 shows a histogram of the number of tasks on the runqueue during every invocation of `schedule()` on an 8-way system executing the DSS scheduler. The mean and

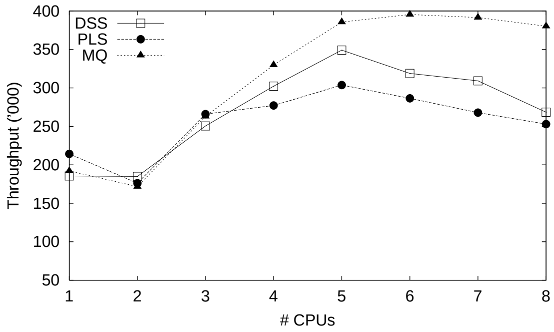

Figure 2: Chat (10,100)

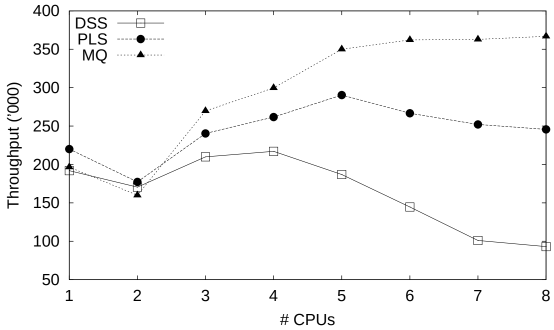

Figure 3: Chat (20,200)

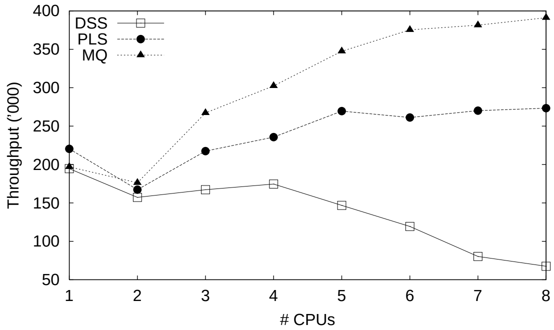

Figure 4: Chat (30,300)

maximum values of the runqueue length for nine configurations and all three schedulers are shown in Table 4. Per-CPU values of the runqueue length are shown for MQ and are much lower than those for DSS and PLS as expected. For DSS and MQ, the lock hold times are proportional to the runqueue lengths shown. The data also confirms the ability of the Chat benchmark parameters to manipulate the load.

Fig. 2 compares DSS, MQ and PLS for the (10,100) configuration. For 2 and 3 CPUs, all three perform comparably. The highest throughput achieved by MQ is 13% higher than that of DSS, while PLS does

DSS	#CPUs						
	2	3	4	5	6	7	8
Contention	18.3%	31.2%	42.6%	54.4%	66.6%	77.5%	86.6%
Hold Mean(Max)	15 (118)	13 (164)	14 (187)	19 (253)	22 (258)	21 (287)	22 (302)
Wait Mean(Max)	26 (105)	24 (1045)	34 (1270)	58 (1633)	78 (2886)	87 (3753)	105 (4197)
PLS	2	3	4	5	6	7	8
Contention	3.7%	10.7%	15.6%	22.2%	29.6%	43.4%	50.7%
Hold Mean(Max)	2.3 (28)	2.9 (101)	2.9 (56)	3.8 (114)	4.2 (97)	6.7 (212)	7.3 (170)
Wait Mean(Max)	2.7 (23)	4.0 (79)	3.4 (43)	6.0 (125)	6.1 (180)	16 (379)	19 (498)
MQ	2	3	4	5	6	7	8
Contention	5.0%	5.2%	3.5%	5.8%	6.0%	6.7%	6.3%
Hold Mean(Max)	3.2 (8.6)	2.9 (33)	2.3 (290)	2.5 (26)	2.5 (35)	2.7 (34)	2.9 (37)
Wait Mean(Max)	4.3 (30)	3.6 (25)	2.8 (20)	2.9 (22)	3.1 (22)	3.1 (26)	3.4 (18)

Table 3: Lock statistics for Chat (20,200)

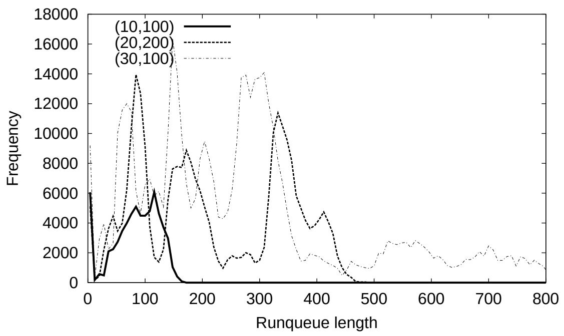

Figure 5: Distribution of runqueue lengths for Chat running on an 8-way using the DSS

Config.	Run queue length mean(max)		
	DSS	PLS	MQ
(10,100)	84(165)	82(234)	6(23)
(10,200)	135(276)	130(299)	14(52)
(10,300)	160(306)	167(353)	18(47)
(20,100)	82(169)	122(364)	7(26)
(20,200)	218(484)	198(503)	24(70)
(20,300)	217(571)	307(564)	31(95)
(30,100)	113(232)	176(423)	6(28)
(30,200)	276(763)	270(827)	36(94)
(30,300)	279(899)	443(816)	47(113)

Table 4: Runqueue length statistics for Chat (20,200) on DSS

13% worse. More importantly, MQ scales better than DSS and PLS as seen by the gradual decrease from the point at which maximum throughput is reached. Table 5 summarizes the speedup achieved by the schedulers. The ideal speedup values are reported using a 2-way system as the base as it is the minimum SMP configuration.

In Figs 2, 3 and 4, results for 1-way are shown only for completeness. The numbers were obtained by running an SMP-enabled kernel with 1 CPU. Typically, the uniprocessor kernel would be used instead.

Moving on to the (20,200) configuration, we see that the differences amongst the schedulers start showing up immediately after 2 CPUs. The peak throughputs of MQ and PLS are 68% and 34% above those of DSS. MQ's performance does not show any degradation for CPU counts scaling up to 8-way as seen in Fig. 3 and in Table 5.

Increasing the load still further, we find the DSS scheduler unable to scale in the (30,300) configuration. In fact its maximum throughput, achieved on a 4-way, is only 11% higher than the 2-way performance and rapidly declines thereafter. Both PLS and MQ scale well and do not show any performance degradation at higher CPU counts. The peak throughputs of MQ and PLS are 124% and 57% higher than those of DSS respectively.

While DSS scales up to 5 CPUs in the (10,100) configuration, it starts breaking down rapidly at higher CPU counts or increased loads. Our hypothesis, as mentioned earlier, is that the increased lock contention is the basic problem exacerbated by the increased lock hold time due to the linear search. To verify this, in Table 3, we show statistics on the runqueue lock for the (20,200) configuration. With increasing CPU counts, the DSS scheduler causes lock contention to increase substantially from 18.3% on the 2-way to 86.6% on the 8-way. This causes an increase in lock wait time, with the maximum wait time increasing from 105 microseconds on the 2-way to 4197 microseconds on the 8-way. As expected,

	Maximum		On 8-way
	Achieved at CPU#	Value (Ideal)	Value (Ideal)
(10,100)			
DSS	5	1.88(2.5)	1.45(4.0)
PLS	5	1.72(2.5)	1.43(4.0)
MQ	6	2.31(3.0)	2.21(4.0)
(20,200)			
DSS	4	1.27(2.0)	0.54(4.0)
PLS	5	1.32(2.5)	1.43(4.0)
MQ	8	2.29(4.0)	2.29(4.0)
(30,300)			
DSS	4	1.11(2.0)	0.42(4.0)
PLS	8	1.63(4.0)	1.63(4.0)
MQ	8	2.22(4.0)	2.22(4.0)

Table 5: Speedup for Chat

the lock hold times do not increase in the same proportion since they primarily depend on runqueue length. Comparing PLS to DSS, we see a significant decrease in lock contention. This is primarily due to the decrease in lock hold time. Reduced lock contention and lock hold times together result in substantially reduced lock wait times. For MQ, we report the lock statistics averaged over the per-CPU runqueue locks. We see that lock contention is virtually eliminated, ranging between 5.0% and 6.7%. The low lock hold times also reflect the distribution of tasks amongst the runqueues. Due to the low lock contention, lock wait times are much significantly lower than those for the other schedulers.

5.3 Reflex

Reflex is a microbenchmark designed to exercise the `schedule()` and `reschedule_idle()` functions in a controlled way. The program creates a number of threads which are grouped into sets called *active sets*. All threads in one active set pass a token around using blocking reads and writes on message pipes. After receiving the token, a thread performs several rounds, each consisting of some computation followed by an explicit yield of the processor. In the last round, instead of yielding, it passes the token onto its neighbor in the active set and blocks on a read for the same token. The explicit yields result in the invocation of `schedule()` (through the system call `sys_sched_yield()`) whereas the blocking reads on the message pipe result in `reschedule_idle()` being called.

Reflex reports the average time spent per round (obtained by dividing the benchmark run time by the total number of rounds performed by all threads). The round time consists of the average computation time and the scheduling time. The computation time per round is a parameter of the benchmark. For the results shown here, we choose it to be zero so that the scheduler overhead is overexposed and the differences between the three schedulers can be clearly seen. Another important parameter is the number of times a thread yields before passing the token. It is determined by a probability parameter which can effectively control the relative frequency of invocation of `schedule()` and `reschedule_idle()`. The results below are obtained by keeping these frequencies almost equal. The number of runnable threads in the system is determined by parameters which controls the number of tokens (or active sets) and the number of threads created.

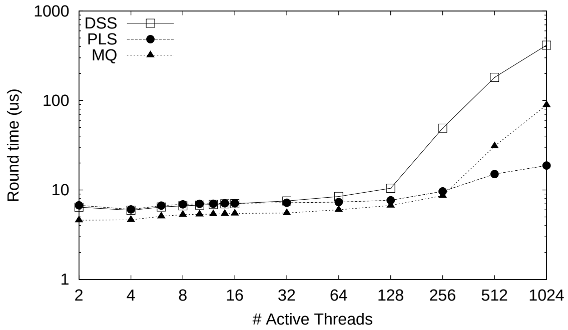

Figure 6: Reflex benchmark on 2-way SMP

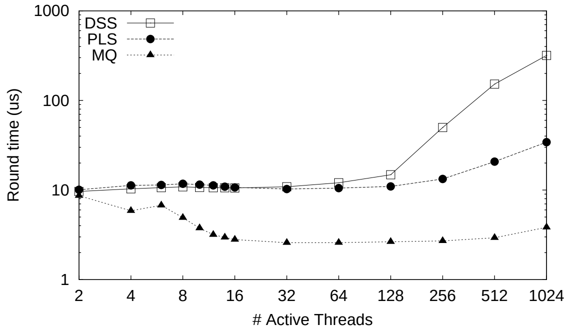

Figure 7: Reflex benchmark on 8-way SMP

Fig 6 compares the three schedulers on a 2-way system as we increase the number of active sets which is the same as the number of active threads on an average. The y-axis is shown on a log scale. At low thread counts (up to 32 runnable threads), PLS and DSS are almost equivalent while MQ does better. At higher thread counts, MQ and PLS out-

	#CPUs			
	2	4	6	8
DSS				
Contention	27.3%	88.4%	92.6%	93.0%
Hold Mean(Max)	2.0 (7.8)	2.8 (11)	3.7 (14)	4.6 (21)
Wait Mean(Max)	2.6 (9.2)	8.0(131)	22 (631)	40 (777)
PLS				
Contention	32.7%	87.8%	92.1%	92.8%
Hold Mean(Max)	2.1 (10.0)	2.9 (15)	3.8 (24)	4.7 (25)
Wait Mean(Max)	2.7 (12)	8.0(130)	22 (654)	40 (953)
MQ				
Contention	2.3%	5.7%	6.4%	11.3%
Hold Mean (Max)	1.0 (7.2)	1.3 (9.4)	1.9 (22)	3.0 (52)
Wait Mean (Max)	2.0 (5.4)	2.1 (21)	2.7 (35)	3.3 (48)

Table 6: Lock statistics for Reflex with 16 runnable threads

perform DSS significantly. Beyond 256 runnable threads, MQ performance is affected by scanning long runqueues and hence it does worse than PLS which is able to arrive at a scheduling decision in almost constant time.

Fig 7 compares scheduler performance on an 8-way SMP. While PLS and DSS demonstrate the same characteristics as in the 2-way graph, MQ does better than both of them at all thread counts and shows better scaling.

Table 6 shows the lock statistics for Reflex with 16 runnable threads. We see the same trends as observed in the Chat benchmark, with DSS and PLS causing lock contention up to 93% on an 8-way system. In contrast, even though the lock hold times are comparable in all three schedulers, MQ shows significantly lower lock contention.

6 Conclusion and Future Work

The Linux 2.4 kernel provides a concise SMP scheduler that does well for small SMPs running moderate loads. However, we have shown that, as the number of CPUs or the load increases, the scalability limitations of the scheduler start showing up. Profiling data for a range of workloads show that the problem is due to high lock contention and large lock hold times.

Reducing the lock hold times, as is done in the PLS scheduler presented here, does alleviate the problem somewhat with a corresponding improvement in scalability. However, this is not sufficient to address the overall scalability as the number of CPUs increases. Also, at low loads, the overheads of PLS make it perform worse than DSS. The MQ scheduler directly addresses lock contention by breaking up the single runqueue and its associated locks into per-CPU equivalents. This brings a significant improvement in lock contention, scalability and overall performance of the scheduler.

We are currently working on more extensive evaluations of the ideas presented in this paper. We want to use more realistic workloads such as those seen on compute and database servers. Further extending the MQ design, we are looking at schedulers which use *CPU pooling*. CPU pooling divides the CPUs of a system into a set of pools. Each pool consists of one or more CPUs. Scheduling decisions are localized to the individual CPU pools, and load balancing algorithms are put in place to balance the load among the pools. CPU Pooling provides a continuum between complete runqueue separation, as provided in [4], and MQ with its global scheduling decisions.

It is our belief that CPU pooling will be beneficial on large SMP machines where making global scheduling decisions will become more expensive. In addition, CPU pooling may be a good choice for NUMA architectures where CPUs on individual compute nodes can be mapped to CPU pools.

7 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. In particular, we would like to recognize John Hawkes, for running our implementations on some large systems at SGI, and Bill Hartner for related discussions and help with the experiments. 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] Hewlett Packard Inc. Process resource managers for Linux : Linux plug-in schedulers. http://resourcemanagement.unixsolutions.hp.com /WaRM/schedpolicy.html.

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