Exploiting Malleable Parallelism on Multicore Systems

Daniel James McFarland

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

June 16, 2011
Blacksburg, Virginia

Keywords: Malleable, Resizable, Rigid, Thread

Exploiting Malleable Parallelism on Multicore Systems

Daniel James McFarland

(ABSTRACT)

As shared memory platforms continue to grow in core counts, the need for context-aware scheduling continues to grow. Context-aware scheduling takes into account characteristics of the system and application performance when making decisions. Traditionally applications run on a static thread count that is set at compile-time or at job start time, without taking into account the dynamic context of the system, other running applications, and potentially the performance of the application itself. However, many shared memory applications can easily be converted to malleable applications, that is, applications that can run with an arbitrary number of threads and can change thread counts during execution. Many new and intelligent scheduling decisions can be made when applications become context-aware, including expanding to fill an empty system or shrinking to accommodate a more parallelizable job. This thesis describes a prototype system called Resizing for Shared Memory (RSM), which supports OpenMP applications on shared memory platforms. RSM includes a main daemon that records performance information and makes resizing decisions as well as a communication library to allow applications to contact the RSM daemon and enact resizing decisions. Experimental results show that RSM can improve turn-around time and system utilization even using very simple heuristics and resizing policies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In the last few years, parallel computing has become much more mainstream. Nearly all new computers and even some handheld devices come with multiple processors. While we are currently in the multicore era of computing, we will soon shift to many-core systems with hundreds of processors on a single chip. Harnessing this computational power can lead to massive speedups in performance for applications.

Research and development on parallelism for shared memory systems has been extensive. However, most previous work assumes that the thread count for a given process is static and context unaware. By "static," we mean that the thread count for a particular job[1] or parallel section of a job is set once and for all, perhaps when the program is written, or perhaps at runtime before the first parallel section is encountered, e.g. in a fork-join programming model. However, the choice of number of threads is not revisited during execution to take into account the performance of the particular job and the performance of other jobs on the machine including their thread counts.

While some parallel code sections are limited to a fixed degree of parallelism, many applications do not need to have a particular thread count for their execution. For example, many scientific applications and everyday applications like video rendering have very high degrees of parallelism. They simply perform a certain amount of work within a loop before getting to a barrier. These applications do not need to be run with a specific number of threads. Jobs that can run on any arbitrary number of threads are considered "malleable" or "resizable"[2][15]. These malleable jobs are contrasted with "rigid" or "nonresizable"[3] jobs, which must run using one particular thread count that cannot be changed [15].

[1]A job refers to a particular instance of an application running on a particular set of data.
[2]We will use the terms "malleable" and "resizable" interchangeably for the duration of this thesis.
[3]We will also use the terms "rigid" and "nonresizable" interchangeably for the duration of this thesis.

There are several scenarios where context-aware malleability could improve performance for both the overall system and for individual applications. Suppose a number of jobs are running on a system and all but one finish. This single job now could potentially use the entire system since no other jobs are running. Using the entire system even if the job does not parallelize well, could potentially lead to a drastic reduction in the time required to complete its calculations. However, since it cannot take advantage of the available cores, it makes no difference if the machine is busy or empty (assuming that it is below capacity). In this case, expanding the job to use more cores can lead to performance gains.

Take another similar scenario as an example. Suppose a single job has filled the machine to full capacity. Then suppose a simple rigid job arrives. It uses a single thread but puts the machine above capacity. This oversubscription will cause both jobs to take longer since they compete for time slices. Barriers in the large job would cause all threads to be frequently delayed, and not getting time slices in the small job would cause it to run longer as well. In this case, if the large job could simply run at one thread smaller than the machine capacity, it could potentially run much faster since all threads would not be forced to wait for a single thread that could not run due to its time slice being used by the small job. Additionally, this would always allow the small job to run without having to wait for a core on the machine. In this case, contracting a job can potentially lead to performance gains due to oversubscription.

For a final example, suppose there are two jobs that are both using half of the cores on a machine. Suppose one of these jobs is embarrassingly parallel while the other can barely take advantage of multiple threads at all. If the embarrassingly parallel job could use a larger portion of the machine, it could potentially finish much sooner without hurting the other job too much. This would lead to a significant increase in performance for the embarrassingly parallel job, but with a small decrease in performance for the other job. However, the sooner that the embarrassingly parallel job finishes, the sooner the other job could potentially have the whole machine to itself. While the other job may be hurt, a system supporting context-aware malleability would help the embarrassingly parallel job far more than it would hurt the other job.

1.2 Research Goals

The main goal of this work is to explore and understand the potential of context-aware malleability on multicore systems. Specifically, we wish to answer the following questions with our research:

- What assumptions must be true about a system that implements context-aware malleability, the resizable job, and the job mix in order for resizing to be beneficial?
- How useful is it to take the performance of resizable jobs and the job mix as a whole into account when resizing?

- What issues affect the potential performance benefits of resizing? How sensitive are the performance benefits of resizing to these issues?
- What extra overhead is introduced by resizing in this context? In what situations does overhead outweigh its benefits?
- How is resizing on a shared memory system different from resizing on a distributed memory system?

1.3 Approach

In order to answer these research questions, we implemented a prototype system, RSM (Resizing for Shared Memory), which includes a simple programming API and runtime system, and which allows experimentation. RSM has several notable characteristics that allow for easy adoption as well as intuitive implementation. The system is portable and requires no kernel modifications. This allows it to run with user privileges. RSM is also lightweight. If it used significant system resources, it would drastically reduce the parallel benefits that user jobs can realize. RSM also needs to work with minimal modifications to common shared-memory parallel programming models, e.g., RSM should not cause programmers to think differently about how to write their code just because it is included. One particular model that fits the types of applications that we are targeting is OpenMP [26]. RSM also allows for easy experimentation with different thread count adjustment strategies. RSM needs to follow all of these characteristics in order to help us realize our research goals.

There are also several characteristics that our prototype system must avoid in order to help us reach our research goals. First, it must not require all jobs to be malleable. It needs to be aware of nonresizable jobs as well as resizable jobs. Secondly, RSM must not assume that jobs are homogeneous. Previous work on resizability for distributed memory systems assumed that jobs were homogeneous in that they did the same amount of work between points at which they checked-in with the resizing system [37]. Including heterogeneous jobs allows us to target a much larger audience of potentially malleable jobs. Finally, our prototype must not require a job-management or queuing system. When jobs arrive at the system, they should immediately run, just as they would in a typical shared memory environment. These restrictions will allow us to fulfill our research goals more completely.

While building a prototype that fulfills our research goals is the main approach that we take, we also need a flexible experimental framework to allow us to perform the exploration necessary to provide answers to our research questions. We use synthetic parallel jobs to allow fine-tuning of specific properties of those jobs in intuitive ways. Some properties of those jobs should include the ability to provide a burst of computation in the form of a single job with many threads or multiple jobs with many threads arriving at nearly the same time. This "burstiness" will allow us to test how well our prototype reacts to incoming jobs. The effect of bursts has been shown to be important [25] especially in massive storage systems [21], but since cores in a shared memory system are also a shared resource amongst jobs, it is important to account for bursts here too. Additionally, these jobs should have varying degrees of parallel performance both between jobs and within a job. We need a well defined measure of how well a job parallelizes in order to determine how well our prototype measures and reacts to performance. Finally, our experimental framework must allow us to produce both malleable and rigid jobs. Without this key characteristic of the experimental framework, it would be impossible to fulfill our research goals and our approach in building RSM.

1.4 Outline

The remainder of this thesis is organized as follows. In Chapter 2, we describe related work to show how our work relates to and builds on what others have done. In Chapter 3, we detail the design and implementation of RSM. In Chapter 4, we provide experimental setups, results, and discussion that show how well our implementation performs against simple static scheduling. In Chapter 5, we conclude with closing remarks as well as topics for future research.

Chapter 2

Related Work

Much work has been done on scheduling and malleability in parallel environments. We begin broadly and focus towards work that is most similar to this work. We wrap up this section with a comparison between previous work on ReSHAPE [37] [39] and this work.

2.1 Dynamic Scheduling in Parallel Environments

Parallel scheduling has received numerous studies and evaluations. Feitelson et al. provide a good general overview of recent work on this topic [15] [14]. In distributed environments, gang scheduling [27] and backfilling [23] are classic scheduling methods in parallel environments. Additional work has looked at choosing processor sizes for "moldable" jobs [8] [35]. "Moldable" jobs are considered jobs which can change thread counts before runtime but remain static during execution. Cirne and Berman also investigated models for converting rigid jobs into moldable jobs [7]. They also claim that most parallel jobs are already moldable. This observation improves the basis of this work since moldable applications can easily become malleable through the RSM API described in Section 3.2.3.

2.2 Scheduling on Shared Memory Multiprocessors

There are two main methods of scheduling that are similar to RSM in research literature. The first is signature analysis. In signature analysis, a scheduler determines an optimal schedule based on information that a process provides to the scheduler. This can be done through related thread IDs [29], gang scheduling of divide and conquer algorithms [3], modular scheduling frameworks [16], or application binary analysis [31]. Gathering data beforehand can lead to a well performing static schedule but introduces a lot of overhead for the programmer. The second main method of scheduling on shared memory platforms is cache awareness. Schedulers use different methods to determine how best to co-locate threads on a chip. Fedorova et al. claim that L2 cache contention “has the greatest effect on system performance” [10]. Examples of these methods include co-locating threads that require a lot of communication between each other [17], that access the same data [40] [34] [24], that have high cache usage with those that have low cache usage [1] [5], that are performance independent of co-located threads [12], and by extending the quantum of a process that has less cache contention due its co-located threads [20] [11]. These methods try to improve process performance by lessening the strain on that particular chip’s cache through intelligent pairings of threads with cores. Snavely et al. describe a method for determining how best to co-schedule jobs with dynamically adjusting priorities [32] [33] similar to work on threads that access the same data. While not directly related to this work, another area of research on this topic includes topology aware scheduling [22] [13] [2]. Work in this area tries to make the scheduler aware of differences between cores allowing smarter scheduling decisions to be made. These are all major examples of thread scheduling in shared memory systems. However, the work in this section focuses only on static scheduling and does not consider malleability.

2.3 Malleability in Parallel Environments

The malleability of applications has also received some attention. Malleable and rigid jobs are contrasted in [15]. Malleable jobs are those jobs which can change thread counts dynamically during execution. Introductory work describing malleable jobs on distributed memory systems claims that, in UMA systems, the performance benefits outweigh the overhead of resizing malleable jobs [30]. This has led to a thorough exploration of those malleable jobs on those systems [19] [4] [6] including hierarchical supercomputers [9]. One particular work is quite similar to RSM: Hungershöfer et al. describe malleability in a shared memory environment in [18]. Jobs change thread counts during runtime based on messages received from a central server. However, only the context of the machine is considered and individual job performance is ignored. One particular resizing policy takes into account formulas that describe the possible speedup of each job based on precalculated constants, but the actual validity of those formulas is not matched to actual performance at runtime. However, the most similar work to RSM is the distributed memory version it is based upon: ReSHAPE by Sudarsan et al. [37] [39].

2.4 ReSHAPE on a Distributed Memory Platform

2.4.1 ReSHAPE

Sudarsan and Ribbens describe the ReSHAPE Architecture in [37] and [39]. They also describe the resizing library, API, and redistribution algorithms they use in [36]. Further scheduling strategies and scenarios are described in [38].

ReSHAPE is a framework that contains a scheduler which supports malleable MPI applications on a distributed memory system. The scheduler can grow or shrink a job based on performance history or needs of the system as described by one of the numerous scheduling policies that it can employ. Since ReSHAPE is built for distributed memory systems, the framework includes a library that supports resizing functions and redistribution of data amongst nodes. Among the resizing functions are several data structures that facilitate redistribution of data. It also contains an API for interaction between applications and the scheduler. The framework can improve overall system throughput as well as individual job turn around time.

2.4.2 RSM and ReSHAPE

While we know how ReSHAPE performs on a distributed memory platform, a shared memory platform provides new benefits as well as challenges. Many things become easier but others become much more difficult. There are older problems that simply fade away and new problems that are very difficult to manage.

There are many similarities between distributed ReSHAPE and shared RSM. For instance, the main goal is still the same: increase performance of applications by controlling the number of cores they use. The method of controlling the system is still the same. Likewise, RSM should still have the ability to reward those programs that use the system efficiently through useful work. A final similarity is the need to handle both resizable and nonresizable jobs. ReSHAPE and RSM will not work for all jobs. Therefore, the system must recognize those jobs that are not resizable and factor them into the calculations of those that are.

While there are still many similarities between the two systems, there are many more differences. The most obvious difference is that there is no longer a need to move data between computing nodes. Main memory is accessible by all cores in a shared memory system. This removes much of the overhead from the distributed ReSHAPE. It also removes the requirement of certain types of malleable data structures from the distributed ReSHAPE. Another difference is that we can no longer assume that jobs will run for days or even hours. In a shared memory environment, there may only be minutes or seconds in which to evaluate a program and determine its impact on the overall system. Likewise, there is no concept of a "queued job" in a shared memory environment. Applications simply appear when they start and disappear when they end. A shared RSM must handle "burstiness," the idea that many threads can be started at the same time with no warning. Similarly, a shared ReSHAPE must maintain a responsive system. Users are much more likely to be closer to a shared memory system, where they expect immediate results as opposed to a distributed memory supercomputer where they will retrieve their results in several days. Before, in a distributed ReSHAPE, there was no concept of oversubscription since jobs that would put the machine above capacity simply waited in a queue until there was room on the machine. Shared RSM must be aware of oversubscription in order to keep the machine responsive. These differences point to a new approach that is needed in order to fully take advantage of shared memory systems. Simply porting ReSHAPE to RSM is not sufficient.

Chapter 3

RSM Design and Implementation

In this chapter we provide an overview of the design of RSM followed by implementation details.

3.1 Design

In Section 1.3, we described characteristics of the prototype of RSM. In this section, we describe the high level details of the system that would be helpful to someone implementing our framework. A diagram showing the major functional components of RSM is shown in Figure 3.1.

We use a main RSM Daemon to make resizing decisions. This daemon is always on and runs side-by-side with user jobs. This allows us to avoid kernel modifications and run with user level privileges. The daemon waits for information from a user job. When a RSM-enabled job J checks-in and sends this information to the daemon, the daemon records this information in the Performance Storage module so that it can compare job J with other malleable jobs that it knows about. The Resizing Analyzer uses performance information from job J and from other jobs as input to two resizing policies detailed in Section 3.2.1 (fulfilling another necessary characteristic of our prototype) in order to determine what thread count job J should use. After making this resizing decision, the daemon sends this information back to job J and waits for another job to check-in.

User jobs that wish to communicate with the RSM Daemon do so through the RSM Interface. When user job J reaches a barrier and wants to check-in with the RSM Daemon, one thread of job J simply calls a method in the RSM Interface. This method automatically collects all the necessary information, sends that information to the RSM Daemon, and then waits for a resizing decision response. After the resizing decision is received, the RSM Interface contacts a Job Resizer that takes advantage of the OpenMP programming model to adjust

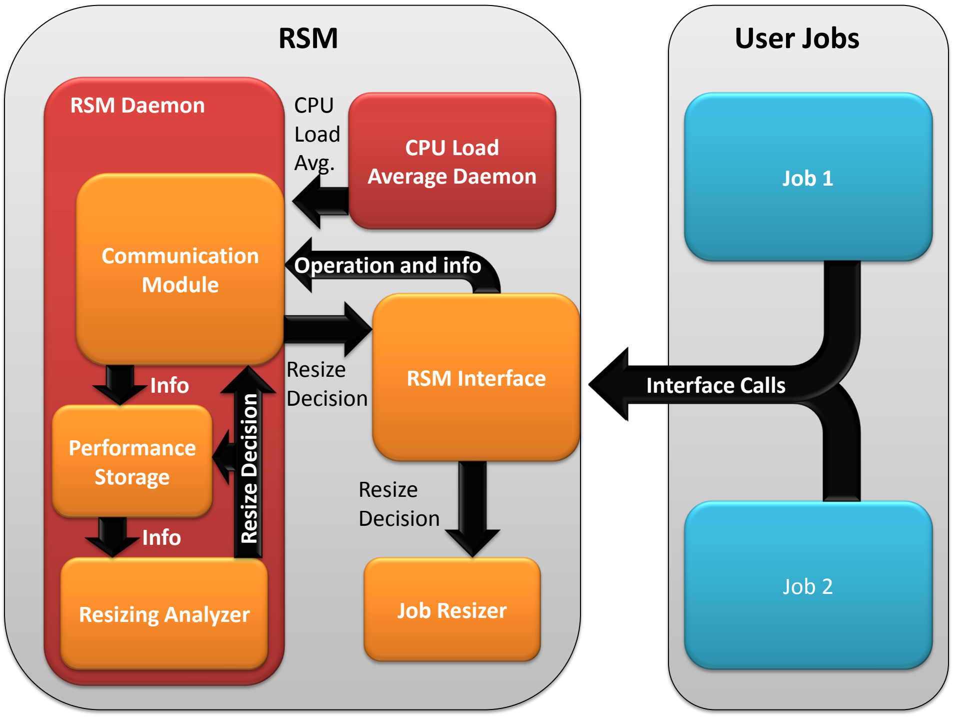

Figure 3.1: Diagram of communication between components of RSM.

job J's thread count. This fulfills the minimal modifications characteristic of our prototype. After job J's thread count is adjusted, the RSM Interface returns control of execution to job J to continue doing useful work.

The approach so far allows us to be fully aware of all malleable jobs on the system, as long as they check-in with RSM periodically. However, our prototype needs to be aware of jobs running on the system that do not check-in as well. When the RSM Daemon is making resizing decisions, it requires a good estimate of the total current CPU load on the system. All jobs running on the system contribute to the CPU load. We retrieve this number through a separate CPU Load Average Daemon. This CPU Load Average Daemon regularly pushes its estimate of the CPU load average to the RSM Daemon waiting for a response. The RSM Daemon simply records this information for use during resizing decisions and continues waiting for User Jobs to check-in.

3.2 Implementation

In this section, we describe the low level details specific to our prototype implementation. All code is written in C using TCP sockets to facilitate communication between processes.

3.2.1 RSM Daemon

The main daemon is an always-on RSM Daemon. This daemon receives all the information from resizable applications: their initializations, check-ins, and exits. It also keeps track of information that it uses to make resizing decisions. It keeps track of time since a job's last check-in and the number of threads that it used. It also records the instructions retired by the application by using Perfmon2[28]. The RSM Daemon also uses the CPU load received from the CPU Load Average Daemon to discover nonresizable jobs. Upon receiving a check-in, the RSM Daemon is provided all information necessary to make a resizing decision, and then decides what new thread count the checking-in job should run at given the context of the checking-in job and the context of all other jobs on the system.

Resizing Policies

The RSM daemon currently uses one of two resizing policies: Greedy or Global. When a job checks in, the Greedy policy looks only at the current state and capacity of the system, i.e., the current estimate of the load average and the total number of cores on the system. This policy almost exclusively uses the CPU load to determine whether a job should increase its thread count if there is room on the system, maintain the same thread count if the machine is full, or decrease its thread count if there are too many threads running. In the Global policy, the RSM Daemon ranks all currently running malleable applications based on useful work done given the amount of resources used. Currently, this is implemented using the following equation:

$$UsefulWork = \frac{InstructionsRetired}{Threads \times Time},$$

where the data is collected over the interval since the job last checked-in. Since we are focusing on CPU-bound applications, this measure quantifies the fuzzy concept of useful work and can reward those programs with good load balancing and few waits. In other words, it rewards those that use the system efficiently for useful work. After ranking the running applications, if the application checking-in is in the top half of most efficient jobs that have checked-in with RSM, it can increase its thread count (if there is room to expand). If it is in the bottom half, it will decrease its thread count. These percentages can be adjusted to allow a range in the middle where no resizing occurs, but we did not use this in our experiments.

Additional heuristics are used when determining by precisely how much a job should expand or contract. When expanding, a job will try to expand by 50% of its current thread count. If expanding by this number would put the machine above capacity, then the job will only expand to put the machine at capacity. When contracting, a job will contract by 25% of its current thread count. All changes in thread counts must be integer changes since a job cannot run on a fraction of a thread. This means that a job cannot contract below 3 threads. In order to keep the machine as close to capacity as possible, we use one final general heuristic. If job J is contracting by a large number (greater than 4 threads) but the machine is less than 4 threads over capacity, then job J only contracts by 3 threads instead of the larger number. In this special case, the machine is just above capacity, but the simplistic contraction heuristic is probably overcompensating with a large contraction. This heuristic leads to a much smaller reaction, which keeps the machine closer to capacity.

Measuring Useful Work

We measure the instructions retired by a job using Perfmon2. Perfmon2 uses a system call available in Linux 2.6.32 and above to query registers on each CPU. It maintains this statistic for all threads of a process on all CPUs without additional work by the programmer. While other performance counters allow querying the processor registers directly, Perfmon2 is sufficient for RSM since jobs are not checking in often enough to see a visible penalty for the system call overhead. The rate of instructions retired per thread is a simple measure of how well a parallel job is performing in the recent past, i.e. it does not assume that the job has behaved a certain way since it was launched. However, more sophisticated measures of performance are possible. For example, more advanced versions of RSM could take I/O activity by a process into account since these types of applications do much useful work with few instructions retired.

3.2.2 CPU Load Average Daemon

The second daemon that RSM uses is a CPU Load Average Daemon. We implemented a separate daemon that regularly queries the number of running threads provided in the /proc/loadavg file in Linux. This query is done ten times per second, after which the daemon sends an average over the last second to the RSM Daemon. The average sent to the RSM Daemon is sent only once per second. Both of these parameters can be adjusted by arguments sent to the daemon. There is a trade-off between keeping an accurate and quickly adjusting average and not using up an entire core maintaining that average. We found that this combination works well for responding to quick bursts of program instantiation but does not use too many CPU cycles such that the core is unusable by other applications.

There is another trade-off between using a separate daemon to track the CPU load and simply reading this number as a job is checking in. We found that the Linux-supplied getloadavg system call, which returns the running thread average over the last minute, was not nimble enough to catch shorter running jobs or rapid changes in load average and did not allow RSM to react fast enough to incoming jobs. Similarly, an instantaneous snapshot of the CPU load can provide inaccurate readings, which lead to poor resizing decisions. However, using a separate daemon may also cause some threads to be missed and others to be counted twice. As threads from a job reach a check-in barrier, they no longer become running threads and, therefore, disappear from the thread count in /proc/loadavg. If the first threads to reach the barrier end more than a second before the last thread to reach the barrier, those first threads will be missing from the CPU load seen at check-in. One heuristic we use to get around this limitation is to add the known number of threads of the checking in job to that count to get a more accurate count. If an average over the last second is used, this will cause some threads to be counted twice if they finish within a second of the last thread to finish. However, using an instantaneous count may cause some threads owned by other jobs to be missed completely if they finish around the same time as the job checking in. Additionally, noise from system processes or other unknown processes can cause the single value from an instantaneous read to not be the true count of the CPU load. These incorrect readings can be amortized using an average over the last second. This trade-off is discussed further after our first experiment in Section 4.2.3.

There are a few cases where the CPU load reported by the CPU Load Average Daemon may not actually represent the true CPU load. For example, an interesting phenomenon occurs when the machine goes above capacity. In our synthetic testing framework, a job checks-in immediately following an OpenMP parallel for loop since work is done in the loop and only one thread continues after the loop completes. Since there are synchronization barriers at the end of each OpenMP parallel for loop, some threads can finish an iteration much faster than others due to poor load balancing or oversubscription where threads must contend for cores on the machine. Threads sitting at an OpenMP barrier are not considered "ready-to-run" and, therefore, may not show up in the count of ready and running threads estimated by the CPU Load Average Daemon. We use the following heuristics to compensate for this. If the CPU Load Average Daemon reads a load average less than 4 threads of the current job checking in, we add the current number of threads of the checking-in job to the CPU load that we received. This often leads to an extremely inflated number being used since several threads tend to finish around the same time which causes these threads to be double counted. However, since our resizing decisions are simply expand or contract, the inflated number leads us to the correct conclusion, i.e. to contract. This does cause the heuristic where the machine is just above capacity to not be used very often since the machine is seen to be much higher over capacity than it actually is. This heuristic allows RSM to deal with load imbalance caused by an oversubscribed machine.

3.2.3 RSM Programming Interface

The RSM Interface is defined in a simple header file that user jobs include. Users call rsm_init from an initial thread with the number of threads they wish to start at just as they would call omp_set_num_threads in an OpenMP application. This notifies RSM that the job wishes to begin being monitored and resized without the use of a queuing system. Jobs arrive and immediately run. Resizing occurs when an application calls rsm_check_in. This interface call should occur after computational barriers since only one thread should make this call after all threads have completed their parallel section of work. During this call, the RSM Interface automatically collects necessary information (such as total instructions retired, the

Table 3.1: Duration of Job J in Experiment 0. The table shows the duration in seconds that job J took under the given scheduling condition and with the given number of barriers.

Barriers	Resizing	Static
1	66.521	66.519
10	66.597	66.590
100	66.698	66.769
1000	67.443	67.405

current time, and the number of threads the application was using for redundancy) and sends that information to the RSM Daemon which has the information necessary to convert these numbers into information since the last check-in. This removes the need for the interface to store information from previous check-ins, leaving this responsibility to the RSM Daemon. The RSM Interface collects time through the use of the gettimeofday system call. Thread count is retrieved through OpenMP directives. rsm_check_in will also automatically resize the application to the new thread count received from the RSM Daemon. After the job finishes its computation, it calls rsm_exit to let RSM know that it is finished and to no longer expect information from this job. Simply put, if an application follows a looping pattern with multiple computational barriers, there is very little programming effort required to include the RSM Interface into that application.

We examined the overhead of checking-in through a simple experiment. A single resizable job J runs at machine capacity for approximately a minute. We vary the number of barriers that occur during job J. We ran two cases: one where job J checks-in with RSM and one where job J does not check-in. In this experiment, the best possible resizing decisions that can be made are simply to always leave job J at machine capacity. As seen in Table 3.1, checking-in with RSM essentially performs exactly the same as not checking-in with RSM. RSM makes the correct resizing decision to always keep job J at capacity. This results in less than a tenth of a second difference in the duration between job J checking-in and not checking-in for these runs. In fact, checking-in with RSM actually performs better than not checking-in with RSM in the 100 barriers case. This shows that checking-in with RSM takes less time than standard noise on the machine. Since no job in any of our experiments comes remotely close to achieving even 100 barriers per minute (nearly all do not check in more than 10 times per minute), the time required for our synthetic jobs to check-in with RSM is negligible. Additionally, checking-in more often with the RSM Daemon has little impact since increasing the number of barriers by a magnitude of 3 results in only a 1% increase in duration. In the 1000 barriers case, job J actually checks in more often than the CPU Load Average Daemon. If job J needed to be resized to anything other than full machine capacity, this could lead to overreaction and poor resizing decisions. However, since job J does not need to resize, the extreme amount of barriers and check-ins cause very little increase in duration. The machine used for this experiment is described at the end of Section 4.1.2.

Chapter 4

Experiments and Results

In this chapter, we describe several experiments that illustrate and evaluate the performance benefits made possible by our implementation of RSM compared to standard static scheduling. Section 4.1 describes the experimental setup including the synthetic applications used in our experiments. The remaining sections present results from four experiments. Sections 4.2 and 4.3 detail the first two experiments, which illustrate the performance of RSM in the presence of a "burst" of nonresizable jobs. Sections 4.4 and 4.5 detail the last two experiments, which evaluate RSM's ability to handle multiple resizable jobs.

4.1 Experimental Setup

4.1.1 Parallelizability factor

There are intrinsic properties of any parallel application that determine how well that application performs on a particular parallel platform. Communication between threads, data dependencies, and synchronization can all cause certain jobs to achieve less than perfect parallel speedup. To simplify our experiments and allow us to explore a variety of job mixes, we use a simple model of parallel performance that depends on a single parameter, α, where $0 \leq \alpha \leq 1$. On one end of the spectrum, a job with an α of 1 is embarrassingly parallel and is perfectly scalable. On the other hand, a job with an α of 0 is entirely serial in nature and will not achieve any parallel speedup at all. Jobs with an α between 0 and 1 will scale to an extent but will always achieve less than perfect parallel speedup.

In the experiments described in this chapter, we use synthetic applications with a degree of load imbalance defined by α. In a perfectly load balanced job under ideal conditions, all threads finish at the same time. When given additional threads to complete the same amount of work, that job can scale linearly. However, when there is a load imbalance, a job will not scale as well. Suppose one thread consistently runs longer than all other threads.

The factor by which that thread runs longer relative to the other threads determines how well that job can be parallelized. We use this concept to implement the parallelizability factor α.

In our synthetic applications, we cause thread 1 of a job to dominate the running time of that job. We use α to determine the amount of work that is taken from all other threads and given to the first thread. Even though RSM is perfectly capable of handling heterogeneous jobs that perform varying amounts of work per check-in, we use a constant amount of work per check-in for the synthetic jobs in our experiments to allow us to more easily evaluate resizing decisions.

The following formulas determine the percentage of total work per check-in that each thread performs. Let R_i be the percentage of work done by thread i. Let t be the number of threads that the job is currently running on. The fraction of work done by R_i is given by:

$$R_1 = \frac{t-(t-1)\alpha}{t}$$
$$R_i = \frac{\alpha}{t}, \quad i = 2, ..., t$$

Notice that

$$R_1 + R_2 + R_3 + ... + R_t = 1.$$

Figure 4.2 illustrates the percentage of work done by each thread for various cases. Note that when α is less than 1, thread 1 does more work than all other threads. These percentages lead to the speedup bounds shown in Figure 4.1. These bounds are calculated using the equation $Bound = \frac{1}{R_1}$ since R_1 is the bottleneck by design.

4.1.2 Synthetic Jobs

Our synthetic jobs are implemented in C and use the OpenMP[26] directives for parallelism. Each job consists of RSM Interface calls and an OpenMP parallel for loop to perform useful work (see psuedo code for a synthetic job in Figure 4.3). To simulate the arrival of multiple jobs, we use a bash script to run a series of jobs separated by sleep calls. These synthetic jobs and run scripts are generated by a simple C program for a given set of parameters. A file is passed to this C program to build all jobs within a particular job mix. This file uses a single line to represent a job. Each line contains: the number of seconds that should pass before starting the job following this one, whether the job is resizable or not, the number of threads the job should start with, the number of seconds that should pass between check-ins at the initial thread count, the number of check-ins that should occur during the job, the parallelizability factor α of that job, and optional additional phases that a job may go through by specifying additional α's.

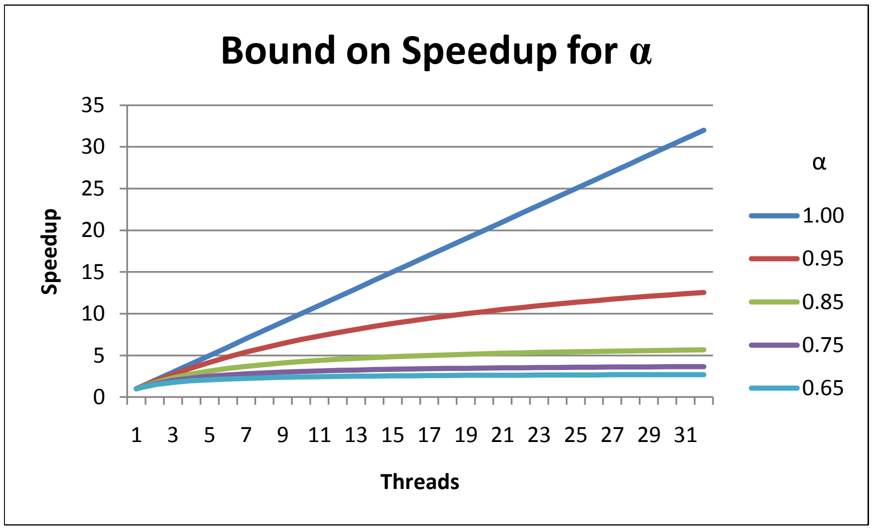

Figure 4.1: Maximum parallel speedups that can be achieved for various values of α.

Each synthetic job performs a constant amount of work between check-ins. This fixed amount of work is determined for each job by taking the number of seconds per check-in multiplied by the initial thread count. These thread-seconds are translated to instructions per check-in by multiplying by a constant instructions per thread-second. This final number represents the total number of simple integer additions that occur between check-ins that are distributed amongst threads of the synthetic job. In our experiments, we always use a 10 second check-in interval for malleable jobs. The percentage of this work per check-in that individual threads perform, R_i, may change after each check-in. R_i is calculated during each execution of the OpenMP parallel for loop. Even though the total work per check-in is fixed, the work per thread can vary drastically based on α.

All of our synthetic jobs are compute bound. The useful work that each job performs is a set number of integer additions. While this does not cover the full scope of scientific applications, compute bound jobs represent the simplest and most intuitive way to demonstrate the power of malleability. More advanced versions of RSM could potentially incorporate advanced heuristics that attempt to distinguish between compute bound, I/O bound, and memory bound applications and perform the resizing decisions necessary to benefit each the most. However, we chose to limit the scope of this work to compute bound jobs.

All of the experiments were performed on a single compute node consisting of Quad-socket AMD 2.3 GHz Magny Cour 8 Core Processors (equating to 32 cores) and 64 gigabytes of memory attached to 40 terabytes of disk storage. The operating system is SUSE Linux Enterprise 11 using kernel version 2.6.32.12.

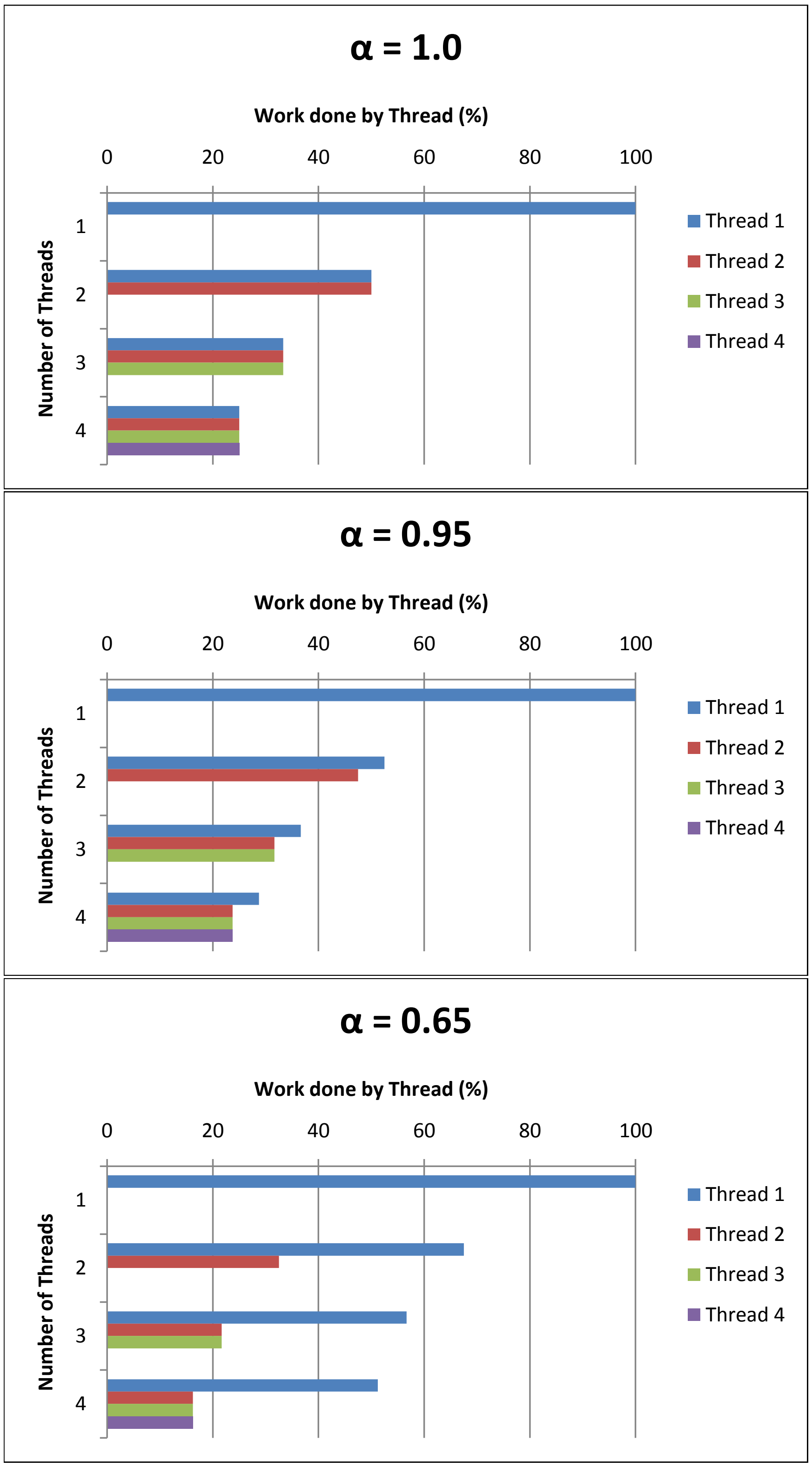

Figure 4.2: Percentages of work done by threads for various values of α.

```
determine total useful work to perform between check-ins
rsm_init ( Initial thread count )
For ( 1 to total number of check-ins )
    OpenMP Parallel For ( thread 1 to total number of threads )
        Get percentage of work to do based on thread number and alpha
        Perform percentage of useful work
    End OpenMP Parallel For
    rsm_check_in
End For
rsm_exit
```

Figure 4.3: Pseudo code for a synthetic job.

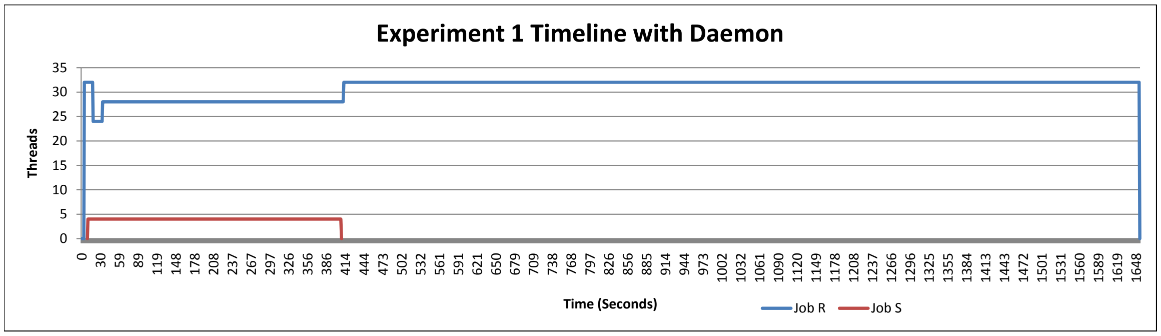

Figure 4.4: Timeline of Experiment 1 when job S begins with 4 threads, showing number of threads used by each job.

4.2 Experiment 1: Single Large Job Arriving

4.2.1 Description

Our first experiment is designed to demonstrate how well the system handles a simple load "burst" caused by the start of a single job that requests a large number of threads. There is one resizable job, R, that runs for the duration of the experiment. One nonresizable job, S, which varies in thread count, competes for resources with job R. Job S always performs a constant amount of work, i.e. running on 2 threads causes job S to finish approximately twice as fast as running on 1 thread. Since job R begins at machine capacity, i.e., 32 threads, any threads that job S requests will put the machine above capacity. The RSM Daemon uses the Greedy resizing policy for this experiment, which does not take a job's performance into account when adjusting the thread count.

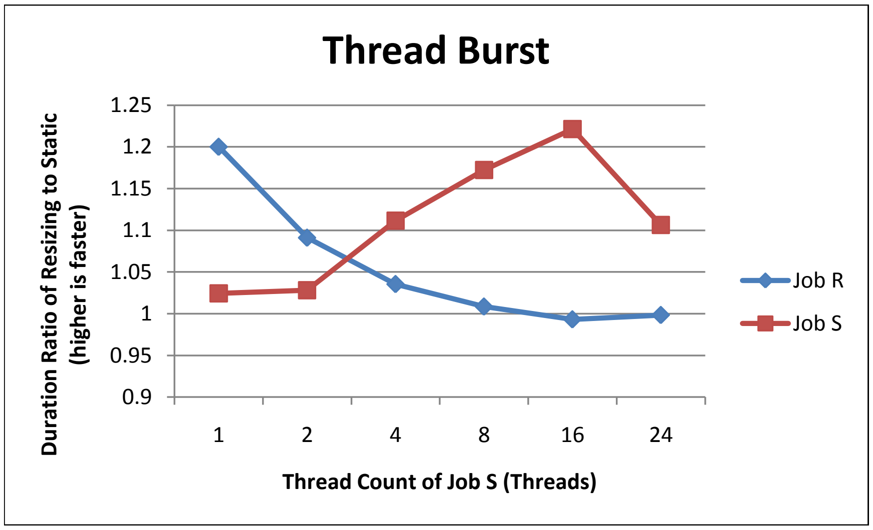

Figure 4.5: Ratios of job R and job S when resized and when scheduled statically in experiment 1. Ratios above 1 correspond to resizing finishing faster than static scheduling.

4.2.2 Results and Discussion

Figure 4.5 shows the duration ratios for jobs R and S of the resizable case compared to the static case. With resizing, job R completes in almost the same time no matter how many threads job S requires. However, with static scheduling, the duration of R is directly proportional to how it contends with job S for cores on the machine. When statically scheduled and job S has 1 thread, job R takes 20% longer (385 seconds) than resizing. As job S grows in threads and shrinks in duration, the static scheduling of job R eventually reaches the same duration as resizing. This happens because a short, intense load burst means that RSM is unable to make the drastic adjustments required to keep the machine at or below capacity before job S finishes, as seen in the zoomed timeline in Figure 4.6. By the time RSM notices and reacts to the burst, it is almost over; it would have been just as good to not resize job R at all, i.e. use static scheduling.

Looking at job S's performance, we see that completion time rises dramatically as the thread count increases. When statically scheduled, job S begins to perform quite poorly with as few as 4 threads. Resizing allows job S to barely notice job R and maintain a consistent duration with as many as 8 threads. However, once job S has 16 threads and two or more resizes of job R are required to get the load average under capacity, job S begins to take drastically longer. The penalty for job S is always lower when job R is resized than it is with static scheduling. Even though job S maintains a constant amount of work, the period of time in which it is in contention for threads is almost the entire experiment when it has 1 thread. However, at 24 threads, job S is only in contention for threads for about 4% of the experiment. As the time that job R and job S contend for cores on the machine in the static case decreases, the time it takes job R to finish also decreases. However, as job S grows

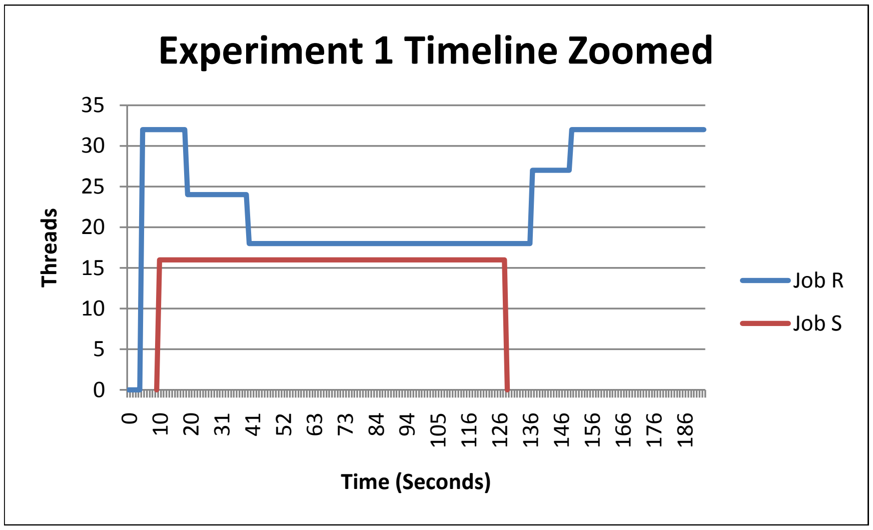

Figure 4.6: Timeline of the first 190 seconds in Experiment 1 when job S begins with 16 threads.

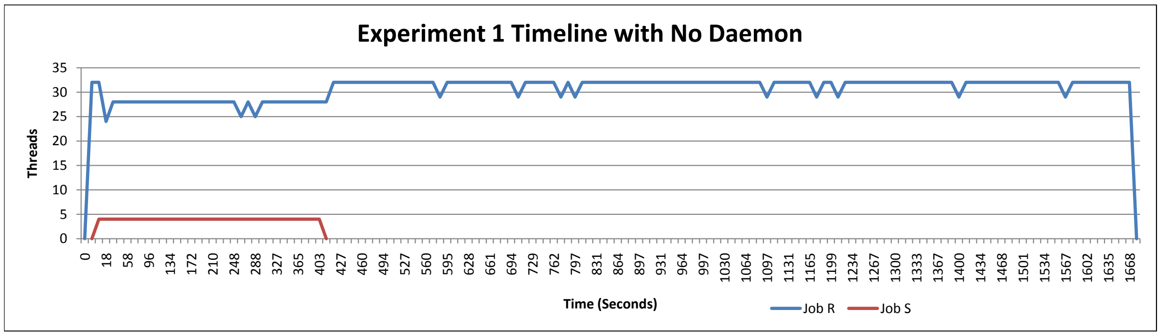

Figure 4.7: Timeline of Experiment 1 when job S begins with 4 threads, showing number of threads used by each job. No CPU Load Average Daemon was used.

larger in thread count, it is impacted much more by job R's presence in the system. Overall, there is a 5% decrease in duration for both job R and job S over all runs.

4.2.3 Use of CPU Load Average Daemon

As described in Section 3.2.2, RSM uses a separate daemon to monitor the load on the machine. We use this experiment as a vehicle for testing our assumption that a daemon is necessary. We investigated the differences between using a daemon to calculate CPU load average and simply using an instantaneous read of the CPU load. Using an instantaneous read could potentially reduce the load that RSM itself places on the system, allowing more cycles to be used for user jobs. However, using this instantaneous read can cause inaccurate

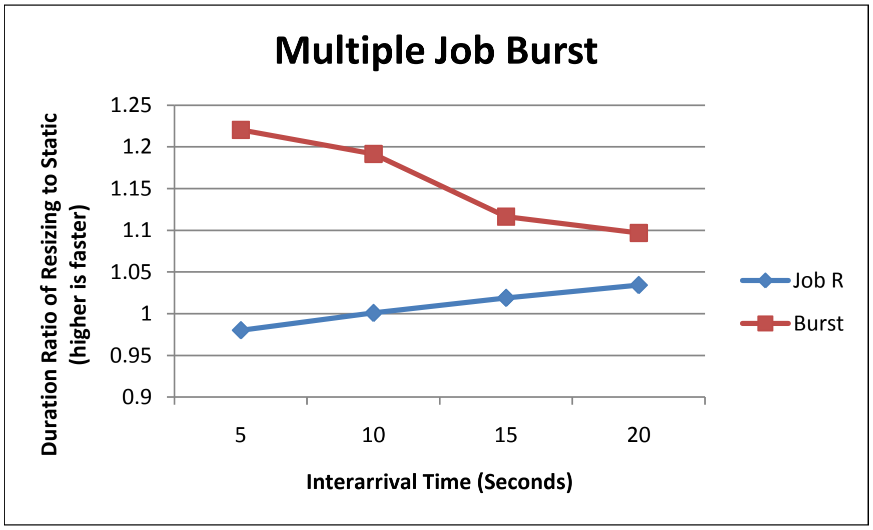

Figure 4.8: Performance ratios of job R and the total duration of all burst jobs for Experiment 2, when resized compared to static scheduling. Numbers greater than 1 mean that resizing finished faster than static scheduling.

CPU loads to be seen on the system, leading to poor resizing decisions.

In this experiment, job R took 4.7% longer on average than an empty machine when there was no daemon. When there was a daemon, job R took 3.5% longer on average than an empty machine. This shows that any benefit seen from not reading the CPU load nearly as often is negated by the poorer quality of information given by this single read. As seen in Figure 4.7, job R is contracted at several random points during the run when it should simply remain constant. These poor resizing decisions lead to the 1% difference in duration between using the daemon and not using the daemon. Using the daemon allows poor load average readings to be amortized over upwards of 9 other reads so that more accurate information is given. Additionally, using an instantaneous read will not notice some threads on the system. If some threads from a job just about to check-in are sitting at a barrier waiting for other threads to finish, these are not considered ready to run threads and will not show up in the load average read for a different job that does check-in. This can lead to poor resizing decisions for both threads. Poor load balancing can compound these poor readings and lead to even poorer resizing decisions. A daemon will notice these changes if the threads finish less than a second before the first job checks-in. In the remainder of the following experiments, we use a CPU Load Average Daemon to determine the CPU load of the system.

4.3 Experiment 2: Many Jobs Arriving at Different Intervals

4.3.1 Description

Our second experiment is designed to measure how well the system handles prolonged "burstiness" caused by multiple nonresizable jobs arriving with various interarrival times. One resizable job, R, begins by filling the whole machine, i.e., with 32 CPU-intensive threads. Job R does the same amount of work in every run. Then, 15 nonresizable jobs arrive with a varied interarrival time, which is randomly chosen between 0 and twice some specified average interarrival time. The order of the nonresizable jobs is varied randomly; a different job can potentially start each time. The first job to arrive always starts at the same time relative to job R, i.e., at 5 seconds after job R begins. Of the 15 nonresizable jobs, 8 jobs have 1 thread, 4 jobs have 2 threads, 2 jobs have 4 threads, and 1 job has 8 threads. Additionally, each thread of these nonresizable jobs always does the same amount of work. This means that a nonresizable job with 4 threads does twice as much work as a job with 2 threads. Additionally, each nonresizable job runs for approximately 30 seconds and has no barriers. Average interarrival times of 5, 10, 15, and 20 seconds are considered. The results in Figure 4.8 are averages of five independent runs for each average interarrival time. At an average interarrrival time of 5 seconds, a maximum of 9 to 12 of the nonresizable jobs run concurrently, and at an average interarrival time of 20 seconds, a maximum of 4 to 5 nonresizable jobs run concurrently. The RSM Daemon uses the Greedy policy for this experiment, which does not take a job's performance into account when adjusting the thread count.

4.3.2 Results and Discussion

From Figure 4.8, it can be seen that job R performs relatively similarly in both the resizing and static cases in the presence of a burst of nonresizable jobs. However, for average interarrival times of 10, 15, and 20, job R performs the same as static scheduling or better. As the average interarrival time increases, job R performs better when resized. This shows that as the RSM Daemon has more time to react to incoming jobs, it begins to outperform static scheduling. It also shows that reducing the number of threads for job R can actually help its performance when the machine is oversubscribed for a lengthy time period. The effect that resizing has on the nonresizable jobs is more significant. The data shown in Figure 4.8 compares the sum of completion times for the 15 rigid jobs in each case. RSM outperforms static scheduling in every case by 9-22%. RSM has some difficulty adjusting job R to jobs incoming at an average interarrival time of 5 seconds, but static scheduling leads to a worse duration overall than when job R was resized. As the average interarrival time of the burst increases, static scheduling performs better and better, but never reaches the performance

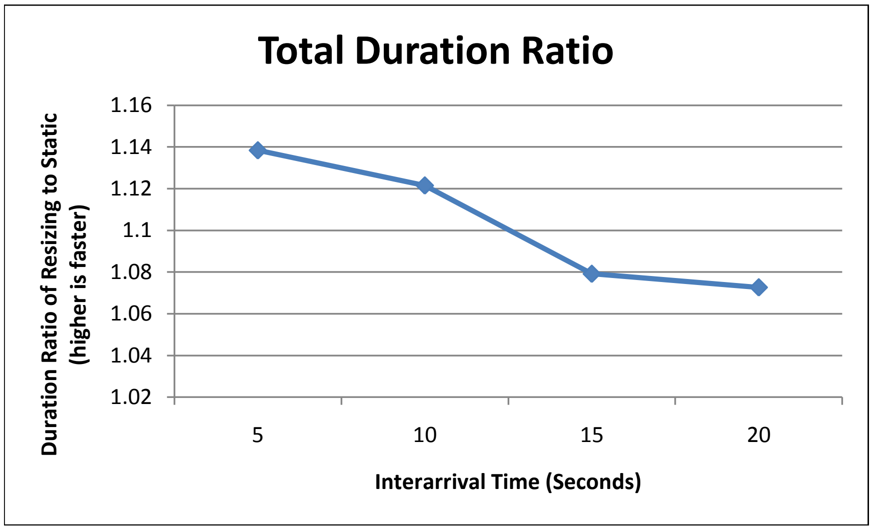

Figure 4.9: Total duration ratios of job R and all burst jobs for Experiment 2, when resized compared to static scheduling.

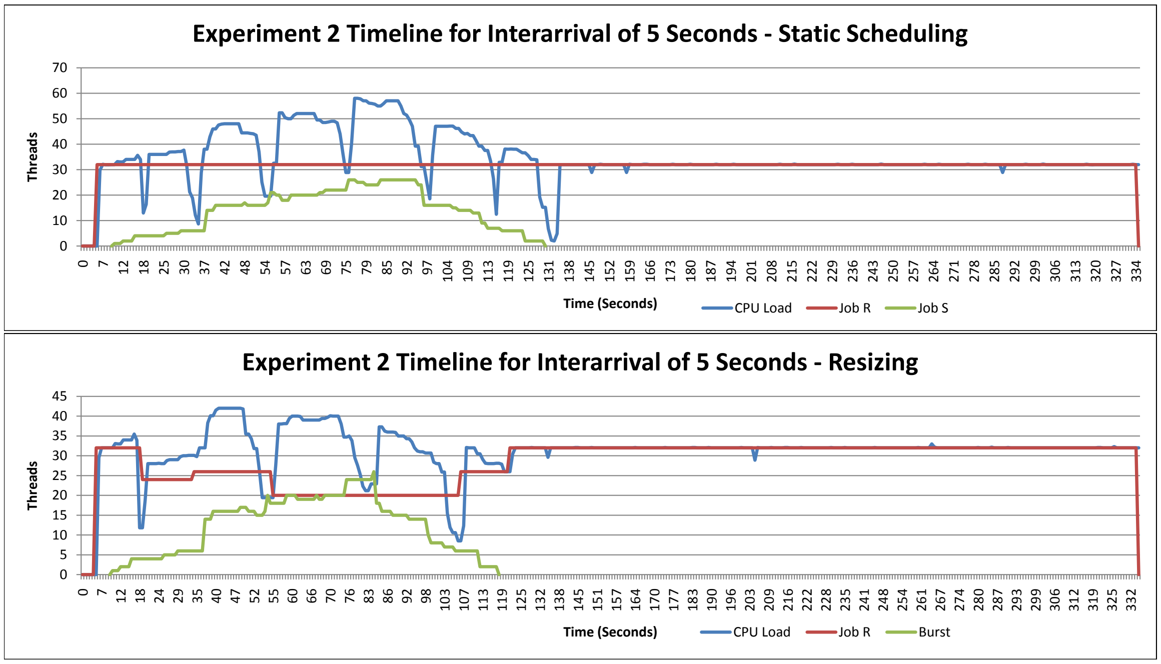

Figure 4.10: Timelines from a run of Experiment 2 showing threads for job R and the burst of jobs with an average interarrival time of 5 seconds. Static scheduling occurs in the top graph and resizing occurs in the bottom graph.

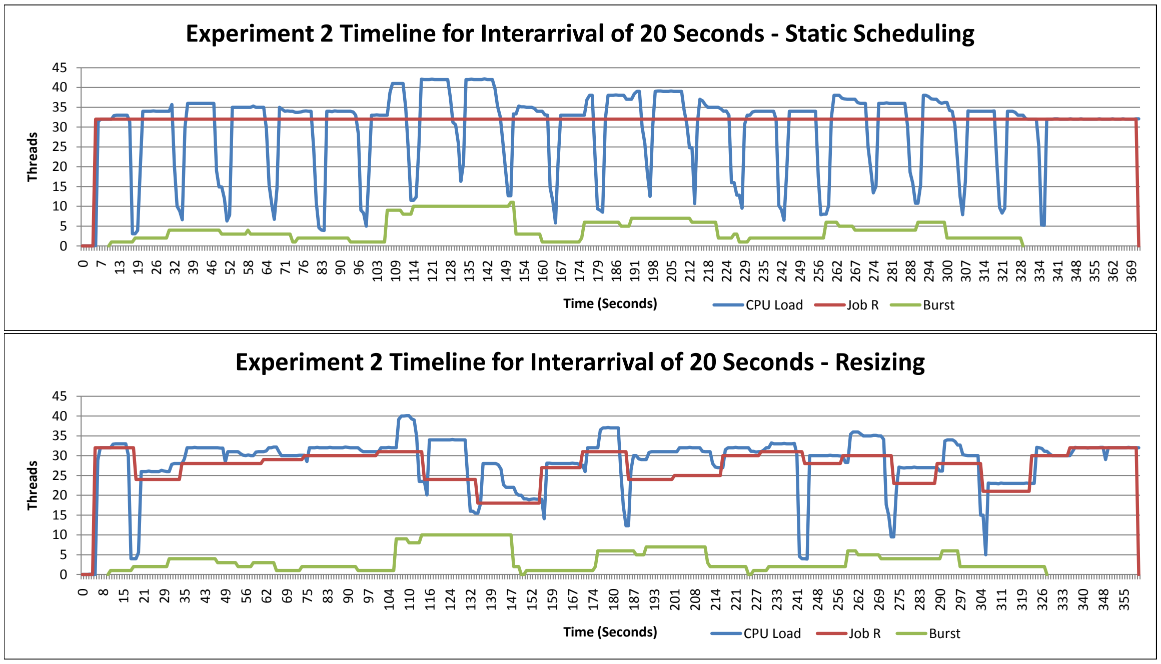

Figure 4.11: Timelines from a run of Experiment 2 showing threads for job R and the burst of jobs with an average interarrival time of 20 seconds. Static scheduling occurs in the top graph and resizing occurs in the bottom graph.

of resizing. This experiment shows that resizing even just one job can significantly benefit other nonresizable jobs. Additionally, resizing does not hurt the resizable job very much and can even lead to performance gains.

The ratio of the total duration of the nonresizable jobs in the burst decreases as the interarrival time grows in both static scheduling and resizing. However, the duration of job R grows as the interarrival time also grows in both scenarios. This shows that short intense bursts have less of an effect on job R compared to prolonged temperate bursts since job R is in contention for resources for a greater percentage of time. The opposite is true of the burst itself. It is much more affected by its short interarrival time and not very affected by longer interarrival times since its duration is much shorter and any delay in available resources is compounded by additional nonresizable jobs arriving. RSM is able to mitigate both conditions. Figure 4.9 shows the performance ratio for all 16 jobs, for each interarrival time. We see an improvement of at least 7% in all cases due to RSM with an overall 10% decrease in duration for all jobs across all runs.

Figures 4.10 and 4.11 show timelines of the number of threads associated with job R and the burst of rigid jobs, along with the CPU load for both the statically scheduled and the RSM case. CPU load is the value supplied by RSM's load average daemon, which reports the average number of ready-to-run threads over the last second. From these figures we see the effect of oversubscription in the static case. During the time of the load burst,

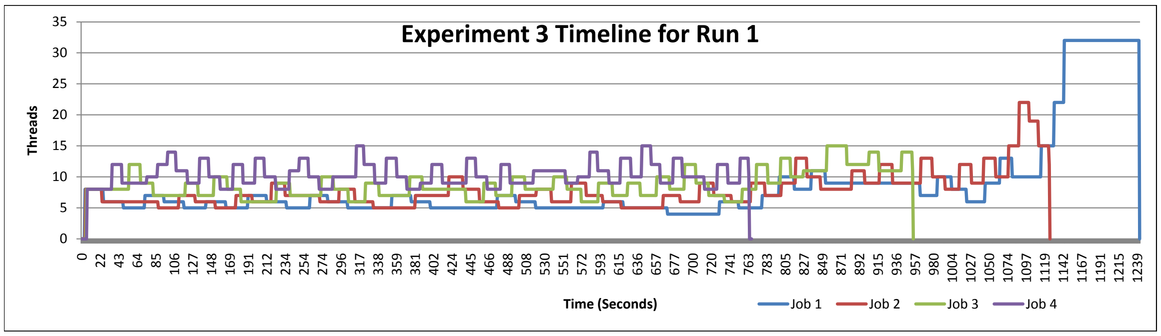

Figure 4.12: Timeline from Run 1 of Experiment 3, showing number of threads used by each job.

when the number of threads one would think are ready-to-run exceeds the number of cores on the machine (32), we see dramatic oscillations in CPU load. These correspond to the synchronization points in job R, when job R checks in with the RSM daemon. Since all threads in job R must reach such a synchronization point before any can proceed farther, the oversubscription causes some of job R's 32 threads to get to the synchronization point before others, which means many threads are waiting at a barrier, i.e., they are not ready-to-run. If there are sufficiently many threads from the burst to keep the cores busy (e.g., Figure 4.10 (top)), the overall effect on utilization and throughput is minimal. But if the drops in CPU load go well below 32 (e.g., Figure 4.11 (top)), the impact on utilization is noticeable. Note also that job R is perfectly load balanced in this experiment; if it were not, the negative impact on utilization of job R's synchronization points would be worse. (We consider cases where resizable jobs are not perfectly parallelizable in the next two subsections.) Finally, the figures illustrate well the smoothing effect that RSM has on CPU load, as it attempts to reduce oversubscription during peak load times, and boost utilization during under-utilized periods. This is most clearly seen in Figure 4.11, where the CPU load curve is noticeably smoother in the RSM case than in the statically scheduled case.

4.4 Experiment 3: Multiple Resizable Jobs

4.4.1 Description

Having demonstrated that RSM can react to nonresizable job bursts, we now consider resizing more than one malleable job in different ways based on performance. Since one goal of RSM is to improve throughput, it is important to reward those jobs that use the system the most efficiently. In this experiment, four resizable jobs arrive at nearly the same time (separated by 0.1 seconds) and have different α's as shown in Table 4.1. Notice that as run number increases, the disparity of α values increases from a difference of 0.02 for Run 1 up to a difference of 0.10 for Run 5. The maximum speedups for each job in Run 5 were shown

Table 4.1: Table of α's used in each run for Experiment 3.

	Job 1	Job 2	Job 3	Job 4
Run 1	0.89	0.91	0.93	0.95
Run 2	0.83	0.87	0.91	0.95
Run 3	0.77	0.83	0.89	0.95
Run 4	0.71	0.79	0.87	0.95
Run 5	0.65	0.75	0.85	0.95

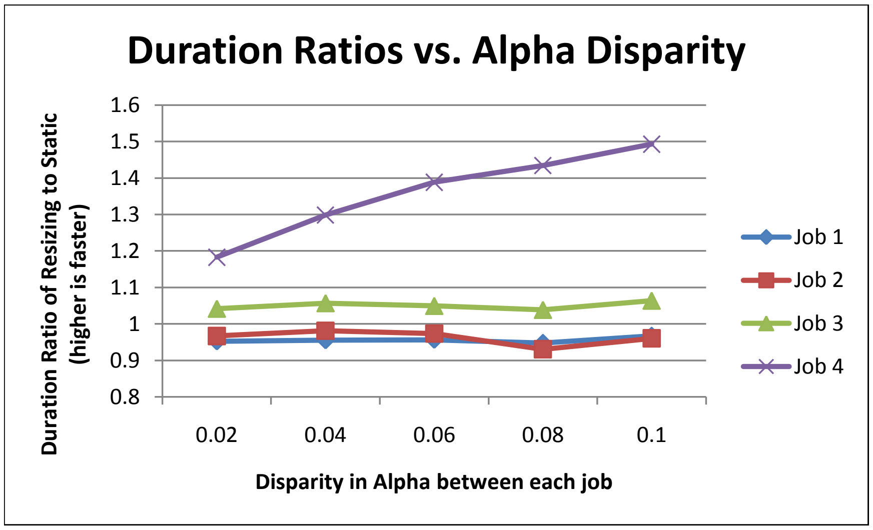

Figure 4.13: Duration ratios for each job for differing values of α in Experiment 3. A ratio greater than 1 means that the resized case is faster than the static case.

previously in Figure 4.1. All of these jobs initially arrive with 8 threads, which fills the machine. All jobs do the same amount of work. We vary the disparity of the α's of the different jobs to show how RSM greatly rewards those with high α's. In this experiment, RSM uses the Global resizing policy to adjust thread counts for resizable jobs.

4.4.2 Results and Discussion

In Figures 4.12, 4.15, and 4.16, we show the thread count history for Runs 1, 3, and 5, respectively. Figures 4.13 and 4.14 compare the performance with and without resizing for each run. Clearly, RSM greatly rewards job 4 since it has the highest α in all cases. RSM also rewards job 3, the job with the second highest α, which also finishes about 5% sooner in the resizable case for every run. These two jobs finish faster than their statically scheduled counterparts. Jobs 2 and 1, the jobs with the second worst α and the worst α respectively,

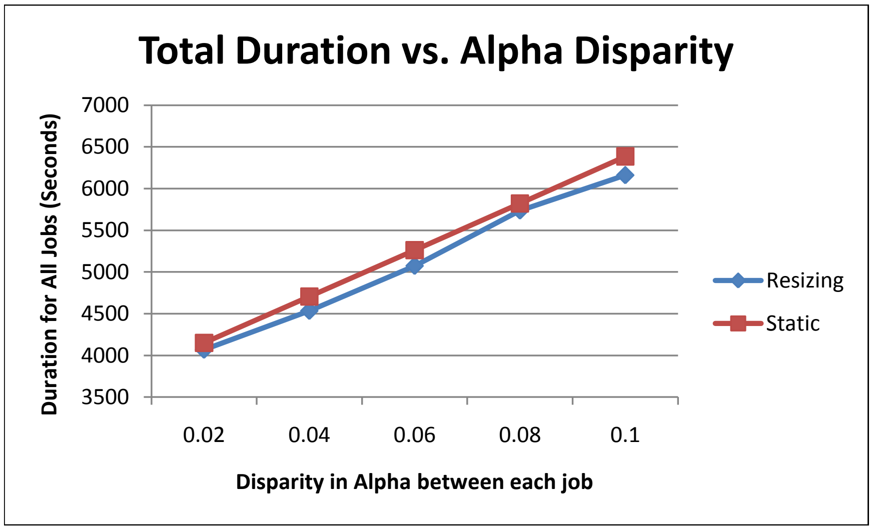

Figure 4.14: Total duration for differing values of α in Experiment 3.

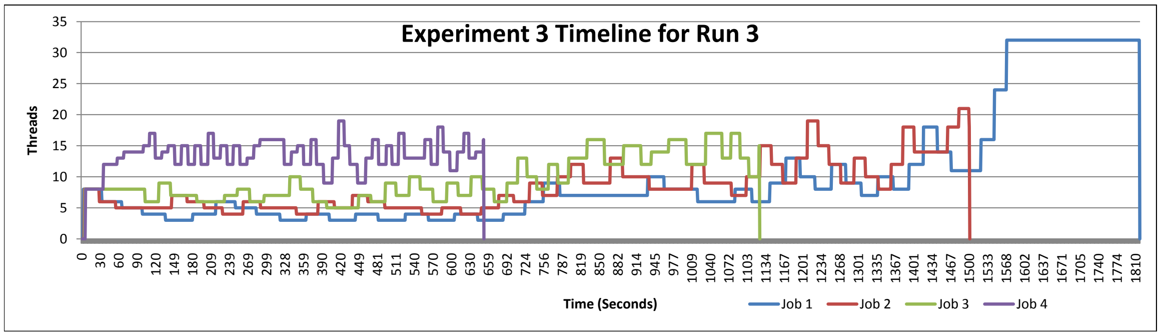

Figure 4.15: Timeline from Run 3 of Experiment 3, showing number of threads used by each job.

finished 96% and 95% as fast as the statically scheduled jobs respectively. Even though these jobs finished slightly slower, the speedups for the jobs with the two highest α's more than make up for this loss in performance. Overall, there is a 3% reduction in job duration for all jobs over all runs equating to 12.6 minutes saved using RSM. In the graphs of the timelines for Run 1, 3, and 5 (Figures 4.12, 4.15, and 4.16), job 4 takes many more threads when there is a larger disparity in α. The same is true for each subsequent job as jobs with higher α's finish. In Run 1, all jobs have relatively similar thread counts, but in Run 5, the job with the highest α has many more threads than the job with the second highest α. This means that RSM sees that the measure of useful work per core for each job is about the same at these thread counts. As the disparity in α grows, the disparity in thread counts that cause similar useful work ratios also grows. It should be noted that the machine was never oversubscribed at any point in this experiment for static scheduling or for resizing.

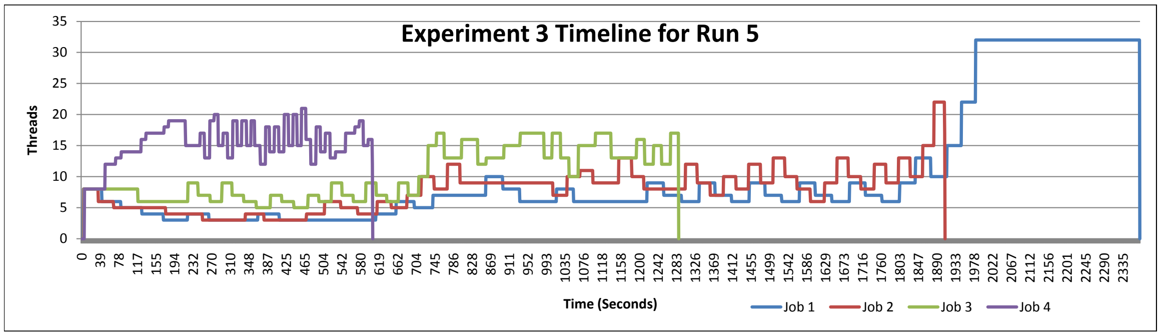

Figure 4.16: Timeline from Run 5 of Experiment 3, showing number of threads used by each job.

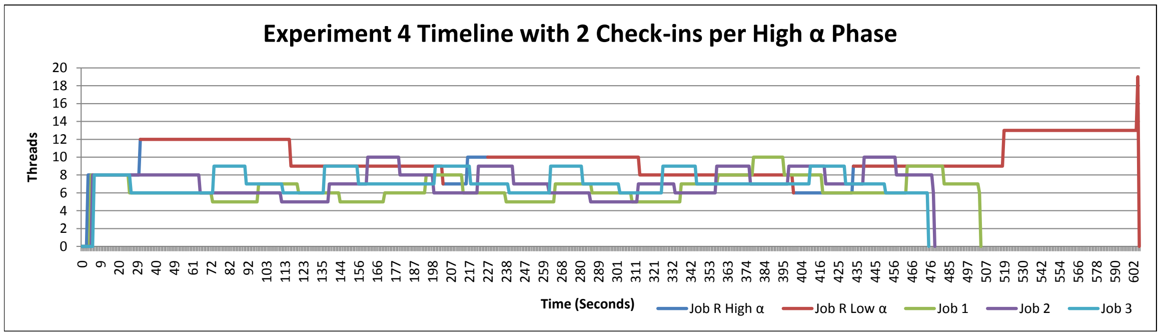

Figure 4.17: Timeline from Experiment 4 when job R's high α phase lasts 2 Check-ins.

4.5 Experiment 4: Multiple Resizable Jobs with Phases

4.5.1 Description

One key issue with jobs that can be resized is that their parallel performance may not be constant, i.e., they may have phases that parallelize well and other phases that do not. Applications that go through phases can truly demonstrate the power of RSM. As an application performs calculations that can be easily parallelized, RSM can expand the job to better utilize the machine. However, when an application goes through a nearly sequential phase, RSM can contract the application to make room for other jobs that can better utilize the machine. This experiment uses one job, R, that goes through phases with a very high α (0.99) followed by a very low α (0.1). We keep the amount of work per check-in constant across all runs but vary the number of check-ins that occur in the high α phase. This means that doubling the number of check-ins in the high α phase causes job R to perform twice the work in that phase. The low α phase always lasts for two check-ins and remains constant in all runs. We vary the number of check-ins for job R's high α phase from 2 to 20. Three other resizable jobs of a lower α (0.9) constantly run while job R goes between different phases. As job R increases in length, the other jobs also increase in length such that they finish at

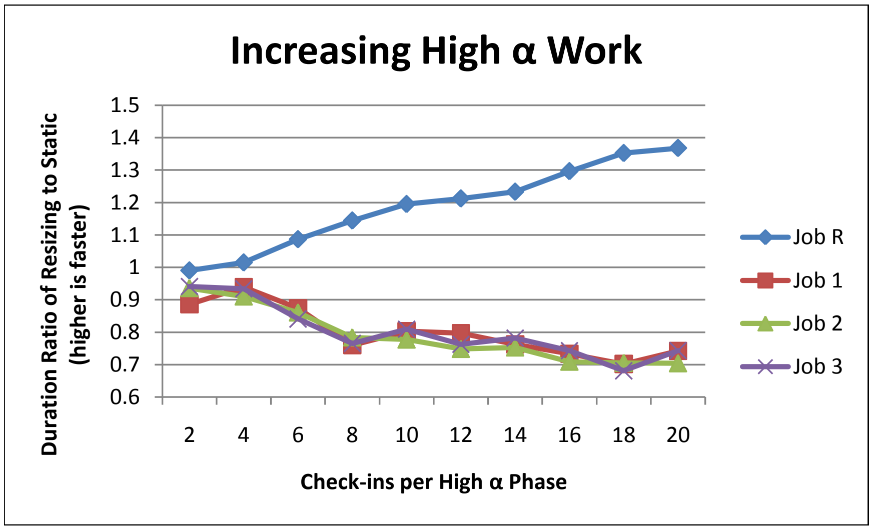

Figure 4.18: Duration Ratio of individual jobs with respect to Phase Length in Experiment 4. The ratio compares resizing to static scheduling with numbers greater than 1 corresponding to resizing being faster than static scheduling.

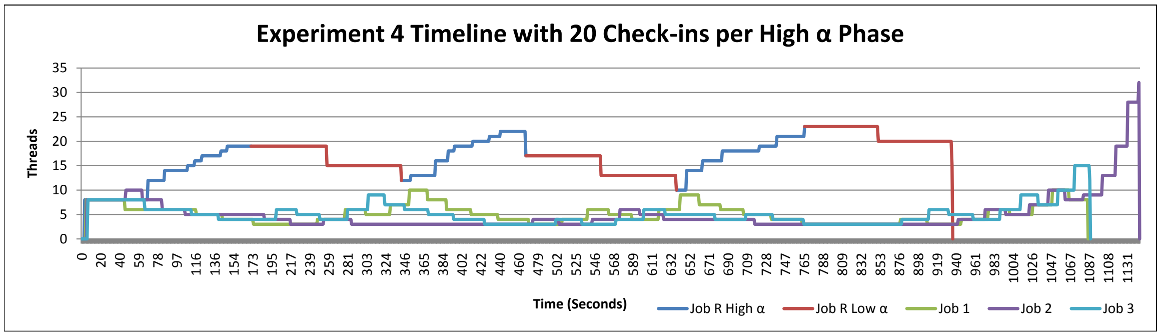

Figure 4.19: Timeline from Experiment 4 when job R's high α phase lasts 20 Check-ins.

about the same time for both resizing and static cases. All jobs start with 8 threads, filling the machine, and at nearly the same time, separated by 0.1 seconds. The Global resizing policy is used by the RSM Daemon for this experiment.

4.5.2 Results and Discussion

As seen in Figure 4.18, with only two check-ins at $\alpha = 0.99$ under RSM, job R finishes at approximately the same time as with static scheduling. This can also be seen in Figure 4.17. Job R receives very few resizes due to the limited number of check-ins. Job R's performance with RSM increases linearly as the length of the high α phase grows. A timeline for 20 check-ins per high α phase is shown in Figure 4.19. Job R dominates the machine when it is

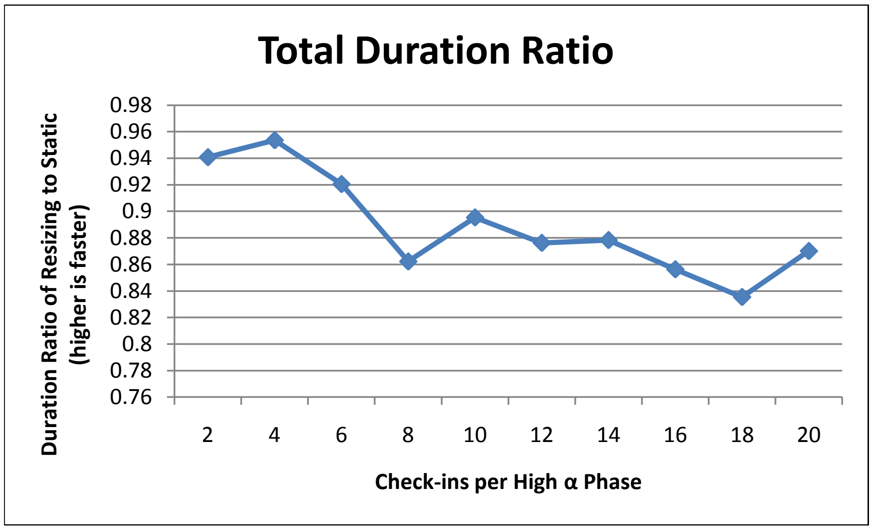

Figure 4.20: Total Duration Ratio of all jobs with respect to Phase Length in Experiment 4. The ratio compares resizing to static scheduling with numbers less than 1 corresponding to resizing being slower than static scheduling.

in the high α phase, then slowly gives up cores during the low α phase. Jobs 1, 2, and 3 all finish slower than their statically scheduled counterparts. As job R's high α phase increases, their performance decreases. This happens because Job R uses more cores of the machine during the high α phase, but does not give up those cores very quickly. In Figure 4.19, Jobs 1 and 3 start to take several cores near the end of job R's low α phase, but then quickly give them up again after a switch in phases.

Figure 4.20 shows that RSM causes a total increase in duration of 12% for all cases, with a greater penalty for longer high α phases. This may seem counter-intuitive, since RSM should be able to give more threads to job R during these long high α phases. From Figure 4.19, we can see, however, that the problem is the low α phases, when the thread count comes down very slowly from the high achieved during the highly parallel phase. RSM takes a long time to lower the thread count of job R in part because of the simple thread-adjustment heuristics used in the current implementation. The more significant reason is that job R takes a long time to check-in when it is running in its low α phase because it is running with poor efficiency. When the high α phase is shorter, RSM does not have time to give job R many more threads (and take threads from the other jobs), so adjusting back to the low α phase is not as difficult.

If job R could inform RSM of the nearly serial code the occurs during a low α phase, then RSM could immediately scale job R back allowing jobs 1, 2, and 3 to take much greater advantage of the idle cores on the machine. Our simplistic approach here does limit the performance benefits seen through resizing. However, there is still great potential demonstrated in this experiment.

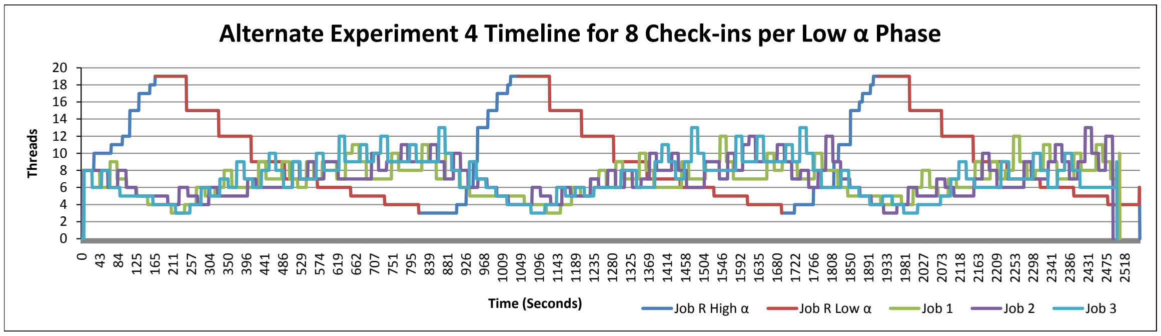

Figure 4.21: Timeline for modified Experiment 4 where job R's low α phase lasts 8 Check-ins.

Table 4.2: Table of duration ratios for a high α phase lasting 20 check-ins and the given low α phase lengths for Experiment 4 and an alternate Experiment 4.

	2 Check-ins	8 Check-ins
Job R	1.37	1.05
Job 1	0.74	0.92
Job 2	0.70	0.91
Job 3	0.74	0.93
Total	0.87	0.95

When the low α phase is long enough to allow jobs 1, 2, and 3 to actually take advantage of the machine, we receive the timeline given in Figure 4.21. In this alternate experiment where the high α phase lasts for 20 check-ins and the low α phase lasts for 8 check-ins, job R gives back enough cores for jobs 1, 2, and 3 to all take advantage of the idle cores on the system. This could be greatly magnified if those idle cores were immediately given up by job R when it enters into a low α phase. The duration ratios for a high α phase of 20 check-ins are shown in Table 4.2. Job R does not perform as well, but jobs 1, 2, and 3 all perform significantly better. Again, these numbers could be greatly increased if job R immediately gave up those cores. These additional notifications are future work that can greatly impact the performance achievable by RSM.

Chapter 5

Conclusion

In this work, we have explored the potential of context-aware malleability on multicore systems. We have built a prototype, RSM, to answer several research questions. We have made several assumptions about the system and the jobs that run on it to achieve these results, such as a general looping pattern for parallel jobs, limits on the amount of thread change in each resizing decision, and a simple winner-or-loser mentality of resizes where resizing decisions are a yes-or-no answer with no form of partial increase or decrease. We have seen that taking performance into account improves throughput for the most scalable job and improves overall system throughput as well. We have determined that obtaining a correct CPU load can greatly impact the potential performance benefits of resizing. Even a slightly incorrect reading or, more importantly, incorrect interpretation of that reading can lead to poor resizing decisions. We have shown that there is little overhead involved with calls to RSM but noted that extreme amounts of check-ins could potentially lead to more poor resizing decisions, making this overhead not worthwhile. We have also identified several areas of difference between a resizing framework for a distributed memory platform (e.g., ReSHAPE) and RSM for a shared memory platform, such as burstiness and oversubscription, and expressed these differences through several experiments.

While this work details a working prototype of RSM, there is still much work left that could greatly improve RSM and build on the research described in this thesis. The first and most obvious improvement would be to run real applications and job mixes using RSM. This would help verify our results from synthetic applications. We could also make several improvements to the RSM Interface, such as extending OpenMP with RSM Interface calls. This could potentially eliminate any programming burden involved with including RSM Interface calls. Another RSM Interface improvement could be the ability to send the RSM Daemon additional information. This information could include a way for an application to inform the RSM Daemon of its ability to take advantage of parallelism, such as setting a maximum number of useful threads for the entire application or phase (especially during nearly sequential pieces of code). This additional information could also include the manner in which the job is being used, e.g., Interactive, High Priority, or Low Priority. This priority system could distinguish certain jobs as more important to users despite their performance. This would be especially useful for interactive jobs that could continually be waiting for user input but require intense computational periods between inputs.

The RSM Daemon could also be greatly improved. More advanced heuristics and resizing policies could lead to much greater performance benefits. With the ability to measure useful work done by a job, RSM could potentially reward heavy I/O jobs or even jobs that are power efficient depending on the desired policy. Another potential improvement could be incorporating RSM into the Linux kernel and scheduler itself. This could potentially improve performance with a single system call to the RSM Daemon. These improvements will be explored in future work.

Bibliography

[1] James H. Anderson and John M. Calandrino. Parallel real-time task scheduling on multicore platforms. In Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International, pages 89 –100, Dec. 2006.

[2] Saisanthosh Balakrishnan, Ravi Rajwar, Mike Upton, and Konrad Lai. The impact of performance asymmetry in emerging multicore architectures. In Proceedings of the 32nd annual international symposium on Computer Architecture, ISCA '05, pages 506–517, Washington, DC, USA, 2005. IEEE Computer Society.

[3] François Broquedis, François Diakhaté, Samuel Thibault, Olivier Aumage, Raymond Namyst, and Pierre-André Wacrenier. Scheduling dynamic openmp applications over multicore architectures. In Proceedings of the 4th international conference on OpenMP in a new era of parallelism, IWOMP'08, pages 170–180, Berlin, Heidelberg, 2008. Springer-Verlag.

[4] J. Buisson, O. Sonmez, H. Mohamed, W. Lammers, and D. Epema. Scheduling malleable applications in multicluster systems. In Cluster Computing, 2007 IEEE International Conference on, pages 372 –381, Sept. 2007.

[5] J.M. Calandrino and J.H. Anderson. Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study. In Real-Time Systems, 2008. ECRTS '08. Euromicro Conference on, pages 299 –308, July 2008.

[6] Márcia Cera, Yiannis Georgiou, Olivier Richard, Nicolas Maillard, and Philippe Navaux. Supporting malleability in parallel architectures with dynamic cpusetsmapping and dynamic mpi. In Krishna Kant, Sriram Pemmaraju, Krishna Sivalingam, and Jie Wu, editors, Distributed Computing and Networking, volume 5935 of Lecture Notes in Computer Science, pages 242–257. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-11322-2_26.

[7] Walfredo Cirne and Francine Berman. A model for moldable supercomputer jobs. Parallel and Distributed Processing Symposium, International, 1:10059b, 2001.

[8] Walfredo Cirne and Francine Berman. Using moldability to improve the performance of supercomputer jobs. Journal of Parallel and Distributed Computing, 62(10):1571 – 1601, 2002.

[9] Pierre-François Dutot and Denis Trystram. Scheduling on hierarchical clusters using malleable tasks. In Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures, SPAA '01, pages 199–208, New York, NY, USA, 2001. ACM.

[10] Alexandra Fedorova, Margo Seltzer, Christoper Small, and Daniel Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In Proceedings of the annual conference on USENIX Annual Technical Conference, ATEC '05, pages 26–26, Berkeley, CA, USA, 2005. USENIX Association.

[11] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. Cache-fair thread scheduling for multicore processors. Technical report, Harvard University, 2006.

[12] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. Parallel Architectures and Compilation Techniques, International Conference on, 0:25–38, 2007.

[13] Alexandra Fedorova, David Vengerov, and Daniel Doucette. Operating system scheduling on heterogeneous core systems. In Proceedings of the First Workshop on Operating System Support for Heterogeneous Multicore Architectures, in conjunction with PACT, 2007.

[14] Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Parallel job scheduling - a status report. In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, Job Scheduling Strategies for Parallel Processing, volume 3277 of Lecture Notes in Computer Science, pages 1–16. Springer Berlin / Heidelberg, 2005. 10.1007/11407522_1.

[15] Dror Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth Sevcik, and Parkson Wong. Theory and practice in parallel job scheduling. In Dror Feitelson and Larry Rudolph, editors, Job Scheduling Strategies for Parallel Processing, volume 1291 of Lecture Notes in Computer Science, pages 1–34. Springer Berlin / Heidelberg, 1997. 10.1007/3-540-63574-2_14.

[16] Matthew Fluet, Mike Rainey, and John Reppy. A scheduling framework for general-purpose parallel languages. In Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP '08, pages 241–252, New York, NY, USA, 2008. ACM.

[17] E. Frachtenberg. Process scheduling for the parallel desktop. In Parallel Architectures,Algorithms and Networks, 2005. ISPAN 2005. Proceedings. 8th International Symposium on, page 8 pp., Dec. 2005.

[18] Jan Hungershöfer, Achim Streit, and Jens michael Wierum. Efficient resource management for malleable applications, 2001.

[19] L.V. Kale, S. Kumar, and J. DeSouza. A malleable-job system for timeshared parallel machines. In Cluster Computing and the Grid, 2002. 2nd IEEE/ACM International Symposium on, page 230, May 2002.

[20] R. Knauerhase, P. Brett, B. Hohlt, Tong Li, and S. Hahn. Using os observations to improve performance in multicore systems. Micro, IEEE, 28(3):54 –66, May-June 2008.

[21] Mario Lassnig, Thomas Fahringer, Vincent Garonne, Angelos Molfetas, and Miguel Branco. Identification, modelling and prediction of non-periodic bursts in workloads. In Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, CCGRID '10, pages 485–494, Washington, DC, USA, 2010. IEEE Computer Society.

[22] Tong Li, Dan Baumberger, David A. Koufaty, and Scott Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In Proceedings of the 2007 ACM/IEEE conference on Supercomputing, SC '07, pages 53:1–53:11, New York, NY, USA, 2007. ACM.

[23] David Lifka. The anl/ibm sp scheduling system. In Dror Feitelson and Larry Rudolph, editors, Job Scheduling Strategies for Parallel Processing, volume 949 of Lecture Notes in Computer Science, pages 295–303. Springer Berlin / Heidelberg, 1995. 10.1007/3-540-60153-8_35.

[24] Jiayuan Meng, Jeremy W. Sheaffer, and Kevin Skadron. Exploiting inter-thread temporal locality for chip multithreading. In IPDPS, pages 1–12, 2010.

[25] Tran Ngoc Minh, Lex Wolters, and Dick Epema. A realistic integrated model of parallel system workloads. In Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, CCGRID '10, pages 464–473, Washington, DC, USA, 2010. IEEE Computer Society.

[26] OpenMP, 2011. URL: http://openmp.org/wp/.

[27] J. K. Ousterhout. Scheduling techniques for concurrent systems. Proceedings (International Conference on Distributed Computing Systems : 1990) / International Conference on Distributed Computing Systems, pages 22 – 30, 1982.

[28] Perfmon2 libpfm v4.0, 2011. URL: http://perfmon2.sourceforge.net/.

[29] Mohan Rajagopalan, Brian T. Lewis, and Todd A. Anderson. Thread scheduling for multi-core platforms. In Proceedings of the 11th USENIX workshop on Hot topics in operating systems, pages 2:1–2:6, Berkeley, CA, USA, 2007. USENIX Association.

[30] K.C. Sevcik. Application scheduling and processor allocation in multiprogrammed parallel processing systems. Performance Evaluation, 19(2-3):107 – 140, 1994.

[31] D. Shelepov and A. Federova. Scheduling on heterogeneous multicore processors using architectural signatures. In Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture, 2008.

[32] Allan Snavely and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. SIGPLAN Not., 35:234–244, November 2000.

[33] Allan Snavely, Dean M. Tullsen, and Geoff Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, SIGMETRICS '02, pages 66–76, New York, NY, USA, 2002. ACM.

[34] Fengguang Song, S. Moore, and J. Dongarra. Analytical modeling and optimization for affinity based thread scheduling on multicore systems. In Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on, pages 1 –10, 31 2009-Sept. 4 2009.

[35] S. Srinivasan, S. Krishnamoorthy, and P. Sadayappan. A robust scheduling technology for moldable scheduling of parallel jobs. In Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on, pages 92 – 99, Dec. 2003.

[36] Rajesh Sudarsan and Calvin J. Ribbens. Efficient multidimensional data redistribution for resizable parallel computations. In Ivan Stojmenovic, Ruppa Thulasiram, Laurence Yang, Weijia Jia, Minyi Guo, and Rodrigo de Mello, editors, Parallel and Distributed Processing and Applications, volume 4742 of Lecture Notes in Computer Science, pages 182–194. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-74742-0_19.

[37] Rajesh Sudarsan and Calvin J. Ribbens. Reshape: A framework for dynamic resizing and scheduling of homogeneous applications in a parallel environment. In Parallel Processing, 2007. ICPP 2007. International Conference on, page 44, Sept. 2007.

[38] Rajesh Sudarsan and Calvin J. Ribbens. Scheduling resizable parallel applications. Parallel and Distributed Processing Symposium, International, 0:1–10, 2009.

[39] Rajesh Sudarsan and Calvin J. Ribbens. Design and performance of a scheduling framework for resizable parallel applications. Parallel Computing, 36(1):48 – 64, 2010.

[40] David Tam, Reza Azimi, and Michael Stumm. Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. In Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, EuroSys '07, pages 47–58, New York, NY, USA, 2007. ACM.