RBS: Resource Based Scheduler

Joe Roback, Marshall McMullen, José W. Gifford
Department of Computer Science
University of Arizona
{robackja,mcmullen,gifford}@.cs.arizona.edu

Abstract

This paper describes the Resource Based Scheduler (RBS), a modification of the Linux 2.6 kernel scheduler. RBS provides an optimized scheduler which makes more intelligent scheduling decisions based on the current resource needs of the processes to be scheduled. We report on the design of RBS and the lessons we have learned by implementing a subset of its design.

1 Introduction

This paper describes Resource Based Scheduler (RBS), which provides a modified implementation of the vanilla Linux 2.6.13.4 scheduler in the pursuit of optimal scheduling based on current system resource usage for maximal throughput. RBS was designed with the intention of providing all of the benefits of the existing vanilla Linux 2.6 scheduler as well as increased system throughput through more intelligent scheduling decisions. The most salient and challenging aspects of this project were trying to preserve the O(1) runtime, starvation avoidance, process interactivity, and all other outstanding performance aspects of the vanilla Linux 2.6 kernel.

Resource scheduling in general remains an intensive research topic due to its many practical applications. Despite the active ongoing research in this area, early attempts to incorporate resource utilization in scheduling has met with continual problems and obstacles [17]. We seek to overcome these problems and obstacles through a unique use of resources in process scheduling.

Although RBS is a resource-based scheduler, it is radically different in form and purpose from traditional resource-based schedulers. This is attributable to how it makes use of resource utilization information in increase throughput rather than fairness while simultaneously guaranteeing minimal overhead in collection methodology. Traditional resource-based schedulers normally try to guarantee system fairness to specific system resources for all users or processes [4]. Instead, RBS seeks to monitor and track system resource requirements per-process and use that information to make more optimized scheduling decisions. Moreover, RBS is unique in that it has almost no overhead, maintains fairness, and maintains the constant O(1) scheduling of the vanilla Linux 2.6 scheduler. The main goal of RBS is to improve system throughput by minimizing resource contention and avoiding thrashing hardware resources. The secondary goal is to maintain the O(1) scheduling and fairness implicit in the the vanilla 2.6 scheduler.

2 Background

In the current Linux 2.6 kernel, the scheduler is an O(1) operation, which means scheduling decisions are made in constant time regardless of how many processes are running on the system [1]. This provides the noteworthy advantage of greatly increasing the scalability, making this scheduler superior to traditional schedulers which perform various linear scans of all runnable processes [5]. The methods traditional schedulers employ are inefficient as they do not scale well and deteriorate as additional processes are added which have to be scanned each time the scheduler is called. Since traditional schedulers have to perform more work on each iteration with the addition of more processes, they generally perofrm very poorly when compared to the 2.6 scheduler.

Much of our work relied upon a detailed understanding and utilization of the existing vanilla 2.6 scheduler. As a result, a significant portion of time was invested acquiring a detailed understanding of the scheduler and the means required to facilitate modification and optimization. To provide a more insightful understanding of our scheduler modifications and final implementation, we have included a very *basic* introduction to how the 2.6 scheduler works.

The basic structure in the Linux scheduler is a *runqueue*, which is a list of all of the runnable processes per processor. Each *runqueue* contains two priority arrays: active and expired. There is one-array element for each priority. Linux currently implements 40 priority levels to the user. (i.e. nice values from -20 to 19), which the scheduler scales to priorities

in the range of 0 to 139 based on a dynamic priority calculation that incorporates real-time status, starvation, interactiveness, and other essential metrics.

To pick processes in constant time, a bitmap is stored in each array. The 2.6 scheduler then uses a ***fast find first set*** algorithm, implemented in the function `sched_find_first_bit()`, to quickly scan the bitmap and find the first bit set in the bitmap – indicating which queue to use in the priority array. Once the queue index is known, the scheduler simply picks the task on the front of that queue. Processes are scheduled round-robin from within the active array. When a task uses all of its timeslice, it is moved to the expired array and its timeslice is recalculated. When all timeslices are exhausted, the active and expired array pointers are *simply* swapped, eliminating the need to scan a list of exhausted processes and recalculate their timeslices all at once.

To handle processes which are more interactive, the kernel can award up to 5 bonus points to the process's original priority. Likewise, a processor bound process could to charged up to 5 penalty points, thus lowering its priority and allowing interactive processes to run more frequently. The heuristic for detecting interactiveness of a process is determined by simply monitoring how often a process sleeps. If the process is spending a majority of its time sleeping, then it is I/O bound and considered interactive. Conversely, a process that spends most of its time runnable is processor bound. As an added bonus to helping interactive processes, the scheduler will insert a process back into the active array if it is sufficiently interactive and the expired array is not starving.

3 Motivation

As explained in the background section, the Linux 2.6 scheduler is extremely efficient and highly optimized in finding the first priority array with a runnable process. However, once the priority array has been chosen, it simply selects the top process off of that array's queue of processes. This round-robin selection process lacks an intelligent heuristic specifically formulated to select the *best* process within that queue to match current system utilization. This observation led us to hypothesize that a more intelligent heuristic in selecting processes from within the queue could yield more optimal scheduling decisions and improved throughput by attempting to better select processes based on system resource utilization. RBS focuses on optimal process scheduling for overall system throughput with the assumption that given the proliferation of personal computers, a majority of users have their own computer and should not be insulated from themselves.

4 Conditions For Success

The Linux Kernel has been in development for 12+ years and has been contributed to and developed by some of the most proficient and renown open source programmers worldwide. Moreover, the 2.6 scheduler represents a major rewrite of the scheduler with dramatic improvements and an O(1) runtime. This scheduler is highly aggressive and has undergone years of development and testing. Despite being in a state of perpetual development, the 2.6 scheduler is extremely efficient with constant runtime performance making it extremely scalable as the number of processes do not affect its selection performance. Since this scheduler has separate runqueues for each processor, it scales particularly well on SMP machines. Additionally, other research we discovered on alternative 2.6 schedulers showed little to no improvement to the standard scheduler [6].

These observations, in conjunction with our research led us to determine there is little room for improvement over the 2.6 scheduler – especially in single processor machines. As such, our goals for success were very modest in terms of improving throughput over the vanilla 2.6 scheduler. To give a better measure of our success, we felt it was necessary to compare our scheduler to another 2.6 resource-based scheduler seeking to improve throughput rather than resource fairness. Our primary goal was to attain speedup over a comparable 2.6 resource-based scheduler and to meet or exceed the throughput of the vanilla 2.6 scheduler. Our secondary goal was not only to increase system throughput, but to simultaneously preserve O(1) performance and fairness.

5 Design

One of the major stumbling blocks traditionally faced by resource-based schedulers is in the acquisition of resource utilization metrics for scheduling decisions. This has been an insurmountable obstacle that has hindered the realization of any serious, successful resource base scheduler , with our goals in mind, from coming to fruition. This is primarily attributable to scheduler designers often being forced to make various assumptions about process behaviours, and statically assigning values to parameters to make scheduling decisions [13].

One of our major design considerations was to address this limitation head-on in an attempt to overcome the static nature of many of these parameters by determining them dynamically. To achieve a dynamic resource-based scheduler with the goal of improving system throughput, we were forced to devise a mechanism for obtaining resource metrics at runtime. At the outset, we hoped with all of the ongoing research in adding system accountability into the Linux 2.6 kernel, that the metrics required would already be built into the Linux kernel. However, we found that it was very hard to obtain the information we needed. Unfortunately, the available metrics helpful towards our design were not normalized or standardized in any usable fashion. Thus, our primary focus of the system was to design a resource based scheduler with normalized resource metrics. Once these metrics were obtained and normalized, we essentially formed a decision tree to make

more intelligent scheduling decisions. Various resources can be measured with relatively low overhead, such as processor usage, virtual memory page faults and disk activity. One difficulty is normalizing the metrics into a common scheme in order to directly compare each metric to one another, the system total, or any combination thereof. While we were able to achieve this for processor usage, additional metrics remain for future work.

5.1 CPU Usage

For this metric, we monitored processor usage on a per-process basis to make more intelligent scheduling decisions based on what we considered the most important resource under contention. Our primary observation was that we could improving scheduling based on processor usage, thereby reducing cache misses and greatly improving processor throughput. Monitoring how active a process was previously, and using that metric to determine whether it is suitable to run with existing process on the priority array, is one of the major goals of RBS. While we had several iterations of this metric accumulation and implementation, ultimately we opted for a very simple and effective implementation. Currently we determine the last runtime of a process, using this value as a hint towards how processor intensive that process is and what its current and future source requirements are. Ideally, a weighted moving average of processor time would give a more accurate overview of a process, but this is left for future work.

5.2 Disk I/O

Currently, disk contention is a major bottleneck in computer systems. By scheduling processes in order to minimize disk contention, I/O delays can be reduced and greater throughput can be achieved. For this resource, we track the number of bytes read and written on a per-process level. We then collect smaller I/O operations into one larger operation. This idea can be equated to the the Log-Structured File System (LFS), but at a much finer scale as it aims to group sequential I/O together into one larger block transfer [14]. The basic foundation for this presently exists in the scheduler, as the scheduler tracks the number of bytes read and written on a per-process basis. Additionally, the scheduler also tracks the number of system calls made to read and write data to the disk. Using these metrics, more intelligent scheduling decisions can be made in aggregating transfers and performing more useful work while the large block of data transfers. Using a similar decaying average as discussed in section 5.1, the scheduler can more effectively determine the I/O history of a process and use it as an indication of its future requirements. *Using disk I/O metrics are left as future work.*

6 Implementation

We have implemented our resource-based scheduler changes in the Linux kernel, version 2.6.13.4 [15]. This implementation exists as an initial prototype and proof of concept. We attempt to illustrate that more intelligent scheduling decisions can be made by considering resource usage. For our implementation, we only implemented the CPU Usage metric, though it would be trivial to also implement the Disk I/O metric since the foundation is already laid out in the scheduler.

6.1 Scheduler

One of our most important goals was to avoid introducing any additional overhead in the scheduler to preserve its O(1) runtime and its very efficient and scalable operation. We did not want to introduce any complex accountability process or tracking system that we have observed in other papers that used performance metrics [18]. Nakajima and Pallipadi implemented a resource accounting system that utilized a complex user-mode micro-architectural scheduling assist daemon to monitor resource utilization of processes [18]. Our main goal was to avoid exactly this sort of implementation. Our desire was to integrate all resource accounting into the existing structures and flow of the 2.6 scheduler.

Our approach incorporated the `task_struct` inside the scheduler, which essentially tracks the state of a process from the perspective of the scheduler. It maintains various metrics and values which are periodically updated by the scheduler. Our approach was to simply modify the `task_struct` so that it had additional fields for us to track our metrics in order to make our scheduling decisions. These metrics formed the foundation of our heuristic to improve the scheduling selection process.

To implement this efficiently, we modified `sched.h` and `sched.c` so that the `task_struct` keeps track of the amount of processor time each process used in its last run. Since one of our primary goals was maintaining the overall O(1) scheduling, we avoided scanning all running processes, as this would introduce high overhead and make the scheduler O(n), and consequently not scalable.

RBS limits itself to the current queue it selected from the priority array's bitmap. It has a fixed upper bound (configured at compilation time) on the number of processes per queue that it will consider when making a scheduling decision. This bounded window keeps the runtime constant and the scheduler remains O(1). Because the number of processes we consider is fixed by the window size, the number of processes running on the system do not affect the runtime performance of our scheduler. This maintains the O(1) scheduling property we sought to maintain, and guarantees our scheduler remains highly scalable as is the current vanilla 2.6 scheduler. One interesting idea we hope to explore in future research is to *dynamically* adjust this window size based on system utilization

while still setting a fixed upper bound.

To fully implement resource accountability for processor usage, we needed to provide a mechanism to ensure each process' runtime metric is updated correctly while avoiding any complex iterations over all processes. To implement this, we added in this update when a process is running so that each process is updated appropriately. Specifically, each time a context switch is made, the new process' previous runtime counter is reset. Subsequently, in the `update_processor_clock()` function, the process' previous runtime is calculated and stored in that process `task_struct`. Our scheduler then makes uses of this metric in order to more intelligently pair complementary processes and system resources when selecting a new process to run.

The final implementation of this resource is utilized when the scheduler is deciding which process to run next. As explained previously, the vanilla 2.6 scheduler simply iterates over the currently selected priority array's queue of processes and selects them in a round-robin fashion. We greatly modify this selection process so that rather than simply selecting them in a round-robin fashion, we iterate over the ***bounded*** queue looking for a process with non-conflicting resource requirements. Essentially, we compare the previous running process to the process we are currently considering. If the previous process was mostly processor bound, RBS attempts to schedule the least processor bound process in that queue and vice-versa. The overall system design seeks to always select the process that is the most directly ***opposite*** in resource requirements to the most recently executed process. In this way, we are always selecting the process that requires resources that are **not** in use. Intuitively, this avoids hardware contention and greatly reduces hardware thrashing. The net result of this is an increase in throughput and overall system performance.

7 Evaluation

7.1 Hardware Test Setup

The testbed that we used in order to test our scheduler against the other schedulers was as follows:

- Pentium 4, 3 GHZ
- 2 GB Ram
- 300 GB SATA
- Fedora Core release 4 (Stentz) Operating System

One of the important design assumptions we make based on our hardware specifications was a system with a single-processor and a single hard drive. In actuality, our design should perform equally well on an SMP system, but we did not have the opportunity or need to test our design on an SMP system.

7.2 Kernels Tested

For all of the different schedulers we tested, we used the default configuration provided by Fedora.

- Vanilla Linux 2.6.13.4 Kernel.
- Staircase Linux 2.6.13 Kernel
  - The Staircase Scheduler is an alternative Linux 2.6.13 scheduler. We present a full discussion of it under the *Related Work* section.
  - Even though the staircase scheduler has been under development for some time, the staircase scheduler failed under several benchmarks.
  - When it did not fail, its performance was far from that of the vanilla Kernel. Perhaps this stems from the fact that as processes move down the priority queue, they are matched against the many lower priority processes that exist in a standard system. This scheduler uses a modified version of a multi-level feedback queue. We speculate it does not properly ensure fairness and thus suffers from a starvation problem.
- RBS Linux 2.6.13.4 Kernel

7.3 Benchmark Results

In every one of the benchmarks we ran, both the vanilla scheduler and our scheduler correctly completed without any segmentation faults or failures. As noted earlier, this was not the case for the Staircase scheduler. The Staircase scheduler failed on several tests, specifically on the gzip portion of SPECINT2000 and on the Kernel+Apache benchmark. Another important observation to keep in mind while looking at these results is that even a nominal speedup is drastically important since the scheduler is such a vital part of the kernel and is called so ***frequently***. Kernel developers have acknowledged that the two most critical parts of a kernel are generally the memory subsystem and the scheduler since they directly affect the design and implementation of nearly every other aspect of the kernel [12]. In the setup we used, we compiled all of the kernels with a frequency of $250HZ$. This resulted in a scheduler tick of $4ms$, or put another way, the scheduler was called every $4ms$. Because the scheduler is called so often, even a nominal speed up will have dramatic overall system performance implications since each time the scheduler is called, the speedup will be compounded, yielding an exponentially increasing system throughput. The affect of choosing a more optimal process to run greatly impacts overall system performance in a profound way the longer the scheduler runs.

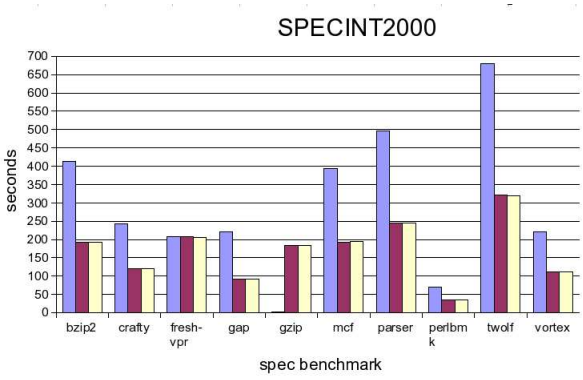

Figure 1: SPECINT2000

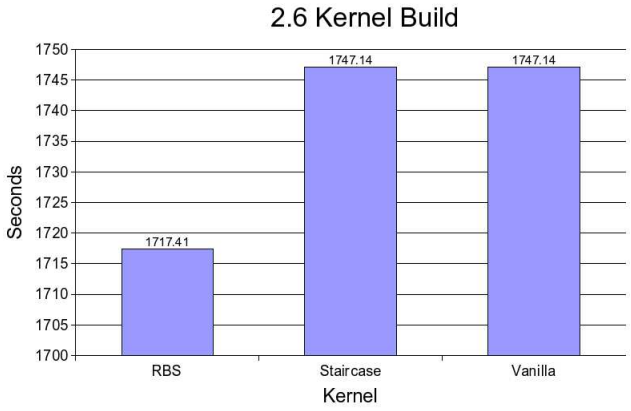

Figure 2: 2.6 Kernel Build

7.3.1 SPECINT2000

We used the following tests from the SPECINT2000 benchmark: *bzip2, crafty, fresh-vpr, gap, gzip, mcf, parser, perlbmk, twolf, vortex.* The SPECINT2000 benchmark is a very CPU-intensive benchmark that exclusively uses integer operations to stress test integer operations on the CPU. The aim of this benchmark is to simulate server applications which are traditionally very processor intensive [11]. As shown in figure 1, our results beat the Staircase scheduler in every test for SPEC2000. Moreover, we were as good, or better, than the vanilla kernel in every case. Specifically, RBS outperformed the vanilla kernel on *mcf*, whereas the vanilla kernel exceeded RBS on *twolf* and *fresh-vpr* by a minimal amount.

Another interesting observation we can glean from SPECINT2000 is the consistency of RBS. Staircase clearly has very poor consistency. This is seen in the dramatic range of runtime values for staircase on the various tests in SPECINT2000. Compared to Staircase, RBS is extremely consistent and shows comparable results on every run. Moreover, the performance and runtime of RBS is relatively comparable to the performance of the vanilla kernel on every test case whereas Staircase was significantly worse in every case. This benchmark is not particularly well suited to exhibit the improved performance of our scheduler since it involves only single processes. While it is true that there are always other processes running in the background, there is very little resource contention since the benchmark itself will receive almost all of the system resources and no other process will contend with it. With no concurrent processes competing for system resources, our scheduler simply falls back on the default scheduling decisions normally made in the vanilla scheduler. These results are confirmed when examining figure 1, wherein the performance of RBS is nearly identical to the vanilla scheduler in every case. One really encouraging observation, however, is that despite the overhead we introduce in our scheduler, it remains nearly identical in overall runtime to the vanilla kernel. We will examine this in more detail when we isolate and examine the overhead in section 7.4.

7.3.2 Linux 2.6 Kernel Build

In this benchmark, we measured the time to untar (bzip2 compression), configure, make, and remove the Linux 2.6.13.4 kernel. The tar file was created with bzip2 compression, making the extraction more processor intensive. The compilation of the kernel was run as a parallel make (*make -j2*) in order to force greater resource contention between the concurrent builds of the kernel source. This benchmark is perhaps the best benchmark to showcase the speedup we obtained with RBS. The reason this benchmark is so ideal in showcasing our speedup is because unlike SPECINT2000 which was a single process running with no resource contention, this benchmark involves multiple processes running simultaneously. Since we performed the compilation with the *-j2* option, two separate processes were run simultaneously to perform the kernel compilation. These parallel processes are both concurrently competing for the same system resources. Thus, our scheduler has the greatest opportunity for speedup by making better scheduling decisions based on the resource utilization of the processes it has to choose from. It will choose processes that are not competing for the same system resources so as to minimize resource contention and allow the two processes to both progress at faster rates and finish faster than if they had been scheduled normally as in the vanilla scheduler.

Our experimental results confirmed our expected results as this same result was depicted very clearly in this benchmark. As shown in figure 2, our RBS scheduler performed better than *both* the vanilla scheduler and the Staircase scheduler. Specifically, the kernel build with RBS took over 30 seconds less than both RBS the vanilla and Staircase schedulers. As depicted in figure 2 and summarized in table 2, RBS was 1.73% faster than the vanilla scheduler and 1.73% faster than the Staircase scheduler.

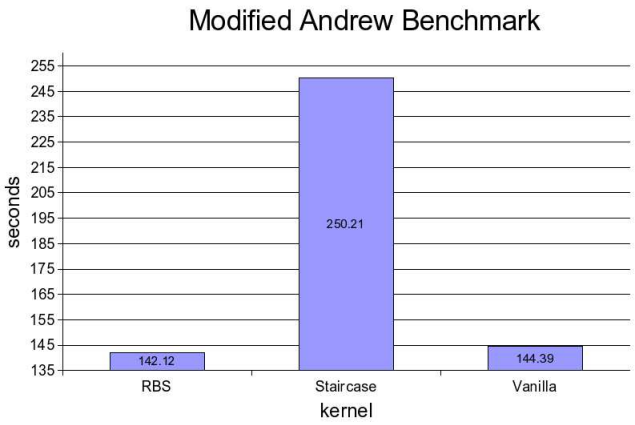

Figure 3: Modified Andrew Benchmark

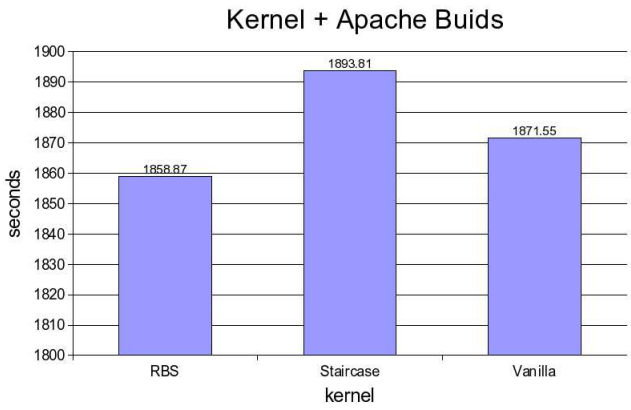

Figure 4: Kernel + Apache Builds

7.3.3 Modified Andrew Benchmark

In this benchmark, we measured the time to untar, configure, make, and remove the Apache 2.0.55 source tree. The interesting aspect of this benchmark is that it untars a very large source tree, and performs a very long compilation, followed by removing a very large number of files created during the extraction and compilation stage. Unlike the kernel build, this build was not performed in parallel. Thus, we would expect that the performance difference between RBS and the vanilla scheduler will not be as dramatic. As expected, these results are confirmed by figure 3. What is promising is that despite not being compiled in parallel, RBS still performed better than both the Staircase and vanilla schedulers. Specifically, as shown in figure 3 and summarized in table 2, ***RBS performed*** 1.60% ***faster than the vanilla scheduler and an astounding*** 76.06% ***faster than the Staircase scheduler!***

Another very important observation that can be gleaned from this benchmark concerns the ***fairness*** of the the different schedulers. Normally, all other things being equal, we would expect that the system time plus the user time would equal the real time. However, in this benchmark, our results (not shown) indicated that the gap between the $system + user$ times for the Staircase scheduler were significantly greater than for the other two schedulers. The gap between the RBS and vanilla schedulers were nearly identical. This implies that there is a greater degree of starvation in the Staircase scheduler than in the other two schedulers as processes were not given enough time or opportunity to run, forcing the throughput to be much longer for the Staircase scheduler. The greater the disparity between the $real$ time and the $system + user$ time, the less fairness in the scheduler. We observed that we excelled at maintainting the fariness of the vanilla scheduler which did a far better job preserving fairness than the Staircase scheduler.

7.3.4 Kernel + Apache Builds

In this benchmark we ran ran both the Kernel build benchmark and the Modified Andrew Benchmark (both described previously) simultaneously, with the results shown in figure 4. As our results show, summarized in table 2, ***RBS ran*** 0.68% ***faster than the vanilla scheduler, and*** 1.88% ***faster than the Staircase scheduler.*** Another important consideration is the correctness of our scheduler. On several runs the Stairstep scheduler segfaulted, just as it did on several runs of the gzip benchmark in SPECINT2000. In order to obtain results we could graph to show comparative results, we were forced to eliminate the results when Staircase segfaulted, running it many more times than the other schedulers to obtain valid results. This benchmark shows that with multiple processes running simultaneously with opposite and competing resource needs, our scheduler outperforms the vanilla scheduler. These results are encouraging since running both the kernel and the apache builds together produce even greater contention and strain on the scheduler than running them separately. This additional strain and contention is clearly indicated by the speedup in this benchmark being less than the speedup when the benchmarks are run separately.

7.4 Overhead

One of our stated objectives in this project was to improve throughput while minimizing overhead so as to avoid the traps of other resource-based schedulers. To test the overhead of our implementation, we created a modified version of our kernel which performs all of the same operations as RBS except for the actual process selection. Thus, it still performs all per-process resource accounting and updates of time usage. It even performs all of the iterations over the queue to select the best process to schedule. The only difference, is that it still uses the *top* process on the queue, just as the vanilla scheduler does. In this way, we isolate the overhead of our kernel, but avoid adding any of the *benefit* of our kernel – thus measur-

	Real	User	System
RBS Kernel	145.16	106.35	33.93
Vanilla Kernel	143.09	107.08	35.54

Table 1: Overhead

	Kernel + Apache	Kernel	MAB
% RBS faster than Vanilla	00.68	01.73	01.60
% RBS faster than Staircase	01.88	01.73	76.06
% Vanilla faster than Staircase	01.19	00.00	73.29

Table 2: Comparative Results

ing only the costs incurred in our implementation. We were very pleased by the results of our overhead analysis as our scheduler incurred only a very nominal overhead. We tested this overhead on the MAB benchmark, and present the results in table 1. Observe our overall system performance of the RBS scheduler was 145.16 seconds versus the vanilla scheduler with 143.09 seconds. This results in only a 1.45% overhead, which is extremely low and *exceeds* our stated goal.

7.5 Results Concluded

In table 2 we present a comparative summary of the percent speedup we obtain in RBS over the vanilla scheduler and the Staircase scheduler. Observe that in all cases our scheduler outperforms the other schedulers, with a minimum speedup of 0.68% and a maximum observed speedup of 1.88% over the vanilla scheduler. Even more profoundly, we observed a minimum speedup of 1.88% and a maximum speedup of 76.06% over the Staircase scheduler. As stated in section 4, we hoped to exceed a comparable resource-based scheduler and to meet or exceed the vanilla Linux scheduler.

As the results summarized in table 2 indicate, we have completely achieved our conditions for success. We ***dramatically*** exceeded the throughput of a comparable resource-based scheduler and also met or exceeded the throughput of the vanilla 2.6 kernel scheduler in every benchmark, excluding two tests within SPECINT2000, where the vanilla scheduler outperformed RBS by a nominal amount. As explained above, this is to be expected since SPECINT2000 is not a benchmark well suited for the conditions our scheduler exploits in order to improve throughput. However, it is nonetheless encouraging that RBS performed comparably to the vanilla scheduler, even in these tests.

Overall, we have definitively attained our stated goals in this project: exceeding the throughput of a comparable 2.6 resource-based scheduler and to meet or exceed the throughput of the vanilla 2.6 scheduler. Our additional goal was to maintain the O(1) performance and the fairness of the vanilla 2.6 scheduler. As explained in the implementation section, we maintain the O(1) performance of the vanilla scheduler by setting a fixed bound on the number of processes we scan in our selection process. Additionally, we demonstrated that our scheduler is as fair as the vanilla scheduler, and dramatically more fair than the Staircase scheduler, as shown and revealed in the Modified Andrew Benchmark (section 7.3.3). Overall, our results definitely support our original hypothesis that utilizing system resource usage in the scheduler can be used to make more intelligent scheduling decisions.

8 Related Work

Considering the large influence upon system behaviour, a significant amount of research has been done towards optimal scheduling algorithms to meet a variety of needs. The primary obstacle for any resource based scheduling has been resource accounting management. The need to balance accurate accounting with acceptable performance degradation has left resource based scheduling to the obscure reaches of fair proportional share schedulers. Thus, the overhead of the additional accounting can be consumed by the fact that fair scheduling is imposed upon the system.

Lottery scheduling by Walspurger and Weihl was one such implementation for the Mach 3.0 microkernel [16]. Users were allowed an arbitrary amount of system resources, and scheduling ensured that users did not take disproportionate resource usages compared to other users. Lottery scheduling does not strive to have optimal resource usage among processes, but rather fair usage. The primary form of attempting reduction of resource contention was the use of resource rights. By making Lottery tickets relative to resource usages, lightly contended resources are more likely to be scheduled, allowing for processes requiring lightly contended resource a better chance to be selected to run. Optimal throughput though, was not the intended goal of Lottery Scheduling. Petrou, Milford, and Gibson implemented a hybrid lottery scheduling algorithm for FreeBSD, with relatively minor impact upon performance with a majority of their benchmarks [4]. Unix, Linux and FreeBSD accounting has been gaining more importance, but primarily for post-mortem analysis. Programs such as `top` can analyze previous system usage to help administrators decide how to manage their systems.

The currently scheduler is O(1), with Igno Molnar and Con Kolivas still trying to change the scheduler to meet different scheduling needs. Igno Molnar has been working on resource accounting for real time applications [8, 9, 10]. Currently, only the superuser can set something to run in real-time status, giving the process potential to overtake the entire system. The rt-limit-patches implement RLIMIT_RT_CPU , which defines the maximum amount of processor time real-time processes may use if there is no idle time. The runtime overhead is minimal, but there is undetermined behaviour if real-time processes use too much processor time.

Con Kolivas has implemented more drastic changes to the current scheduler algorithm, instead proposing the new Staircase scheduler that is a modified multi-level feedback queue [6, 7]. He allows for only one priority array, and tries to schedule based on how hot/active a process is to maintain fair pro-

cessor distribution, with hopes of having optimal process coscheduling based on priorities of processes. Kolivas hopes to increase interactivity of the Linux scheduler while having some increase in system throughput.

A large area of interest for scheduling and system resources has been in Multiprocessor systems. The usage of resources by processes has become increasingly important due to wider decision ability. Multiprocessor machines with processes equally likely on each run to utilize any processor, show considerable system degradation. Mulvihill and Grobman show that by taking into account cache affinity, there were able to gain a 12.8% improvement on synthetic workloads with Linux 2.4 kernel [13]. The use of processor affinity with the Linux 2.6 kernel has shown improvements, but still needs more work. Unfortunately, the research focus has been on processors, not the entire system. While this is important, it does not address the needs of the optimal scheduling for each specific core and the overall system, whereas RBS attempts to consider the entire system at one glance.

9 Future Work

As proof of concept, this project only considers the simplified case of processor contention. Even with this simple metric, the decision for next process to be scheduled was only on the maximal difference among the selection window, where the selection is defined to be the amount of processes under consideration for selection to become the next running process. Thus, the case with all processes in the current priority being of comparable processor usage, there is the wasted effort on continuous scanning iterations. Perhaps also taking into account varying window size, whether statically or dynamically determined, would be beneficial to overall system throughput. This could also come into play when considering that the current Linux scheduler is optimized for the common case of only 1-2 runnable processes [1, 2].

The selection algorithm could be improved to consider such outlying cases. Any extension would also have to take into account other system resources. The typical resources would be hard drives, memory, network, and even multiple processors, with the latter actually being a rather minimal change to the current project. Any specific resource could also be accounted for. The primary problem again, is how to have proper accounting (or at least somewhat relative) accounting for each metric per process, without adversely affecting runtime. There is also the consideration of how to rate resources relative to each other. Initially, a static rating would be easiest to extend for our current status, but in the future a more dynamic, or fluid solution would be ideal. If underutilized resources could be made more available to processes, the ideal throughput could be more likely achieved.

Processes use multiple resources and are apt to change their resource utilization patterns. It would be preferable to maintain a usage history without requiring too much additional overhead. The most important history would be the most recent activity. Thus, an exponentially weighted moving average could be used as it would have minimal additional overhead, giving the most weight to the current value with only one value to keep track of. The amount of additional metric information would depend on memory limitations. The probability of the current process descriptor being in the cache line is significantly higher, assuming the descriptor can fit into cache.

10 Conclusion

We have proposed Resource Based Scheduler, a modified 2.6 Linux scheduler in pursuit of optimal scheduling for overall system throughput. RBS utilizes system resource in order to make more intelligent scheduling decisions. We sought to overcome the limitations and poor performance of traditional resource-based schedulers which sought to guarantee user and process insulation. Instead of using resource utilization in an attempt to guarantee system fairness and provide user and process isolation, we instead use resource usage to make better scheduling decisions and thereby improve throughput.

One of the major lessons we learned from this project is that kernel development – particularly scheduler development – is a very complex task. The ongoing work to add additional process accounting into the Linux kernel has made great strides, however, there is still significant room for research in this area, specifically in making use of process accountability. Our project shows a real and useful application of making effective use of resource metrics within the scheduler to improve system throughput while simultaneously preserving all of the salient aspects of the vanilla 2.6 scheduler.

The first goal for success was to exceed a comparable resource-based scheduler and meet or exceed the vanilla 2.6 scheduler. Our Resource Based Scheduler ***dramatically*** exceeded the throughput of a comparable resource-based scheduler and also met or exceeded the throughput of the vanilla 2.6 kernel scheduler in every benchmark. Our second goal was to maintain constant runtime within the scheduler and preserve scheduling fairness. Our results and analysis illustrate that we met this goal as well. Since we met both of our stated goals, we consider this project a resounding success.

References

[1] Love, R. *Linux Kernel Development.* Sams Publishing. Indianapolis, Indiana, 2004.

[2] Love, R. *Linux Kernel Development, 2nd Edition.* Sams Publishing. Indianapolis, Indiana, 2005.

[3] Torvalds, L., et al. *The Linux Operating System.* See `http://www.linux.org` for more information.

[4] Petrou, D., Milford, J., Gibson, A. Implementing Lottery Scheduling: Matching the Specializations in Traditional Schedulers. In *Proceedings of USENIX 99, Monterey CA,* June 9-11, 1999.

[5] http://www.faqs.org/docs/kernel_2_4/lki-2.html

[6] Kolivas, C. `http://ck.kolivas.org/patches/2.6/2.6.4/experimental/staircase/`

[7] Kolivas, C. `http://kerneltrap.org/blog/con`

[8] Molnar, I. Real-Time Linux Kernel Patch. `http://people.redhat.com/mingo/rt-limit-patches`

[9] `http://lwn.net/Articles/106010/`

[10] `http://people.redhat.com/~mingo/`

[11] `http://www.spec.org/osg/cpu2000/CINT2000/`

[12] Hegde, V. The Linux Scheduler. *The Linux Gazette*. Issue 89, April 2003.

[13] Mulvihill, D., Grobman, I. Multiprocessor Scheduling Using Dynamic Performance Measurements and Analysis. Available at `http://www.cs.wisc.edu/~mulvihil/736_paper.pdf`

[14] Rosenblum, M., Ousterhaut, J. The design and implementation of a logged-structured file system. In *AMC Transactions On Computer Systems,* 10 (1) : 26-52, 1992.

[15] `http://www.kernel.org`

[16] Waldspurger, C., Weihl, E. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the First Symposium on Operating System Design and Implementation*. November 1994.

[17] Chan, W., Chun, D., and Kannan, G. Construction Resource Scheduling With Genetic Algorithms. In *Journal of Construction Engineering and Management*. June 1996.

[18] Nakajima, J., and Pallipadi, V. Enhancements for Hyper-Threading Technology in the Operating System – Seeking the Optimal Scheduling. In *Proceedings of the 2nd Workshop on Industrial Experiences with Systems Software,* USENIX, 2002.