USENIX Association

Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference

Boston, Massachusetts, USA
June 25–30, 2001

Scalable Linux Scheduling

Stephen Molloy, ***CITI - University of Michigan***
`<smolloy@engin.umich.edu>`

Peter Honeyman, ***CITI - University of Michigan***
`<honey@citi.umich.edu>`

Abstract

For most of its existence, Linux has been used primarily as a personal desktop operating system. Yet, in recent times, its use as a cost-efficient alternative to commercial operating systems for network servers, distributed workstations and other large-scale systems has been increasing. Despite its remarkable rise in popularity, Linux exhibits many undesirable performance traits.

Concerned about the scalability of multithreaded network servers powered by Linux, we investigate improvements to the Linux scheduler. We focus on pre-calculating base priorities and sorting the run queue for efficient task selection. We propose an improved scheduler design and compare our implementation in terms of scalability and performance to the existing Linux scheduler. Our analysis shows that improvements can be made to the existing scheduler without introducing overhead, thus improving the scalability and robustness of the Linux operating system.

1. Introduction

Linux, a strong and steadily increasing presence on the Internet today [4], commonly provides cost-effective and load-tolerant solutions for network services. Its familiarity to UNIX users, source code availability, and ability to run on many different architectures explain Linux's rapid increase in popularity. Many software companies, including AOL, Netscape and IBM [6, 8], offer Linux products. Several organizations use Linux on routers, print and file servers, firewalls and, of course, web application servers [10].

At the same time as Linux's popularity has increased, the use of Java for web applications has grown immensely. Java technology is a key component in building scalable application servers. However, communications-intensive Java applications often create large numbers of threads and Linux does not handle such stress gracefully.

Concerned about the scalability of multithreaded network servers powered by Linux, we investigate improvements to the Linux scheduler. Experiments by IBM indicate that as much as 30% of the total CPU time in the system is spent in the scheduler when the number of running threads is high [2]. Our analysis shows the current scheduler uses an expensive and redundant algorithm for task selection. Our goal is to improve the scalability of the Linux scheduler to adapt it to enterprise-scale server workloads. Our analysis shows that our new scheduler implementation achieves these goals.

The rest of this paper is organized as follows. Section 2 gives some background information on our project and provides some insight as to why we chose to tackle the scheduler rather than the Linux threading model. Section 3 describes Linux's current approach to scheduling. Section 4 explains the problems with it. Section 5 outlines our approach to solving the problem and Section 6 describes its performance relative to the current Linux scheduler. All kernel modifications, experiments, and descriptions are against a 2.3.99-pre4 Linux kernel and, when used, Java version 1.1.7 of IBM's JDK.

2. Background

Because Linux grew from a desktop operating system, many issues prevent it from being a dominant force in the enterprise server market. The design and implementation of Linux has traditionally focused on simplicity and versatility rather than small performance gains and scalability. The implementation of the Linux thread model and scheduler illustrate this approach.

The Linux thread model is a one-to-one model, meaning that every user-level thread is mapped onto its own kernel thread. While this model makes programming in the kernel less complicated, it sometimes forces programs to generate more kernel threads than is necessary. Forcing the kernel's default scheduler to accommodate too many threads can adversely affect a server's performance.

Many established operating systems support many-to-one or many-to-many thread models in which each kernel thread has many user level threads mapped to it. In these models, a secondary scheduler chooses which of the mapped user level threads to run. The multi-tier scheduling approach of these models assures that the scheduler at each level will be faced with a more manageable number of threads.

The Linux scheduler, like its thread model, is also an exercise in simplicity. The heart of the scheduler is concisely coded in just a few lines that evaluate runnable threads and then picks the best. The price for this simplicity though, is a linear time algorithm that repeats many of the same calculations that were performed on its last invocation.

In this project, we address the shortcomings of the scheduler rather than those of the threading model. The reasons for this decision are simple. We saw the redundant calculation and O(n) loop in the current scheduler and knew we could improve it. We also know that the Linux kernel community has been very protective of its threading model in the past and we wanted to avoid upsetting any contributors. Finally, the original timeline for the project called for a working design and implementation within the time frame of one semester. Developing a new threading model for Linux would almost certainly require more time than we had available.

Other groups have spent considerable time designing alternative schedulers for Linux [1, 5, 9]. Linux discussion groups provide evidence that the scheduler has been and continues to be an interesting topic for the developer community. However, most alternative scheduler designs focus on reducing latency for real-time processes rather than improving the overall scalability of the default scheduler.

`volatile long`	`state`
`unsigned long`	`policy`
`long`	`counter`
`long`	`priority`
`struct mm_struct`	`*mm`
`struct list_head`	`run_list`
`int`	`has_cpu`
`int`	`processor`

Table 1: This table shows the fields of the task structure that are most relevant to Linux scheduling.

While it is our goal to improve scalability and performance of the scheduler when faced with a large number of runnable threads, it is not our intent to change the criteria it uses for thread selection. We feel that the current criteria are carefully chosen and sufficient to make good decisions with a minimum amount of calculation. Our primary concern is simply that these criteria are not being used in an optimal algorithm by the scheduler.

In the remainder of this paper, because Linux uses a one-to-one threading model, we do not distinguish between a user thread and a kernel thread. Also, to match the terminology used in the kernel source code, we refer to any thread in the system as a task.

3. Current Scheduler

To understand why the current Linux scheduler scales poorly with the number of runnable threads in the system, it is necessary to be familiar with its data structures, algorithms, and conventions. This section outlines the existing scheduler to clarify our observations and design decisions.

3.1 Task Structure

The basic execution context in Linux is referred to as a task. The task structure is responsible for maintaining a task's address space information, whether that address space is shared with other tasks, and other state information about the task and its registers. It also tracks task statistics for memory management and resource control, privileges, file descriptors, signal handlers and other task specific information. The various fields of the task structure used in the scheduler are illustrated in Table 1.

The task's `state` field can be set to one of six values, each representing a different state that in which a task might find itself (such as blocking or sleeping.) `TASK_RUNNING` is the value of `state` when a task is runnable.

The `policy` field is set either to `SCHED_FIFO`, `SCHED_RR` (round robin) or `SCHED_OTHER` to determine the scheduling policy for the task. The first two options are for real-time tasks, while the third is for all other tasks. Real time tasks are always run before regular tasks if they are runnable. The `policy` field is also used to track yielded tasks. When a non-real-time task gives up its processor via the `sys_sched_yield()` system call, a bit (`SCHED_YIELD`) in the task's `policy` field is set so this information can be passed on to the scheduler.

The field `has_cpu` is set to 1 while a task is executing on a processor and 0 otherwise. Upon setting `has_cpu`, the field `processor` is set to the processor ID on which the task will execute. The task structure also contains pointers that identify the memory map in which it runs and its place on the run queue.

The two most important factors in determining which task executes next are represented by the `priority` and `counter` fields. `Priority` is an integer between 1 and 40. Higher numbers represent higher priority. Twenty is the default value for all tasks. (Real-time tasks also use a priority value, but it ranges from 0 to 99 and is stored in a separate field called `rt_priority`.) `Counter` is a value that indicates the time remaining in the task's current quantum. `Counter,` measured in 10ms ticks, can range from zero to twice the task's `priority`. Linux uses this field to enforce a fairness policy.

It is worth noting that all tasks, whether they are lightweight threads or full-fledged processes, are treated the same by a Linux system. All processes and threads are visible in various system status commands such as ps and top. Consequently, the default scheduler, which is responsible for accommodating all tasks in the system, can be placed under considerable stress when running multithreaded applications.

3.2 Run Queue

The run queue in Linux is a circular, doubly linked list containing all tasks in the `TASK_RUNNING` state. The scheduler traverses this list when it looks for a task to run. The list is not maintained in sorted order. When the scheduler finds two equivalent tasks, the one closer to the front of the list is chosen. Newly created or awakened tasks are placed at the beginning of the run queue. The list is doubly linked and circular, so tasks can also be added to the end of the run queue.

3.3 Schedule()

The Linux kernel function `schedule()`, as in other operating systems, is called from over 500 places within the kernel, underscoring its significance to overall system performance. The `schedule()` function is called by a task when it yields the processor, blocks for I/O, expires its quantum, or is preempted by another (higher priority) task. `Schedule()` uses the execution context of the task that called it (referred to as the previous task in the scheduler). `Schedule()` is charged with finding the best task to take the previous task's place on the processor. In doing so, it makes use of a heuristic computed by the function `goodness()`.

3.3.1 Goodness Calculation

The scheduler uses the `goodness()` function to determine the utility of running a given task. A high goodness value means it would be a sound decision to run the given task next. For tasks that are marked `SCHED_FIFO` or `SCHED_RR`, `goodness()` returns 1000 plus the value stored in the tasks `rt_priority` field. For other tasks, however, `goodness()` returns a much lower number and shows more discretion in its evaluation.

For `SCHED_OTHER` tasks, four factors are taken into consideration. The first factor is a task's `counter` value. If a task has a `counter` value of zero, then `goodness()` returns a utility of zero. This lets the scheduler know a runnable task was found but its time slice is used up. If a task's `counter` value is not zero, then its goodness value is set to the sum of its `counter` and `priority` values.

The third and fourth factors are bonuses for processor affinity and sharing an address space with the previous task. A small, one point advantage is given to tasks that share memory maps, because of the reduced overhead for the context switch. A somewhat larger (15 point) bonus is given to tasks whose last run was on the current processor, to try to take advantage of memory lines that may still reside in the processor's cache. These bonuses are added to the previously calculated goodness value to determine the task's final goodness value.

3.3.2 Scheduling Algorithm

The scheduler begins by executing all outstanding bottom-halves (delayed functions that were too substantial to run during an interrupt.) After some additional administrative work, the scheduler enters the heart of its code: an examination of all runnable tasks. The previous task is the first task looked at by the scheduler. If the `SCHED_YIELD` bit is set for the previous task, then the scheduler clears the bit and uses zero as the task's goodness value. Otherwise, it calls `goodness()` to determine this value.

Next, the scheduler walks through the run queue, evaluating the goodness of each task not currently running on another processor. After all runnable tasks have been examined, the task with the greatest goodness value is chosen to run on the processor. If no task has a goodness greater than zero[1], then the scheduler jumps to a piece of code responsible for recalculating the `counter` values of all tasks in the system (runnable or otherwise) and returns to search the run queue again.

While the `goodness()` function by itself is very simple, executes quickly and considers the most appropriate factors in making intelligent scheduling decisions, it is expensive to recalculate `goodness()` for every task on every invocation of the scheduler.

4. Problem

Efficient handling of multiple threads is crucial for enterprise servers to make best use of system resources, communicate with many parties at the same time, and reduce the average time that service requests spend waiting for an available server. Multiplexing I/O system calls (such as `select`) can help in some situations, but they are not always available. The popular Java programming language is a prime example.

Threads are an essential element in the Java language: because the Java language lacks an interface for non-blocking and multiplexing I/O, threads are especially important in constructing communications intensive applications. Typically, one or more Java threads are constructed for each communications stream used by a Java program. Therefore, a natively threaded Java Virtual Machine (such as IBM's JVM [7]) can put a strain on the Linux scheduler, which, as we have seen, examines the goodness function for every thread in the run queue. This can be an exhausting process.

Experiments at IBM show the impact of the Linux scheduler on the performance of a multithreaded network application written in Java [2]. VolanoMark is a benchmark written to measure the performance of VolanoChat, a Java implementation of a chat room server. Because its results have been widely published in magazines such as JavaWorld [3], VolanoMark is an important benchmark for comparing the performance of different implementations of the Java Virtual Machine.

The VolanoMark benchmark establishes a socket connection to a chat server for each simulated chat room user. Because Java does not provide non-blocking read and write, VolanoMark uses a pair of threads on each end of each socket connection (4 threads per connection) to simulate non-blocking I/O. For a 5 to 25-room simulation, the kernel must potentially deal with 400 to 2,000 threads in the run queue. The key measure of performance reported by VolanoMark is message throughput, i.e., the number of messages per second (over all connections) the server is able to handle during a benchmark run. The measurements for IBM's report were taken while running VolanoMark over a loopback interface, eliminating any network overhead involved; the heap size for the test was large enough for the overhead of Java garbage collection to be less than 5% of the total elapsed time throughout the experiments.

The results of the VolanoMark experiments show that 25-room throughput decreased by 24% from 5-room throughput due to the additional threads in the system. A profile of the kernel taken during the VolanoMark runs showed that between 37 (5-room) and 55 (25-room) percent of total time spent in the kernel during the test is spent in the scheduler.

5. ELSC Scheduler

To reduce the amount of time spent in the scheduler we developed a new scheduling solution, called the ELSC scheduler. Our goals in implementing this scheduler are as follows:

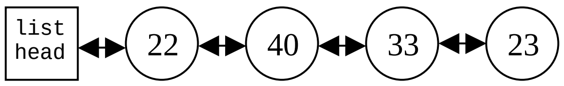
1) Keep changes local to the scheduler. Do not change current interfaces to the scheduler.
2) Keep the concept and implementation simple.
3) Behave like the current scheduler as much as possible.[2]
4) Maintain existing performance for light loads. Scale gracefully under heavy loads.

The ELSC scheduler is a table-based scheduler that keeps the run queue in a sorted order, making scheduling decisions easier and faster. We chose a table based design because it relieves us of the overhead of sorting lists. It also avoids complexity when inserting or removing tasks, unlike, say, a heap.

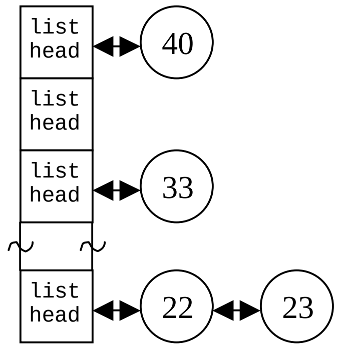
The foundation of the ELSC scheduler is its ability to keep tasks in an order that makes choosing one fast. The key to this sorted order is in how a task's `goodness()` value is calculated in the current scheduler. The `goodness()` calculation consists of a static and a dynamic part. The static part consists of a task's `priority` and `counter` values. While a task is

[1] The run queue must contain at least one task for this condition to count. An empty run queue will schedule the idle task rather than trigger the recalculation.

[2] By behave, we mean that if the current scheduler always selects a real-time task over a SCHED_OTHER task, even if it has a zero counter, then the ELSC scheduler should do the same. Aside from a few optimizations, the ELSC scheduler does adhere to the same quirky rules as the current Linux scheduler.



(a) Run Queue for Current Linux Scheduler



(b) Run Queue for ELSC Scheduler

Figure 1: Illustration of run queue structures for both schedulers. The squares represent list heads and the circles represent tasks. The labels on the tasks indicate the static goodness of that particular task.

on the run queue but not running on a processor, its `counter` value does not change. Likewise, its `priority` almost never changes, though when it does, the ELSC scheduler adapts accordingly. We refer to the combination of these two values as a task's *static goodness*. A task's *dynamic goodness* consists of memory map and processor affinity. Despite the fact that they don't change while a task is on the run queue, they depend on which task and processor are calling `schedule()`. The ELSC scheduler uses static goodness to sort tasks on the run queue.

5.1 Implementation

The ELSC scheduler uses a new structure for the run queue. Previously, the run queue was a simple doubly linked list of nodes that each point to a task as shown in Figure 1a. To make scheduling decisions fast, we need to keep the run queue sorted, while at the same time keeping insertion and deletion times small. The ELSC scheduler does this with an array of 30 doubly linked lists. Each list in the array is used to hold tasks in a certain static goodness range, as demonstrated in Figure 1b. Lists at one end of the table hold tasks with the highest static goodness values while the other end hold tasks with the lowest. A `top` pointer is used to indicate the highest priority list that contains a runnable task.

To change the structure of the run queue from a single list to a table of lists, we need to change four run queue manipulation functions as well: `add_to_runqueue()`, `del_from_runqueue()`, `move_first_runqueue()` and `move_last_runqueue()`. The first of the two functions puts tasks on and removes them from the run queue when appropriate. The next two tasks give a task an advantage/disadvantage in the selection process when another task has the same `goodness()` value. Only `schedule()` manipulates the run queue directly.

The function `add_to_runqueue()` is modified slightly to deal with the new table structure. Like the current scheduler, it adds tasks to the front of a list. The particular list depends on the task. If the task is real-time, it uses one of the ten highest lists, determined by dividing the rt_priority field by 10. If the task is a `SCHED_OTHER` task, then the list is determined by adding `counter` to `priority` and dividing by four. Once the list is chosen, the task is added to the front of that list and the `top` pointer is updated if necessary.

When all tasks in the run queue exhaust their time quantum, their `counters` are all zero. At this time, the current scheduler resets all `counters` in the system. The ELSC scheduler does the same. However, to avoid re-indexing every task in the run queue when their `counter` is reset, we modified `add_to_runqueue()` as follows. If the task being inserted has a non-zero `counter` value, the task is inserted as described above. Otherwise, `add_to_runqueue()` uses a predicted `counter` value for the task, based on its knowledge of how the scheduler resets them. Using the predicted `counter` value and its current `priority`, the task is indexed into the run queue and added to the end of its list. This way, all zero `counter` tasks reside at the end of the list, behind all tasks with a non-zero `counter` value. The zero `counter` tasks are out of the way of the scheduler, but are in position once all other tasks in the run queue exhaust their quanta. A `next_top` pointer is used to keep track of the highest priority list containing a runnable task after `counters` are reset and is set at this time.

In the current scheduler, the `del_from_runqueue()` function removes a task from the list it is on by simply pointing the two nodes on either side of it in the list at each other. Then it sets its own run queue node's `next` pointer to NULL, indicating that it is no longer on the run queue. The ELSC scheduler follows exactly the same process. Afterwards, it updates both the top and `next_top` pointer if the removal of the task caused either one of them to change. In the ELSC scheduler, it is possible for a task to be considered on the run queue but not actually be in one of the lists in the table.[3] Because a node's `next` pointer indicates presence on the run queue by the current scheduler, the ELSC

[3] The reason for this is because we actually remove tasks from the run queue while they are running, but the rest of the Linux system would like to think that they are still on the run queue. This gives us a way to tell precisely if a task is on a list.

scheduler also sets the `prev` pointer to NULL to indicate that the task is not actually on any list, thus leaving the `next` pointer alone if the task is considered “on the run queue” without being on the run queue.

The functions `move_first_runqueue()` and `move_last_runqueue()` were meant to bias decisions in the case of a `goodness()` tie. Consequently, we need only to move tasks within their current lists in the table. A task is moved within its current list to the beginning or end of its section of the list. Recall that lists can contain tasks with both zero and non-zero `counter` values. These functions behave appropriately when faced with mixed-counter lists.

In addition to the modification of these four functions, code was added to initialize the run queue table structure when booting. We also wrote two test routines that determine whether a list contains tasks with zero or non-zero `counter` values.

5.2 ELSC Scheduling Algorithm

Like the current scheduler, the actual ELSC implementation of `schedule()` begins by executing all outstanding bottom-halves and then performing some additional administrative work. It then deviates from the current scheduler as follows.

If the previous task was still running when it called `schedule()`, i.e., it exhausted its quantum, was preempted, or yielded the processor, then the ELSC scheduler inserts the task into the run queue. This step is important because tasks are removed from their run queue lists when they are executing and need to be put back on the run queue. Even if the task has yielded, it will be treated properly in the search loop. So we insert the task in the table now lest we lose track of it. Also, by re-inserting the previous task here, we do not need to treat it as a special case when evaluating the goodness of tasks. Next, just as the current scheduler, ELSC moves exhausted SCHED_RR tasks to the ends of their lists.

The next step determines whether we need to recalculate `counters`. If the top pointer is zero, then there are no runnable tasks in the table with a non-zero `counter` value; either they all have zero `counter` values or there are no tasks in the run queue. If the `next_top` pointer is non-zero, then there are runnable tasks in the table with zero `counter` values, so the scheduler recalculates the `counter` values for every task in the system. If, however, the `next_top` pointer is zero, then the table is completely empty and there are no tasks to run, so we schedule the idle task and skip the rest of the decision process.

If the top_pointer is non-zero, the list pointed to by it is guaranteed to have at least one non-zero `counter` task in it, so we start our search at the top list. The ELSC search loop attempts to emulate the `goodness()` calculation used by the current scheduler. Starting with the first task in the list, ELSC checks to see if the task is still running on another CPU. If so, we shouldn’t schedule it. If all tasks in the list are eliminated by this check, then we consider the next populated list and try again.[4] Next, we check to see if the task has a zero `counter` value. If we find such a task, then the rest of the list is either empty or unusable, so we break out of the search loop. If, however, the task we are considering has a non-zero `counter` value, then we evaluate its goodness. The process of selecting a task from the highest list is described below.

If the task has just yielded its processor, we will run it only if we cannot find another task on the list. This policy is slightly different than the current scheduler, which considers a yielded task to have a goodness value of zero. From this point, the task’s utility is evaluated just like `goodness()`. Bonuses are given for having the same memory map or running on the same processor. In the uni-processor case, if a memory map match is found, then we break out of the search loop and run the task right away because we won’t find another task with a greater bonus.

When we finish examining a task, we mark it to be scheduled next if it has the highest utility seen so far. Then we select the next task in the list and repeat the process. In the worst case, every task in the run queue is placed in the same priority list (and ELSC performance can be no better than the current scheduler). So we limit the number of tasks examined in each list to a number, currently set to be half the number of processors in the system plus five, which is intended to be large enough to find tasks with adequate bonuses on SMP systems, yet still limit the search to a reasonable number of tasks. Not considering the rest of the list shouldn’t be a problem, as all tasks in the list have about the same static goodness.

For real-time tasks, the search is actually much simpler. Again, we examine only the first few tasks and don’t look at those currently running on other processors. But instead of worrying about yielded processes and bonuses, we simply run the task with the highest rt_priority value.

After deciding which task to run next, the ELSC scheduler manually removes the task from its list (i.e., doesn’t use `del_from_runqueue()`) and sets run queue node’s `prev` pointer to NULL. This indicates

[4] This can only happen on SMP systems.

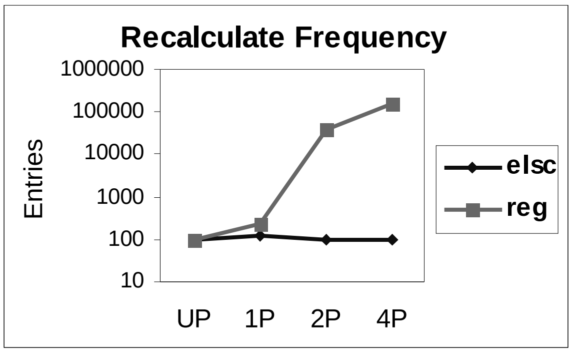


Figure 2: The number of times (on a log scale) that each scheduler enters the recalculate loop during a typical run of the VolanoMark benchmark.

that the task is "on the run queue", even though it is not currently in a list. Finally, if the previous task had yielded the processor, then the ELSC scheduler clears the `SCHED_YIELD` bit to give the task a better chance in future calls to `schedule()`.

We mentioned before that one of our design goals was to make the ELSC scheduler behave as much like the current scheduler as possible. At this point, we describe how the ELSC scheduler behaves differently. First, the ELSC scheduler tries to limit its search to one list in its table. Therefore, it may choose a task in its highest priority list that doesn't receive any bonuses for processor affinity or memory map. In this case, it is possible that a task residing in the second highest priority list, which would receive these bonuses and have had a higher `goodness()` value than the chosen task, is not run. We decided this behavioral difference is acceptable because the difference between the `goodness()` values of the two tasks is small enough to ignore.

The other difference in behavior is one that avoids an undesirable characteristic of the current scheduler. Currently, if a task enters the scheduler because it is yielding the processor and no other tasks can be scheduled, then the scheduler enters a loop to recalculate the `counter` value for all tasks in the system. In this situation, the ELSC scheduler runs the previous task again if it does not have a zero `counter` value. Figure 2 illustrates how many times each scheduler recalculates during a typical VolanoMark run on uni-processor and one, two and four processor SMP machines.

6. Experiments

The ELSC scheduler meets the first three of our four design goals. The design changes are kept local, the solution is simple, and it behaves very much like the current scheduler. The final goal of this project is to make the ELSC scheduler perform as well as the current scheduler in lighter desktop situations while scaling gracefully under heavy loads. We used two tests to determine whether we reached this goal. The first is a simple test that measures the time it takes to compile the Linux kernel. This test is meant to compare scheduler performance for light loads. The second test is the VolanoMark benchmark, described earlier. While VolanoMark may not be representative of a typical workload, it does simulate the behavior of a commercially available application. We use it in this analysis as a stress test for the two schedulers.

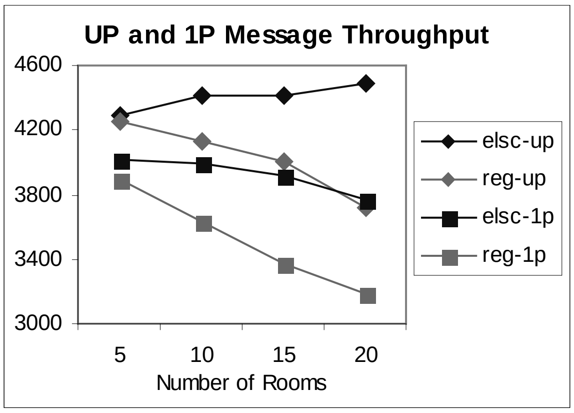
Scheduler	Time to Complete Compilation
Current - UP	6:41.41
ELSC - UP	6:38.68
Current - 2P	3:40.38
ELSC - 2P	3:40.36

Table 2: Average time taken to complete a full compile of the Linux kernel.

We compiled the Linux kernel three times on each of the schedulers, configured to run as uni-processor[5] and two-processor kernels. We ran the test on an IBM Netfinity 5500 with dual Pentium II processors. The kernel version was 2.3.99-pre4 with our ELSC modifications. To run the test, we set up a shell script that would first build a kernel and then run "make clean". This step was intended to reduce the variance in measurement due to file system performance by pulling as much information as possible into the L1 and L2 caches. Then we use the bash "time" command to run the "make -j4 bzImage" command. Table 2 shows the average results given by the time command.

Our confidence in these measurements is very high as the test was run multiple times and results never deviated from the mean by more than 4 hundredths of a second. For all practical purposes, the hundredths of a second reported in Table 2 are insignificant. In the two-processor case the ELSC scheduler barely edges the current scheduler by an insignificant couple hundredths of a second. In the uni-processor case the ELSC scheduler has a distinct advantage. We believe

[5] In these experiments, uni-processor kernels are compiled without SMP enabled, eliminating its overhead. One-processor kernels are compiled with SMP enabled but use only one processor.



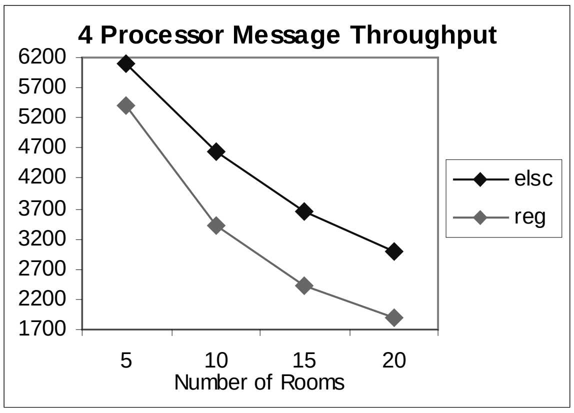


Figure 3: Throughput in messages per second for VolanoMark runs on 6 different scheduler configurations. The Y-scale is adjusted on the second graph to fit all data points.

this is due to the shortcut in the ELSC search loop for the uni-processor scheduler, which ends the search as soon as a memory map match is found.

The VolanoMark benchmark test is more complicated. We ran VolanoMark in loopback mode, which simulates both the clients and servers for the Java chat rooms on the same machine. In loopback mode, communication between clients and servers does not travel across a network. In the exchange of messages between clients and servers, each must have time on the CPU to send and receive it's messages in order to let the other do the same. This type of message exchanging application forces many entries into the scheduler. As suggested by the VolanoMark run rules, we ran the benchmark 11 times for each system configuration and discarded the first run due to its variant startup costs.

We ran VolanoMark with both schedulers configured as uni-processor, one, two and four processor SMP kernels. For each of these configurations, VolanoMark was configured to simulate 5, 10, 15 and 20 rooms, each with 20 simulated users exchanging 100 messages. Each simulated user creates two threads, so each room

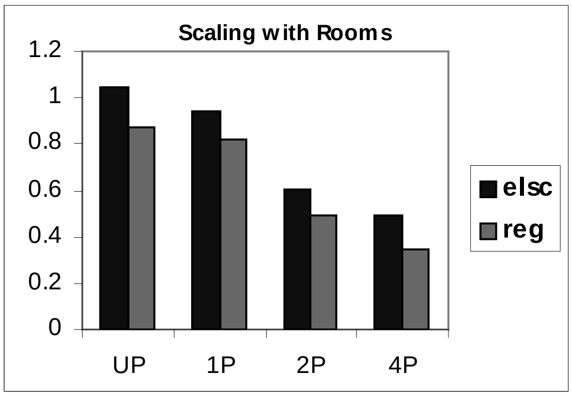


Figure 4: Shows how each scheduler scales from 5 rooms to 20 rooms on various processor configurations. The height of the bar represents the scaling factor (20-room-throughput / 5-room-throughput).

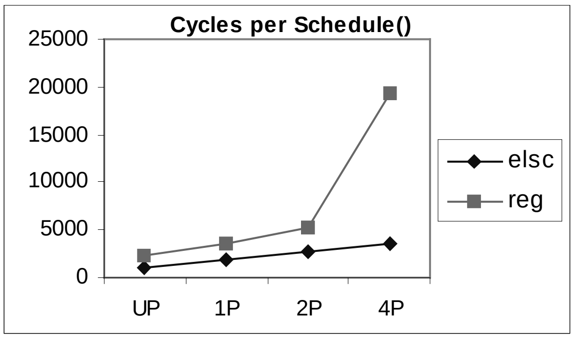
creates a total of 80 threads. It is easy to see that even at 5 rooms the VolanoMark benchmark puts considerable stress on a system. While running VolanoMark, we also collected statistics about what the scheduler was doing and exposed them through the proc file system. The overhead of collecting these statistics exists in both schedulers and in both cases is negligible. The machine used for the VolanoMark runs was an IBM Netfinity 7000 with 4 Pentium II xeon processors.

The metric reported by VolanoMark is message throughput, which we can use as a measure of both performance and scalability. Using the bare results from the VolanoMark runs, we can compare how each of the two schedulers behaves in different configurations. Figure 3 illustrates the performance gains given by the ELSC scheduler.

Figure 4 gives a different interpretation of the same data. To obtain some measure of how well each scheduler scales when faced with a large number of tasks, we can use the 5-room trials as a base measurement and see how performance is altered when the number of threads is increased in the 20-room trials. The number charted in Figure 4 is simply the message throughput achieved in the 20-room trials divided by the throughput achieved in the 5-room trials. As the figure indicates, the ELSC scheduler clearly scales to more threads better than the current scheduler.

But these numbers do not paint the whole picture. We want to understand why the ELSC scheduler scales so much better than the current scheduler and verify that these results are not a fluke. So we collected additional statistics on the schedulers while we ran the VolanoMark tests.

The first statistic that jumps out is the number of cycles spent per entry into the scheduler. For the ELSC



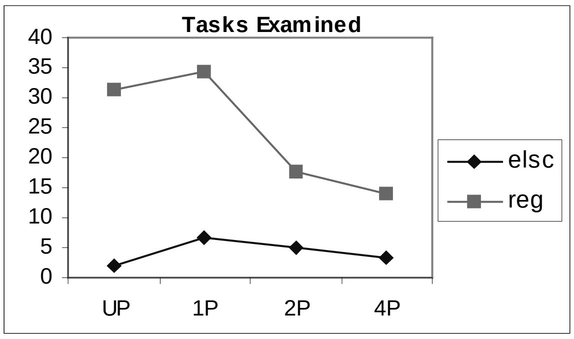
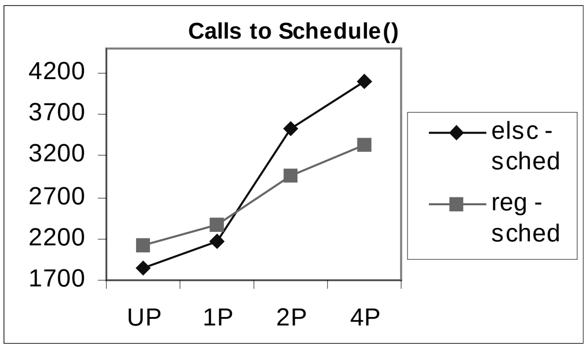


Figure 5: The first chart shows the number of cycles that are spent each time the system enters the scheduler. The second chart shows how many tasks are examined by the scheduler each time it is called.

scheduler, this number is significantly lower than the current scheduler, proving the ELSC scheduler really does spend less time in the scheduler. The explanation is that because ELSC with its table-based approach to scheduling examines far fewer tasks on each entry into the scheduler, as demonstrated by Figure 5.

The ELSC scheduler is not without fault. Although most of the statistics we collected indicate that the ELSC scheduler is faster and better, two of them show the opposite. One of the adverse affects of a table-based scheme is an increase in the number of calls to `schedule()` when running on a machine with more than one processor. As demonstrated by Figure 6, there is a strong correlation with how many times a task is selected without having the processor affinity bonus. These measurements suggest the ELSC scheduler is not choosing the absolute best task in multiprocessor machines. We suspect that this is related to the fact that the ELSC scheduler finds the most suitable task in the highest populated class of static priorities. Thus, some tasks that might have higher `goodness()` values when the processor affinity bonus is added, but reside in lower static classes, may not be considered.

Although the VolanoMark benchmark creates many threads with the same memory map, we do not believe this fact significantly influences the behavior of either scheduler. The only possible difference between the two schedulers would be similar to the passing over of



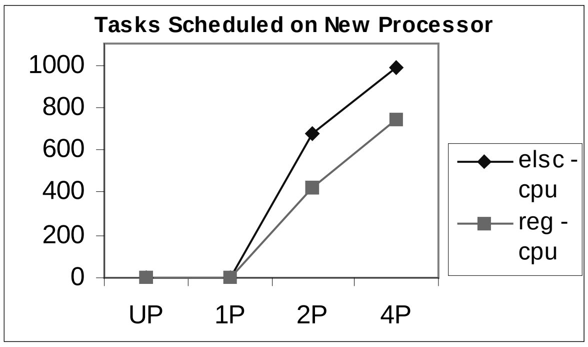


Figure 6: The first chart shows how many times (in thousands) the system enters the schedule() function call in an average 10-room VolanoMark simulation. The second chart shows how many times the scheduler chooses a task to run on a different processor than it ran before.

lower classes of static priorities that happens when running on multiple CPU's. Of course, if a task is inserted into a lower priority list, then adding the one point bonus for sharing a memory map with the previous task cannot raise it's goodness value enough to be greater than any of the tasks in the highest class.

7. Evaluation

An increasing number of organizations continue to evaluate, test, and use the Linux operating system. Although Linux does many things well, we have shown the current scheduler has shortcomings in its design and implementation. When confronted with a large number of tasks, overall system performance declines rapidly. This behavior is unacceptable for large-scale enterprise environments.

We set out to improve the Linux scheduler's scalability, preferring modifications that do not change desktop performance and maintain existing scheduler abstractions, yet scale well when presented with a large number of tasks. We have shown that it's possible to improve the Linux scheduler without introducing a lot of overhead. Though the ELSC scheduler does not always select the best task available on machines with more than one processor, we have demonstrated that the

ELSC scheduler satisfies our goals for both a small and large number of ready tasks and offers a viable alternative to the current Linux scheduler.

The ELSC scheduler is an open source contribution and is freely available for use and modification. The current version of the ELSC patch can be downloaded from `www.citi.umich.edu/projects/linux-scalability/patches/`

8. Future Work

In the future, we would like to see how the ELSC scheduler performs in other multithreaded environments. One such example is a web server running Apache. Would we see the same performance gains we saw while running VolanoMark, or does something other than the scheduler cause primary bottlenecks in these systems? Would the ELSC scheduler be more effective in increasing throughput or decreasing the latency of an Apache web server?

The focus of the ELSC design is to reduce the time spent looking for a task to schedule. We would also like to find ways to allow the scheduler to make greater use of multiple CPUs and examine the effects of modifying the goodness metric. Is Linux considering everything it ought in its scheduling decisions? Do we care about processor affinity after many other tasks have run on the given processor? Can we construct a scheduler that spends less time waiting for spin locks and more time scheduling tasks?

We are also interested in exploring alternative scheduler designs. The table-based design of the ELSC scheduler is one approach; many other possibilities exist, such as sorting tasks by static goodness within heaps for each processor and address space. One could choose the absolute best task available simply by examining the top of each heap. Or perhaps a multi-priority-queue solution would be more beneficial to help the scheduler scale to multiple processors well.

9. Acknowledgments

We thank Ray Bryant and Bill Hartner of IBM, Chuck Lever of Network Appliance and Brian Noble of the University of Michigan for their guidance and assistance; Dr. Charles Antonelli and Professor Gary Tyson for providing hardware used in development at the University of Michigan; the Linux Technology Center at IBM for allowing the use of equipment at IBM for development and testing; and Chris King, Scott Lathrop and Paul Moore of the University of Michigan for their contributions to the initial development of the ELSC scheduler.

We thank the anonymous reviewers for their helpful comments, and our shepherd, Ted Faber, whose insight and suggestions were particularly valuable.

This work was partially supported by Dell, Intel, and the Sun-Netscape Alliance.

We also thank all the past, present, and future developers of Linux for their skilled and selfless contributions.

10. References

[1] Atlas, A. "Design and implementation of statistical rate monotonic scheduling in KURT Linux." *Proceedings 20th IEEE Real-Time Systems Symposium*. Phoenix, AZ, December 1999.

[2] Bryant, Ray and Hartner, Bill. "Java, Threads, and Scheduling in Linux." IBM Linux Technology Center, IBM Software Group. **http://www-4.ibm.com/software/developer/ library/java2**

[3] Carr, John. "AS/400 Leads the league in Java performance." *JavaWorld*, 11 August 2000.

[4] Daggett, Dawn and Gillen, Al and Kusnetzky, Dan. "Linux Overtakes NetWare for the Market's Number 2 Position." *International Data corporation – Press Release*, 24 July 2000. **http://www.idc.com/software/press/PR/ SW072400PR.stm**

[5] Gooch, Richard. "Linux Scheduler Benchmark Results." 30 September 1998. **http://www. atnf.csiro.au/~rgooch/benchmarks/linux-scheduler.html ftp://ftp.atnf.csiro.au/pub/people/rgooch /linux/kernel-patches/v2.2/rtqueue-patch-current.gz**

[6] Lohr, Steve. "IBM goes counterculteral with Linux." *The New York Times On The Web*, 20 March 2000. **http://ww10.nytimes.com/ library/tech/00/03/biztech/articles/ 20soft.html**

[7] Neffenger, John. "The Volano Report." *Volano LLC*, 24 March 2000. **http://www.volano.com/report000324.html**

[8] Shankland, Stephen. "AOL releases open-source software." *Cnet News*, 9 July 1999. **http://www.canada.cnet.com/news/0-1005-200-344644.html**

[9] Wang, Y.C. "Implementing a general real-time scheduling framework in the RED-Linux real-time kernel." *Proceedings 20th IEEE Real-Time Systems Symposium.* Los Alamitos, CA, 1999. 246-55

[10] Woodard, B. "Building an enterprise printing system." *Proceedings of the Twelfth Systems Administration Conference (LISA XII).* USENIX Association, Berkeley, CA. 1998, 219-28.