Performance study and dynamic optimization design for thread pool systems

by

Dongping Xu

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Brett Bode, Co-major Professor
Daniel Berleant, Co-major Professor
Shashi K. Gadia

Iowa State University

Ames, Iowa

2004

Graduate College
Iowa State University

This is to certify that the master's thesis of

Dongping Xu

has met the thesis requirements of Iowa State University

Co-major Professor

Co-major Professor

For the Major Program

DEDICATION

To my parents and my husband, for their love!

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

I am grateful to the many people who helped me in my research and the writing of this thesis.

I would like to give my special thanks to Dr. Brett Bode and Dr. Daniel Berleant for their guidance, patience, mentoring, and support throughout the preparation of this thesis. Their insightful advice has been invaluable to my graduate study in Iowa State University. I would also like to thank Dr. Shashi K. Gadia for his kindness to serve on my committee and to offer his help at any time.

This work was supported in part by Department of Energy (DOE) Scientific Discovery through Advanced Computing (SciDAC) project. I want to thank DOE for providing the funding of this research.

ABSTRACT

Thread pools have been widely used by many multithreaded applications. However, the determination of the pool size according to the application behavior still remains problematic. To automate this process, in this thesis we have developed a set of performance metrics for quantitatively analyzing thread pool performance. For our experiments, we built a thread pool system which provides a general framework for thread pool research. Based on this simulation environment, we studied the performance impact brought by the thread pool on different multithreaded applications. Additionally, the correlations between internal characterizations of thread pools and their throughput were also examined. We then proposed and evaluated a heuristic algorithm to dynamically determine the optimal thread pool size. The simulation results show that this approach is effective in improving overall application performance.

CHAPTER 1. INTRODUCTION

1.1 The Scalable Systems Software Project

As scientific and engineering research continues to evolve, more and more problems rely on high performance computers to compute solutions. Such problems exist in various research areas, including nuclear reaction simulation, global climate change simulation, and protein folding modeling. The continuously increasing demand for computing power pushes parallel computing systems to employ similarly increasing quantities and numbers of components (CPU, memory, and disk) to match the performance requirements. As an example, Earth-Simulator, which is the fastest computer on the planet, has 5120 processors and 10TB of main memory [4]. Even though such developments shed new light on other research fields, the increase in complexity required to manage the resources of a complex terascale computer becomes a critical issue.

To solve this problem, it is desirable to have a system which can control and manage the computing resources (such as compute nodes, networks, and storage systems) with less human interference. A few software packages, including PBS [16], LSF [12], Loadleveler [7], and Condor [13], have been designed and made available for Massive Parallel Processing (MPP) systems by different vendors.

Although existing systems are capable of doing resource management on specific platforms, they are still inadequate for our requirements. In particular, we need a system that allows us to do resource management on heterogeneous environments with special features: ultra-scalable (on thousands of nodes), secure, robust, and load-balanced. For

these reasons, a new terascale resource management project, the *Scalable Systems Software Enabling Technology Center*, was proposed and is currently being carried out [5].

As a part of the SciDAC program, the Scalable Systems Software center provides an application suite designed to support computers that scale to very large system sizes without requiring that the number of support staff to scale along with the machine. Beyond that, this project also intends to create an interoperable framework by defining a software architecture and interfaces between system components. This makes it much easier and cost effective for supercomputer centers to adapt, update, and maintain the components in order to keep up with new hardware and software.

1.2 Design of Job Manager and Queue Manager

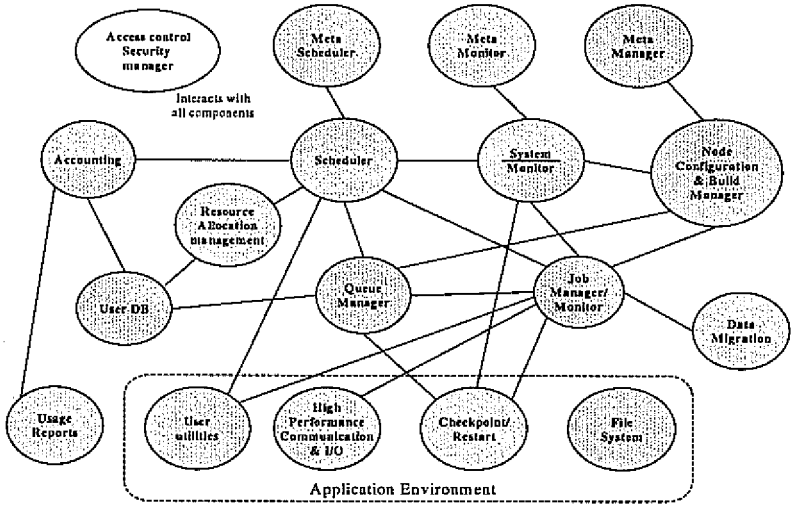

Figure 1.1. The architecture prototype of Scalable Systems Software [5].

As shown in Figure 1.1, the *job manager* is an important component of Scalable Systems Software. The job manager is responsible for managing the jobs submitted by users. In particular, this manager will take care of job submission, job execution and

resource tracking. The purpose of the job manager in the resource management system is to provide a framework which can handle a parallel job consisting of multiple processes on multiple hosts as a single entity. Therefore, these jobs can be executed, suspended, resumed, and terminated easily.

Because it is normal to have a large number of users (>1000) in a large parallel computing system, the job management system has to be deployed with efficient scheduling algorithms to satisfy a high volume of job requests. For this purpose, in our project, there is a component, named the *queue manager*, which is responsible for handling the queuing service to the system. All jobs are maintained in the queue manager, and the job manager will communicate asynchronously with the queue manager as a set of persistent daemons. A queue manager provides functions so that the job manager can create new queues, query jobs, and alter the properties of existing jobs.

1.3 About the Thread Pool Management

As a component of the Scalable Systems Software Project, our queue manager system has to handle the job requests coming from many different users. Because this system is targeted at computing systems with very large node counts, a scalable job manager which can handle a high volume of requests must be deployed. Although existing multiprocess programming techniques are capable of improving the system throughput, they are still inadequate for our requirements. The overhead for managing processes in the operating system is very large, which makes it impractical to use a multiple process model in our job manager. One of the most elegant ways to solve resource- and data-sharing problems is to have multiple light-weight threads of execution inside a single process, which is also

called multithreading. It has been proven that this approach can improve responsiveness and performance of applications [17].

While multithreading provides a clean design approach to handling asynchronous requests, the architecture used to implement multithreading can have a large impact on the thread-creation overhead. Two models, including *thread-per-request* and *thread pool*, are widely used in multithreading programming. The thread-per-request model spawns a thread for each request, and destroys the thread after finishing the request. In contrast, thread pool system spawns and maintains a pool of threads. When a request arrives, the system uses a free thread in the pool to serve a client request, and returns the thread to the pool after finishing the request. Experimental studies suggest that thread pool model can significantly improve system performance and reduce response time [3][15]. Because of its benefits, thread pools have been adopted by a large number of popular server applications, such as Apache and Windows IIS [1][9].

However, optimizing the characteristics of a thread pool remains a trial and error process. Thus many server applications have to rely on system administrators to tune up the thread pool based on their experience. To solve this problem, in this thesis we developed a set of performance metrics for quantitatively analyzing the thread pool performance. These metrics cover three major aspects in a thread pool, including *QoS to submitted tasks*, *throughput of thread pool system*, and *OS workload*. Based on these metrics, we systematically studied the performance and characterizations of the thread pool system.

Additionally, we evaluate the idea of using a heuristic approach to determine the optimal thread pool size based on the information collected so far. This approach makes a

tradeoff between the thread pool performance and the management overhead. The simulation results show that dynamic optimization for thread pool size is very effective in alleviating the management overhead and improving the overall performance. The results imply the potential benefits of using dynamic optimization to replace manual configuration in large multithreaded server applications.

1.4 Organization

The rest of this thesis is organized as follows. In Chapter 2, a detailed description of a thread pool is presented. In particular, we focus on previous works and their problems. The design rationale of our new thread pool research is mentioned in this chapter as well. Chapter 3 presents the performance metrics used in our quantitative analysis. The details of our thread pool implementation and the experimental environment are presented in Chapter 4 and Chapter 5, respectively. Chapter 6 presents our experimental results. Finally, Chapter 7 summarizes this thesis.

CHAPTER 2. DESIGN OF THE THREAD POOL

2.1 Multithreaded Programming

Early on, programs were written using a sequential model, in which code is executed one instruction after the next in a monolithic fashion. However, because of various types of dependencies, programs have to enter an idle state and wait for data to arrive from time to time. This can result in large idle times which are unacceptable because they can dramatically decrease system performance. As an example, for a disk access, applications have to stop until the data is retrieved from disk. During this period, applications can not do any job and all the computational power of CPU is wasted.

To avoid this problem parallelism is introduced on multiple levels. Architecture and system levels become the first choices to exploit the parallelism. As the speed of computers increased, it became advantageous to run multiple programs simultaneously either by time slicing in a single CPU or through the use of multiple processors in an SMP system.

On the software level, *multiprocessing* is an approach to improve the parallelism on uniprocessor machines. The idea of multiprocessing is to have the operating system spawn multiple instances of applications. Each instance is treated as a process and executed independently. Whenever the processor is in idle state, the OS will schedule another instance for execution. This provides the ability to perform multiple tasks simultaneously.

However, multiprocessing does not come without cost. Usually, these processes do not share a common address space. Instead they are clearly separated from each other, and each requires special creation of a process address space. Therefore the overhead to spawn

a new process is relatively large. For a server system which has to handle a large number of service requests, the machine's performance and responsiveness can be significantly impacted when the server application has to spawn many processes. What is worse, each process will occupy a lot of memory, which can quickly exhaust the system resources.

One of the most elegant ways to solve these problems is to have multiple light-weight threads of execution inside a single process. This is called *multithreading* model. In this model, multiple threads are spawned within the same process and the threads share most process resources with each other. Only the minimum information which is required to run one thread is separate. With finer granularity than processes, threads usually improve responsiveness and performance of the application by allowing threads to share resources and reducing the context switch overhead.

Currently the most widely used threading programming interface is *POSIX Threads* (*Pthreads*), which was standardized by ISO in 1996 (POSIX 1003.1c standard). This standard defines the interfaces which must be supported by the pthreads library. The internal design of the Pthreads library is decided by the designer of the library. Different types of Pthread library implementations, which can be classified as *user-space threading*, *kernel-space threading* and *hybrid threading*, have been available on various operating systems such as Unix, Linux, and Microsoft Windows.

Even though multithreading has better performance than multiprocessing, it has some limitations. One big concern with multithreading is security. Because all threads are sharing the same data space, it increases the possibility for malicious users to compromise other users by accessing the threads they are using. Multiprocessing does not have this problem because the data space is independent between different users. Therefore, on a

heterogeneous environment, the security issue must be taken into serious consideration while adopting multithreading. Since our thread pool is used for the Scalable Systems Software Project only, we do not consider the security issue of using multithreading further in this thesis.

2.2 Client-Server Model and Thread Pool

The client-server model is one of the most widely used programming models used in server applications (Figure 2.1). In this model, whenever needed, clients launch requests from the client machine to the server, on which there is a server process waiting for an incoming request by listening on a specific port. The server process will typically handle the request by spawning a new thread (or a process) and assigning the request to it. This new thread will do whatever it is supposed to do (e.g. retrieving the web page in the case of a Web server). The results will be sent back to the client as a response, as shown in Figure 2.1.

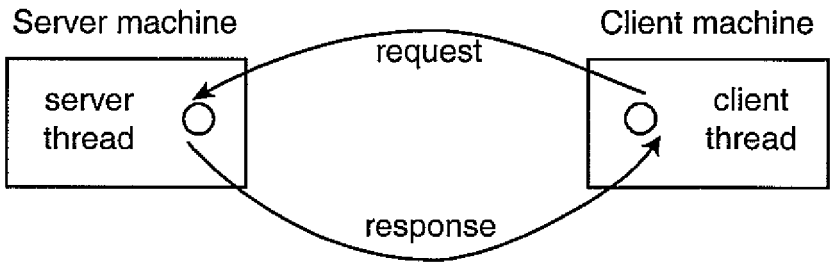

Figure 2.1. The client-server model

This model is pretty straightforward and will function well when the number of clients is small. Unfortunately, the server will become the performance bottleneck when the number of simultaneous requests becomes huge, as the overhead spent on spawning new

threads will become intolerable. Solving this problem efficiently requires addressing issues of response time, scalability, and throughput.

One solution to this problem is an approach called the *thread pool*, which is depicted in Figure 2.2. The idea of a thread pool is to maintain a pool of pre-created threads and dispatch new tasks to idle threads. Each thread can be in one of two states: *idle* and *working*. The state transition diagram is shown in Figure 2.2(b). When a new task is assigned, an idle thread becomes a worker thread and will be moved from the idle thread queue to the worker thread queue. After the task is finished, it will be moved back to the idle thread queue again and will wait for the next request.

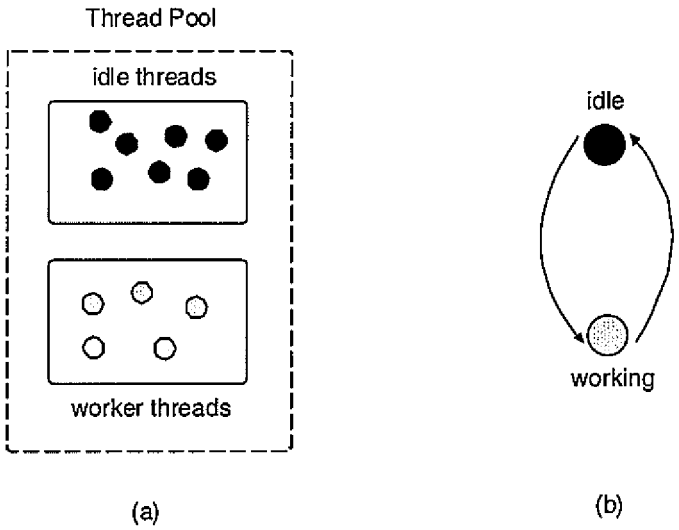

Figure 2.1. Schematic view of a thread pool. (b) state transition diagram of a thread.

Compared with creating threads as needed, the thread pool design only incurs the overhead of creating new threads at the beginning. Because only a fixed number of threads are created and they are continually reused, the total overhead is constant instead of proportional to the number of incoming requests. In addition, the thread pool can dispatch an idle thread for each incoming request almost instantly. This is much faster

than the traditional approach in which the request can only be served after a new thread (or process) is created. Thus, a thread pool will have a better response time.

2.3 Existing Thread Pool Implementations

Because of its advantages, the thread pool has been widely used by various different applications (especially server applications). The performances of these applications rely in part on the throughput which can be delivered by the thread pool. According to the introduction above, the major factor which determines the thread pool performance is the pool size. With a larger pool size, the thread pool can handle more tasks simultaneously with a fast response time. However, this does not come without cost. As the pool size increases, the overhead of thread pool management will become severe and degrade the system performance eventually. Therefore, how to handle this tradeoff becomes important for picking a better pool size.

To solve this problem, people have proposed different approaches to tune the thread pool system. In this section, two representative categories of thread pool size tuning methods are introduced. The advantages and shortcomings of each approach are elaborated as well. The solutions will be discussed in the next section.

2.3.1 Experience-based Approaches

Although widely adopted by many server applications, the thread pool size configuration is still determined based on experience. In this approach, the system administrator is required to constantly monitor the system performance. Whenever a performance bottleneck is noticed, they will tune the configuration to optimize the

performance. To be more concrete, the following examples show the approaches used by Apache and Microsoft IIS, which are the most widely-used Web servers.

Apache. Since Apache is open source and free of charge, it is the most popular web server used around the world. Around 64% of web sites on the Internet choose Apache as their web server. The popularity has proven that the Apache project provides a secure, efficient and extensible HTTP services in sync with the current HTTP standards.

The thread pool system of Apache is named the *Multi-Processing Module* (*MPM*), which implements a hybrid multi-process multi-threaded server. The most important directives used to control this module are `ThreadsPerChild`, which controls the number of threads deployed by each child process and `MaxClients`, which controls the maximum total number of threads that may be launched.

Microsoft IIS. *Microsoft Internet Information Server* (*IIS*) is another popular web server, which is designed for running on Microsoft Windows operating systems. Because of the dominant status of Microsoft Windows, this web server is also very widely used. Like Apache, this server application is also relies on users for performance tuning. According to [9], system administrators are suggested to use PerfMon, a performance monitor application, to collect the statistics (including processor time, request frequency, queue length, and number of concurrent users, etc). In IIS, the initial number of threads per CPU is set to 10 and will change according to the request frequency. Using `ProcessorThreadMax`, users can also specify the maximal number of threads served by one CPU.

Such experience-based approaches have serious drawbacks. The performance monitoring job is very time-consuming and inconvenient for a system administer. In

addition, the configuration drawn up through this approach is often inaccurate, especially when the performance varies a lot over time. Without a theoretical justification, such configurations tend to create too large or too small thread pools. To solve this problem, it is desirable to have a thread pool which can configure itself based on the current status of server system.

2.3.2 Theoretical Approaches

To solve the problems of the experience-based approaches, some researchers have proposed schemes to predict the optimal thread pool size based on heuristic factors [8]. Usually, such approaches have a formula to calculate the thread pool size by using some performance metrics which can be obtained during runtime. The thread pool size will be changed on-the-fly based on this formula. For example, the idea used in [8] for thread pool size estimation is based on the consideration of overhead. Two major metrics, thread creation overhead (c_1) and maintenance overhead (c_2), are used in [8]. Whenever it finds that the cost to maintain the thread pool is larger than the benefits it brings, the system will decrease the number of threads maintained in the thread pool. To obtain accurate system status, all metrics are continuously monitored.

Unfortunately, the formula usually is very complicated. For example, [8] uses calculus to calculate the expected gain as below, where $p(r)$ is the probability density.

$$E(n) = \int_0^n (c_1 \cdot r - c_2 \cdot n) \cdot p(r)dr + \int_n^\infty (c_1 \cdot n - c_2 \cdot n) \cdot p(r)dr$$

It is hard, if not impossible, to apply those formulas in practice because of the complexity. Additionally, some variables used in these formulas are imaginary, such as the probability density $p(r)$ which is assumed to follow a Poisson distribution. With those variables, this

formula cannot model the real behavior of a thread pool system with high accuracy. What is worse, the cost of monitoring those variables is also high.

2.4 Solutions

In summary, instead of using manual tuning, we need a thread pool which can adjust its pool size on-the-fly to have better performance. Existing theoretical approaches have different types of limitations, such as the complexity and the inaccuracy of modeling. To solve this problem, we want to construct a new dynamic optimization approach which is simpler to avoid huge runtime overhead. In addition, in our approach we want to use metrics which can be obtained on-the-fly easily. Bearing these two characteristics in mind, we expect our approach will be more suitable for real implementation.

CHAPTER 3. PERFORMANCE METRICS

Instead of the experience-based approach, we are searching for a quantitative approach which allows us to predicate the thread pool size without interference from humans. According to the discussions in last chapter, we do not want to use an unrealistic formulas for our system. Instead, the performance of our approach relies on a carefully selected set of performance metrics, which are measurable during runtime. In this chapter, a set of metrics will be selected and their properties will be depicted in depth. In the next few chapters, these metrics will be used to evaluate a scheduling algorithm.

3.1 Selection Criteria

There are many possible quantitative criteria for evaluating a scheduling algorithm. Because we want to use these metrics to evaluate the thread pool algorithms, they must meet the following criteria.

- **Measurable.** The first concern about the criteria is whether we can measure it. It is pointless to use a metric which is immeasurable or very hard to measure. As an example, in [8], one performance metric (c_2) is the overhead to maintain a thread in the thread pool. It is hard, if not impossible, to measure this variable during runtime. Therefore, to solve this problem, [8] relies on a simplified model for this variable. This will make the model applicable, but with low accuracy.
- **Low cost.** Another concern is about the overhead to monitor these variables during runtime. We do not want to introduce too much overhead for monitoring these metrics during runtime. Some techniques, such as sampling from time to time, can help to alleviate this problem.

- **Complete but not redundant.** All the metrics must cover major aspects of the thread pool system, but should not have too much overlap.

To select the metrics, let us first look at the components of the whole system, shown in Figure 3.1. According to this figure, it is clear that the whole system is composed of three major components: *submitted tasks*, *thread pool* and *operating system*. Therefore, considerations of performance metrics must reflect the requirements of these components.

From the submitted tasks' perspective, we mainly focus on the *Quality of Service* (*QoS*) in meeting their requests. We want to treat every submitted task in a fair manner and with prompt response. For operating systems, the overhead and performance of the underlying computing systems are our concerns. Finally, in the thread pool we will adjust the number of threads to reduce maintenance overhead.

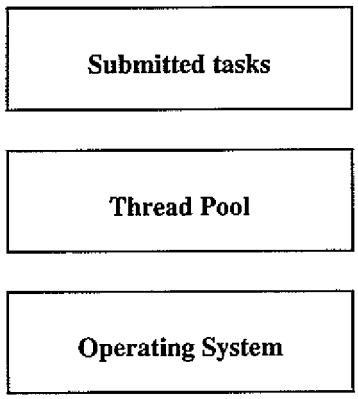

Figure 3.1. The components of thread pool system.

3.2 Selected Performance Metrics

The time flow of the submitted task is depicted in Figure 3.2. When a task is submitted to thread pool, it is first put into the queue (waiting queue) and waits for the next available

worker thread. Whenever there is a thread available, this task will be dispatched to it for execution. After the task is finished, the thread will fetch another task for execution.

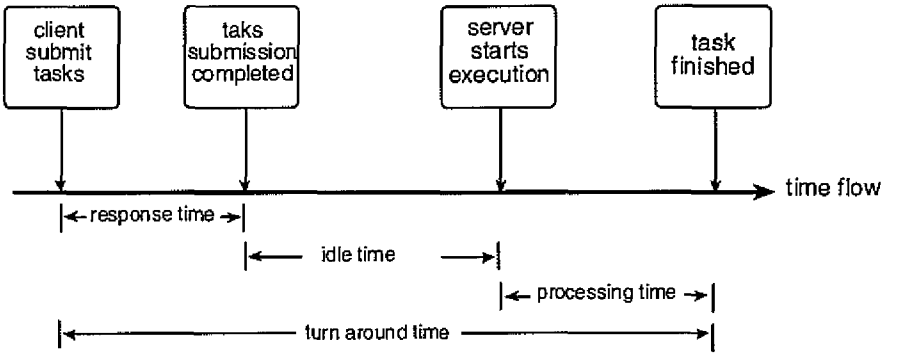

Figure 3.2. The time flow of a submitted task.

The *turnaround time* of a task is defined as the time between submission of a task and completion of the output. Turnaround time can be further divided into three components (Figure 3.2). The *response time* for task submission is the submission response latency time. For each task, the *idle time* is defined as the time spent in the waiting queue. The last one, which is more task-dependent, is *processing time*. This is the time spent for a task to be completed by the thread pool system.

The processing time is quite application dependent. Some tasks will take more time to complete while others take less. Therefore, the processing time and turnaround time cannot be used as performance metrics in a thread pool system. In our experiments, we mainly focus on the response time and the idle time of the thread pool.

There are further metrics that also turn out to be of practical interest. These metrics are more system related. These include the *CPU utilization* and the *thread pool throughput*. Throughput refers to the completed tasks per time unit (usually seconds). Note that this

variable depends on the tasks submitted because different tasks have different processing times. CPU utilization is the percentage of non-idle CPU cycles. This variable indicates whether the CPU is busy with other processing works. Notice that this metric is only important for uniprocessor system, where the thread pool is running together with submitted tasks. If a machine is dedicated to the thread pool, we might be able to skip this variable.

CHAPTER 4. THREAD POOL IMPLEMENTATION

The thread pool package is created to support the evaluation of different thread pool management schemes. Additionally, with its high-level abstraction of the programming interface, the thread pool is suitable for developing a multithreaded management system rapidly.

For a well-designed thread pool package, the following two issues must be addressed carefully. The first issue is thread pool performance. The package needs to provide fast responsiveness and high throughput as well as good performance on other metrics discussed in previous chapters. The second issue is the programming interface. A user-friendly software package should provide easy-to-use interfaces that allow programmers to create and manage the thread pool easily.

This chapter covers the features of our thread pool design. The software organization of the thread pool, including its major software modules and their relationships, is introduced first in section 4.1. Section 4.2 and section 4.3 cover the design of these software components in detail.

4.1 Architecture of the Thread Pool System

The thread pool implementation was written using POSIX C and the pthreads library to handle threading, which can be easily integrated into the existing resource management system. Basically, the software package contains five major modules, which are listed as follows. The relationship among them is shown schematically in Figure 4.1.

- Thread queue
- Task queue

- Thread scheduling
- Performance monitoring and adjustment
- Report of system performance

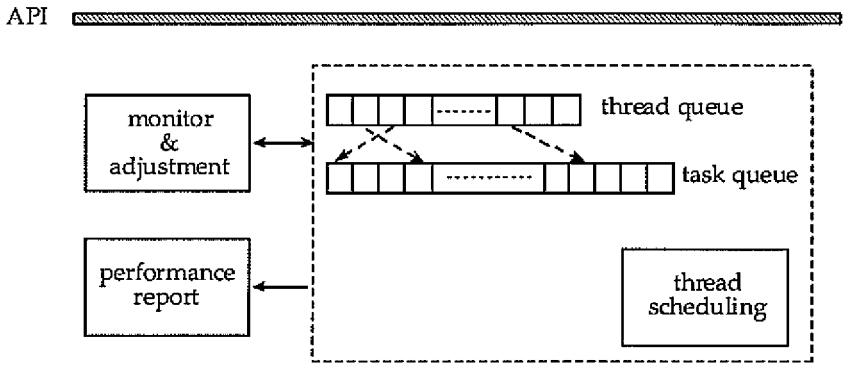

Figure 4.1. Software organization of thread pool system.

Worker threads and *tasks* are two major entities in the thread pool system. Worker threads are responsible for handling the tasks submitted by users. In the thread pool system, these two entities are maintained in two data structures, *thread queue* and *task queue*, respectively. The *thread scheduling* module will determine the detailed scheduling policy dynamically. To be able to tune the performance on-the-fly, the thread pool needs to know the current status of the whole system. This functionality is provided by the *performance monitoring and adjustment* module.

The next sections cover the detailed implementations of these modules. Specifically, section 4.2 is dedicated to the mechanisms of the worker threads and the dispatcher function. The performance monitoring and adjustment module is introduced in section 4.3.

4.2 Design of Thread Queue and Task Queue

As we can observe from Figure 4.1, thread queue and task queue are two data structures used to store the information related to the worker threads and the submitted tasks, respectively. All threads are managed in the thread queue, which is organized as an array of `pthread_t` type. The user can specify the initial number of threads by passing a parameter when the thread pool is first created (by calling `create_threadpool`). The size can be adjusted automatically by the system using the heuristic approaches which will be presented later. On the other hand, all submitted tasks are stored in the task queue. Note that the task queue is dynamic in size. Instead of destroying a node after use with the `free` function, it stores the unused nodes in a separate queue. This removes the overhead of repeated `malloc` and `free` calls.

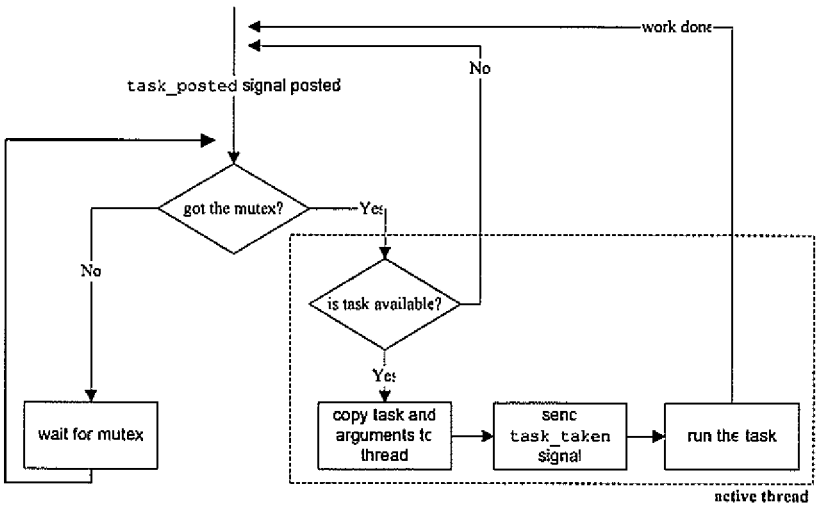

Figure 4.2. The control flows of worker threads.

The control flows of the work threads are shown in Figure 4.2. There are two modes for any thread: *busy* and *idle*. At the beginning, all threads are running in idle mode and waiting for the notification of arrivals of new tasks. Whenever new tasks become

available, a `task_posted` signal will be posted. All worker threads waiting for this signal will be notified and compete for a mutex. The winner (called the *active thread*) will get the mutex and check the pool state. If the task queue is not empty, the active thread will grab one available task in the task queue and run it.

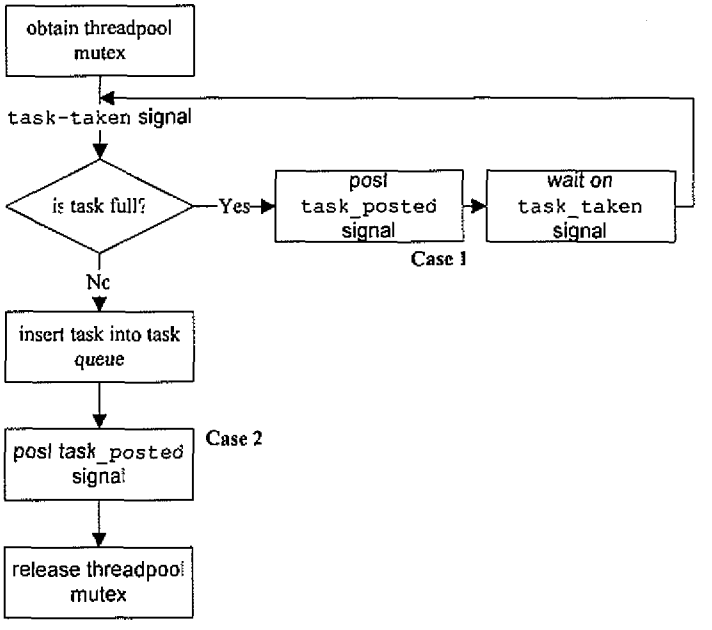

Figure 4.3. The control flow of the dispatcher function.

The thread pool should also allow users to submit tasks for execution. The functionality is provided by *task dispatcher*. Specifically, task dispatcher will put the submitted task into the task queue (shown in Figure 4.1) and notify the worker threads which are waiting for new tasks. The detailed procedure of task dispatcher is shown in Figure 4.3. Note that `task_posted` signal is posted in two places (case 1 and case 2 in Figure 4.3) for different purposes. When the task queue is full (case 1), this signal is posted to inform other waiting threads and ask them to empty the task queue for new tasks. On the other hand, the second case is to inform them of the arrivals of new tasks.

When dispatcher is called, the task to be done is placed into a queue and dispatcher returns immediately. However, the dispatch function has to be blocked when the task queue becomes full. In our current implementation, the task queue is full when it already uses 64KB of heap space. The purpose of this design is to prevent the program from crashing due to lack of heap space. During compilation, this size can be changed by modifying the parameter `MAX_QUEUE_MEMORY_SIZE` in the source code.

4.3 Performance Monitoring and Adjustment

The statistics component of our thread pool system is responsible for performance monitoring. The information collected in this component will be used for analysis and performance optimization in later stages. For users who do not need this feature, performance monitoring can be turned on/off by specifying the `-DSTATISITCS` options in the configuration file (Makefile).

As introduced in Chapter 3, the performance metrics used in this thesis can be divided into three categories: *QoS to submitted tasks*, *throughput of thread pool system*, and *OS workload*. In the rest of this section, we will briefly introduce the approaches to maintaining such information.

- **QoS to submitted tasks**

The data structure `queueNode` is used to store the information related to each task. Within `queueNode`, we define the following variables (Table 4.1) to record the information. The detailed description of each variable is also provided in this table. During runtime, these variables will be updated whenever applicable. For example, when users call the dispatcher function, the variable `submitted_t` will be updated using

current time. After one task is finished, the worker thread will call the `collectJobStatistics()` function of our statistics module to collect these variables for statistics purposes.

Table 4.1. The statistics variables used for each task.

Variables	Descriptions
`Submmited_t`	The task submission time
`accepted_t`	The task acceptance time
`exe_t`	The starting time for task execution
`finished_t`	The ending time for task execution

- **Throughput of the thread pool system**

Even though there are numerous aspects can be studied for the thread pool system, we focus on the information that can be collected by the thread pool easily. All variables related to the throughput of the thread pool, as well as the descriptions, are listed in Table 4.2. Note that the throughput of the thread pool system is defined as follows

$$throughput = \frac{\#\text{ of completed tasks}}{\text{total execution time}}$$

Table 4.2. The statistics variables used for monitoring the thread pool system.

Variables	Description
`threadNum`	The total number of threads in thread pool
`submittedJob`	The total number of tasks submitted to thread pool
`completedJob`	The total number of completed tasks
`executionTime`	The execution time of thread pool so far
`throughput`	The number of finished tasks per unit time (sec.)

- **OS overhead**

In addition to the thread pool system, there are a number of other programs running simultaneously in the same operating system. We do not want the thread pool to use

excessive system resources. The operating system workload is used as a metric to determine the current status of system performance. If the workload is too high, we might want to decrease the size of thread pool. The easiest approach to obtaining the system workload is by calling `getloadavg` function. This function averages the workload of processes in the system run queue over various periods of time. Usually three samples, which represent workload averages over the last 1, 5, and 15 minutes respectively, are provided by the this system call.

The thread pool size adjustment is provided through three functions (shown in Table 4.3). Note that these functions only provide methods to adjust the thread pool size. Determining the best pool size relies on a heuristic approach, which will be introduced in the next few chapters.

Table 4.3. The functions for thread pool size adjustment.

Functions	Descriptions
`expand_threadpool(size, pool)`	To expand the thread pool size to `size`
`shrink_threadpool(size, pool)`	To shrink the thread pool size to `size`
`threadpool_size(pool)`	Return the current size of `pool`

CHAPTER 5. EXPERIMENTAL ENVIRONMENT

5.1 The Design of Benchmark Simulator

As described in the previous chapters, the thread pool can be used by multithreaded applications to minimize thread creation and dispatch overhead due to a large volume of thread requests. A model of a server application which can serve multiple clients is schematically depicted in Figure 5.1. In this figure, all requests are depicted using a solid line, while the responses are shown with dashed lines. Whenever there is a request from a client, instead of spawning a new thread, the server will dispatch the request to the thread pool. When there is a worker thread available, the request will be served. When the task is finished, the server will send the response back to the client.

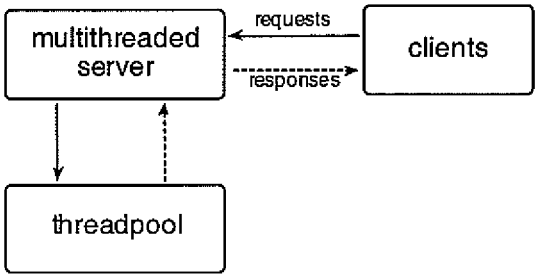

Figure 5.1. The model of a multithreaded server running on thread pool.

To measure the performance of the thread pool, we want to measure it using real-world multithreaded applications that rely on our thread pool library. Unfortunately, it is unrealistic to compile different multithreaded applications with our thread pool, because existing programs already implement their threading system using different approaches. To solve this problem, we have constructed a benchmark simulator to simulate such multithreaded applications.

The purpose of the benchmark simulator is to simulate the functionality of multithreaded servers to the thread pool. More specifically, a request simulator will read the trace data collected from real world examples and simulate its requests to the thread pool. The general architecture of this simulation system is depicted in Figure 5.2.

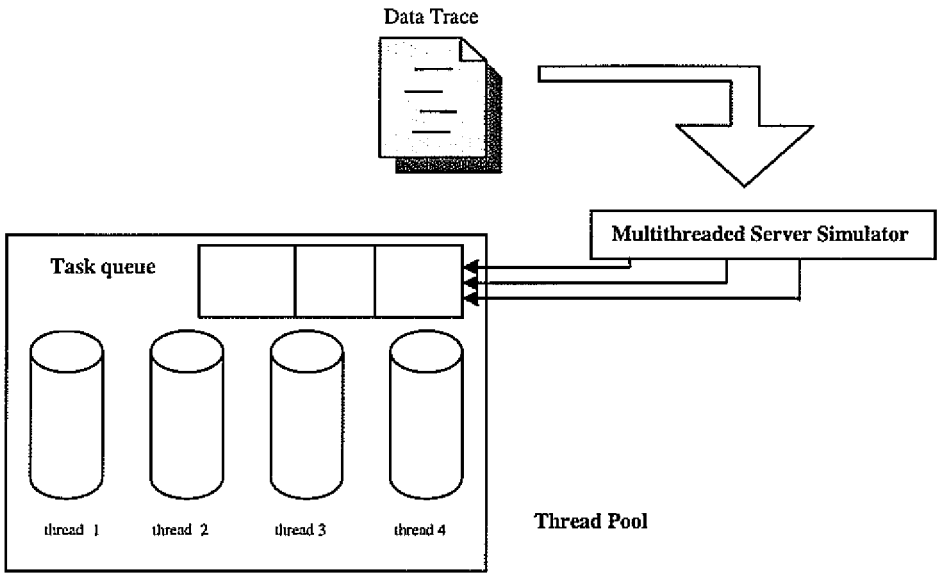

Figure 5.2. The architecture of multithreaded benchmark simulator.

Since the data trace file contains all information related to the submitted tasks, the first step of our simulator is to parse this file and collect information. Each line of the trace file represents one submitted task, and contains all related information, including *request ID*, *application ID*, *starting time*, and *task execution time*. The descriptions of these variables are provided in Table 5.1. Note that the time unit we use in the trace file is microseconds (μs).

Table 5-1. The description of the trace file format.

Variables	Descriptions
Request ID	The unique ID of submitted task
Application ID	The ID of the application the task belongs to
Starting Time	The task submission time (in relative to previous task)
Execution time	The total execution time of submitted task

A small trace file example is given in Figure 5.3, in which four tasks in total are submitted from two applications. In this example, at the beginning (starting time = 0) two tasks are submitted from two applications with execution times of 200μs and 150μs, respectively. After that, there is no task submitted for 300μs. At 300μs, both application 1 and application 2 submit one task with the same execution time (100μs) again.

Request ID	**Application ID**	**Starting Time (s)**	**Execution Time (e)**
1	1	0	200
2	2	0	150
3	1	300	100
4	2	0	100

Figure 5.3. A small trace file example.

Based on the trace file format introduced above, the design of a benchmark simulator becomes straightforward. For each task request, the simulator checks the starting time (s) first. If the starting time is larger than 0, the simulator will stop fetching new task requests (by sleeping for s μs). Otherwise, the simulator will call the dispatch function and ask one thread to execute the task for e μs. The procedure will be repeated until all requests are completed.

At the beginning of implementation, we forced each task to sleep e μs. Later, we realized that this does not reflect the behavior of real world tasks. In reality, a portion of

the time of a submitted task should be spent on other computations, while the rest is in waiting mode because of I/O or other data dependencies. To solve this problem, we changed our design and defined a new variable, called `free_workload`, in our benchmark. The meaning of this variable is shown in Figure 5.4.

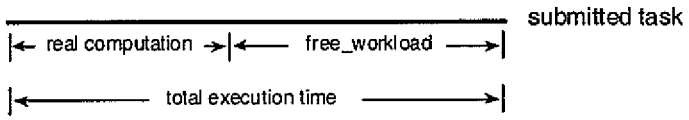

Figure 5.4. The description of `free_workload`.

This variable is used to adjust the frequency of sleep for each submitted task. The larger `free_workload` is, the more time spent on sleep. For example, if the `free_workload`=100, the benchmark will force the task to sleep for 100 times of `e` μs. By adjusting this value, we can emulate different types of multithreaded applications using our simulator. Notice that the real computation time is fixed for each submitted task[1].

5.2 Experimental Environment

The operating system is RedHat Linux 9 running on a single 1GHZ Intel Pentium III processor. The CPU has 32KB (16KB D-Cache/16KB I-Cache) L1 cache and 256KB unified L2 cache. The physical memory size is 512 MB. To limit the number of running processes, the machine is booted using text mode only. All simulations are repeated three times, and the best value is used for performance analysis. The network connecting the test machines is a switched Fast Ethernet running at a maximum of 100 Mbps. The

[1] Actually, the computation time is proportional to `e` in our implementation. Since `e` is fixed when the benchmark reads each task, we consider the computation time fixed compared with the idle time which is also controlled by `free_workload`.

network interface we used is localhost. All network-related tests were conducted in an isolated environment in order to minimize the effects of other traffic.

CHAPTER 6. EXPERIMENTAL RESULTS

6.1 Performance of Thread Pool System

In this section, we focus on the performance improvement brought to multithreaded applications through the use of a thread pool.

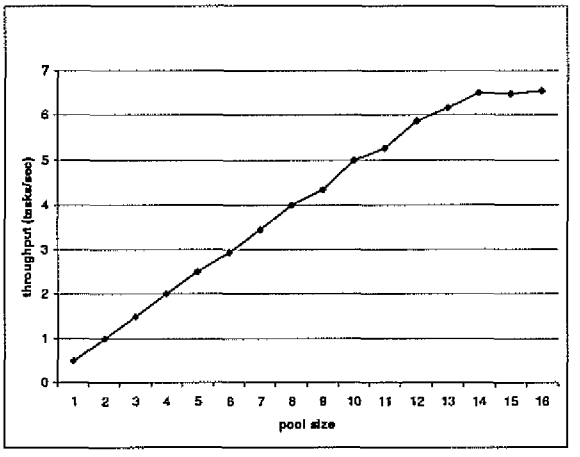

Figure 6.1. Throughput vs. pool size (`free_workload=100`).

Figure 6-1 shows the relationship between the throughput and the thread pool size. From this figure, it is obvious that the throughput of multithreaded applications can be improved proportional to the pool size when the pool size is relatively small. Unfortunately, such improvement cannot be sustained when the pool size is greater than a *threshold* (t = 14 in this example). Actually, the throughput of our benchmark is about the same when the pool size is 14, 15 and 16. We suspect this phenomenon is caused by two issues. First, the application can only benefit from using a limited number of threads. When the pool size passes this threshold, the capacity of the application to utilize available threads becomes saturated and no performance improvement can be obtained. For instance, consider a multithreaded program which only uses six threads during

runtime. A thread pool with size 6 will provide the best throughput for the program. The performance will not improve when the pool size is greater than 6. Second, the maintenance overhead brought by increasing pool size might overshadow the benefits obtained by using more threads.

To verify our hypothesis, we have performed similar experiments for two different benchmarks. These benchmarks are obtained by varying the parameter `free_workload`. Two values, 50 and 10, were picked for our experiments. The experimental results are shown in Figure 6.2 and Figure 6.3, respectively.

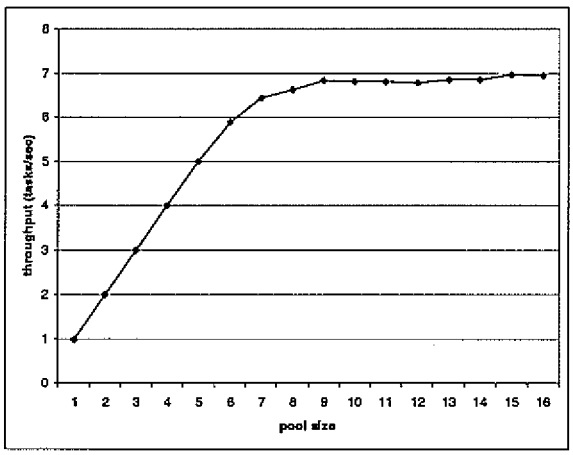

Figure 6.2. Throughput vs. pool size (`free_workload=50`).

Figure 6.2 shows that the throughput is still proportional the pool size at the beginning. However, compared with Figure 6.1, the threshold which can sustain the linear improvement is smaller (t = 6 in this case). This is related to the characteristics of the multithreaded applications used in our experiments. In the first experiment, the real computation workload is lower (`free_workload` = 100). According to Chapter 5, it means the application spends a large portion of execution time on I/O. Therefore, the throughput will become higher because the OS has more chances to schedule other active

threads for running. In contrast, as the `free_workload` decreases to 50, the application becomes more computation intensive. Under this circumstance, OS will be bound to a small number of threads and the throughput will lower. Similar results can be observed when the `free_workload` becomes 10 (Figure 6.3).

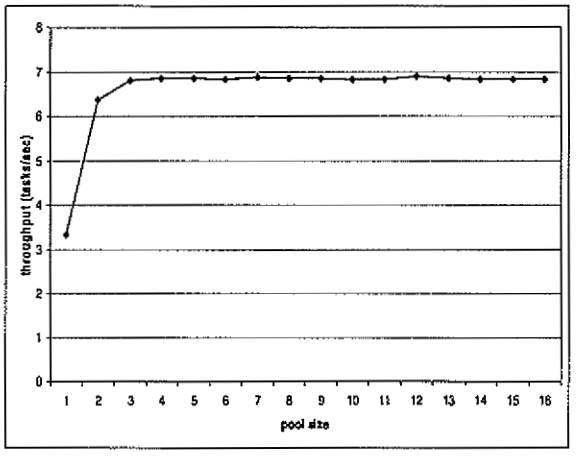

Figure 6.3. Throughput vs. pool size (`free_workload=10`).

Two conclusions can be drawn from the above experiments. First, using a thread pool can help to improve the performance (throughput) of multithreaded applications. Second, the degree of improvement is application-dependent and work load dependant. For computation intensive applications, the benefits of using a thread pool can be smaller. This also demonstrates the need to be able to dynamically resize the pool size for different types of applications, which will be discussed in Section 6.3.

6.2 Internal Characterizations of Thread Pool System

The experiments above show us the impact of a thread pool system on multithreaded applications. The results imply the need to adjust the pool size for different types of applications. To be able to adjust the pool size on the fly, we need to further understand

the internal characteristics of the thread pool system. In this section, the performance metrics of the thread pool presented before are studied in depth. In particular, we want to correlate the metrics which are application independent to the throughput. Such information will allow us to adjust the pool size dynamically.

6.2.1 Average Job Idle Time

To study the relationship between the throughput and the thread pool size, we have picked an internal performance metric, the *average idle time* (*AIT*). The detailed description of idle time is presented in section 3.2. The AIT is much easier to measure inside the thread pool and is independent from the behavior of the tasks. If the results show that the AIT correlates to the throughput, we might be able to use this metric for dynamic pool size adjustment.

As in previous experiments, we pick two values for the `free_workload`[2], 100 and 50, for this study. The experimental results are shown in Figure 6.4 and Figure 6.5, respectively. To compare with throughput easily, we use the *reciprocal of average idle time* (*RAIT*) in our experiments. RAIT is defined as $RAIT_i = \frac{AIT_1}{AIT_i}$, where AIT_i is the average idle time when pool size is `i`. Therefore, AIT_1 is the average idle time of pool size 1. Instead of using 1 as numerator, we use AIT_1 such that RAIT always starts from 1 when the pool size is 1.

[2] The results of `free_workload=1` are not shown here. However, it gives similar results to other sizes.

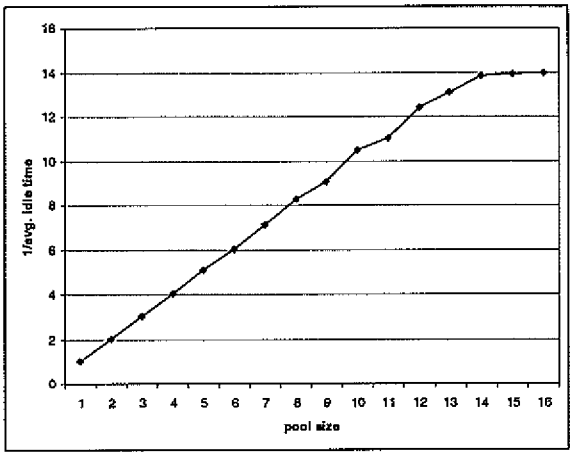

Figure 6.4. The RAIT vs. thread pool size (`free_workload=100`).

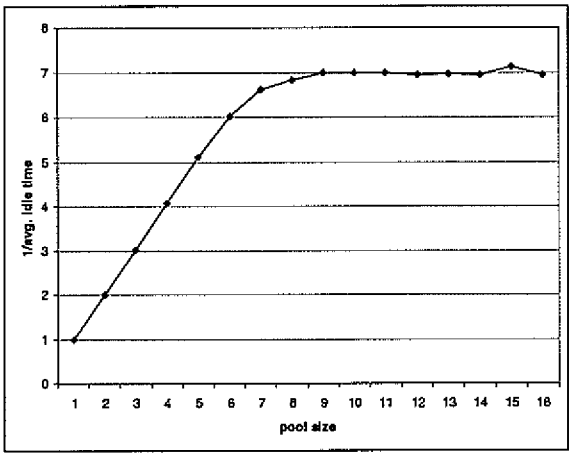

Figure 6.5. The RAIT vs. thread pool size (`free_workload=50`).

By comparing Figure 6.4 with Figure 6.1, it is clear that the average idle time (AIT) of tasks has a strong correlation to system throughput. A similar pattern is also observed by comparing Figure 6.5 and Figure 6.2. These results indicate the possibility of using AIT to infer the potential throughput of multithreaded programs. We can use such information to adjust the thread pool size on the fly.

6.2.2 Overhead of Thread Pool Management

The most attractive benefit of using a thread pool is to avoid the overhead of thread creation. However, that does not mean users should create a thread pool as large as possible. Indeed, the overhead for managing threads in the pool can be a big issue. To examine this problem, we will study the thread pool management overhead in this section.

The most straightforward way of studying the management overhead is to increase the pool size. According to the previous discussion, the performance improvements brought by thread pool will be saturated when the pool size reaches a threshold. After that, increasing the pool size will not help to improve the performance further. Instead, the overhead of thread pool management will degrade the performance (throughput) when the size becomes larger. Therefore, by increasing the pool size, we should be able to observe the impact brought by the overhead.

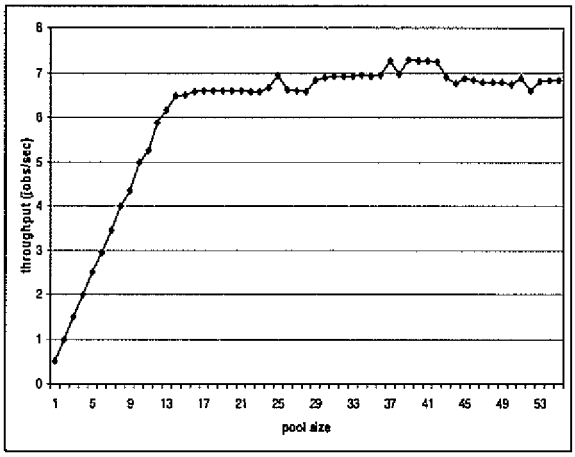

Figure 6.6. The thread pool management overhead (`free_workload`=100).

Following this idea, we have measured the throughput of our benchmark by increasing the pool size from 1 to 55. The results are presented in Figure 6.6. According to this figure, the throughput becomes stable when the pool size reaches 13. After that the throughput

mostly fluctuates around a fixed value. (The best throughput is observed when the pool size reaches 38. After that, it begins to drop gradually.)

The observed behavior is consistent with our expectations. Figure 6.7 shows the expected behavior of thread pool when the pool size increases. The point where the throughput becomes stable is called *stable point*. Beyond this point, the throughput will maintain a relatively steady value. When the size is greater than another threshold, called the *degradation point*, the overhead of pool management will become dominant and offset the benefits brought by using thread pool. The performance will drop after this point. In the previous examples, the stable point is 12 and the degradation point is 40. These two points might change for other applications and other workloads.

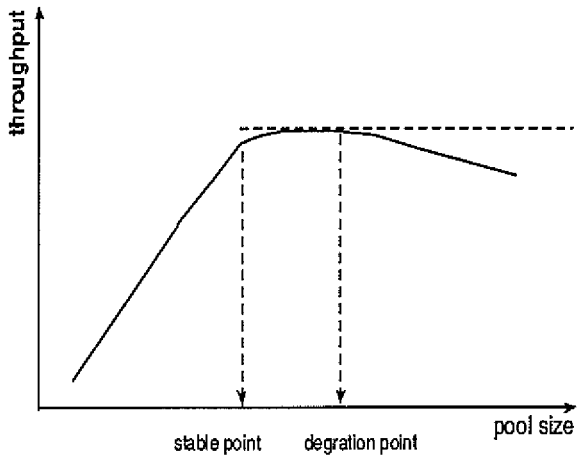

Figure 6.7. The relationship between throughput and the thread pool size.

The design of a dynamic thread pool needs to be able to adjust the pool size as quickly as possible to the *safe zone*, which is defined to be the area between the stable point and the degradation point. This is the area where the throughput will reach a maximum without introducing too much overhead.

6.3 Dynamic Pool Size Optimization

Based on the above observations, we propose to use the task idle time as a criterion for determining the optimal thread pool size. In this section, we first describe the design of our algorithm. We verify our algorithm by implementing it in our current thread pool system. The performance results of this algorithm are also presented in this section.

6.3.1 Algorithm for Dynamic Pool Size Adjustment

```
Algorithm dynamicThreadPool
Input:  stride,
        preAIT,
        prepreAIT
        poolSize;
BEGIN
  Store current average idle time in currentAIT;

  if((|currentAIT - preAIT|/preAIT) > 1%) {
    if(currentAIT > preAIT) {
      if(preAIT < prepreAIT)
        poolSize -= stride;
      else
        poolSize += stride;
    }
    else if (currentAIT < preAIT && preAIT < prepreAIT) {
      poolSize += stride;
    }
  }
  else if(prePoolSize == poolSize)
    poolSize += stride;

  if(poolSize <= 0)
    poolSize = 1;

  adjustPoolSize(poolSize);
  prepreAIT = preAIT;
  preAIT = currentAIT;
  prePoolSize = poolSize;
END
```

Figure 6.8. The algorithm for dynamic thread pool size adjustment.

We have developed a new algorithm, named `dynamicThreadPool`, for thread pool size adjustment. The pseudocode of this algorithm is presented in Figure 6.8. Instead of comparing absolute values, this algorithm checks the percentage of difference between the current AIT and the previous AIT. If the difference is larger than 1%, the pool size is increased or decreased depending on the relationship between other variables. This algorithm is proactive in increasing pool size. By comparing the current average idle time with the previous one, the pool size will be increased by a fixed number (`stride`) whenever appropriate. Decreasing the pool size only happens when the algorithm finds that the previous increase in pool size caused performance degradation.

Three variables are used in this algorithm. The purpose of both `preAIT` and `prepreAIT` is to record the average idle time in the past two cycles. These values will be propagated to each other at the end of each cycle. `stride` determines the degree of decrease and increase of thread pool size. Notice that the initial value of `stride` will affect the runtime performance. In the following experiments, we set the initial value of stride to be 2.

6.3.2 Performance of the Algorithm

First, we want to examine the behavior of our algorithm when it is used in real applications. To do that, we implemented the `dynamicThreadPool` algorithm in our thread pool system. This algorithm is executed at the end of each cycle, which is defined as five completed jobs in our experiments. For thread pools with different initial pool sizes, the behavior of this algorithm might be different. Therefore, we have chosen two initial thread pool sizes, 4 and 16, for the experiments. The results are shown in Figure 6-7. The

experimental results show that, for both initial thread pool sizes, the algorithm continuously increases the thread pool size towards the safe zone. The pool size becomes stable around this area.

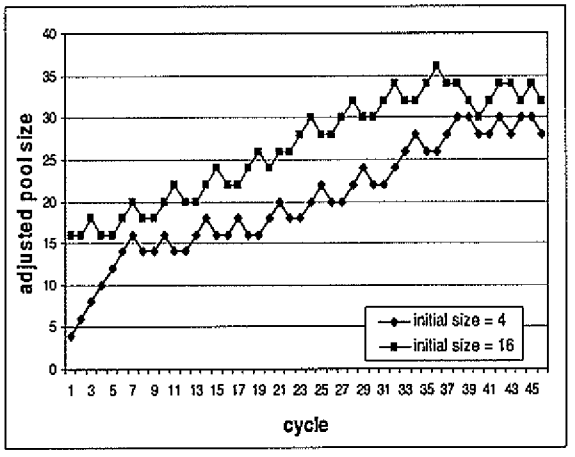

Figure 6.8. The adjusted thread pool size for two different initial thread pool sizes.

Note that the maximal adjusted pool size is not constant when it reaches the stable area. Instead, it fluctuates around some fixed value. Apparently, in a perfect environment, this size is supposed to be the same. However, on real machines, the AIT we obtain might vary due to other factors (such as OS workloads and job behaviors). This will affect the accuracy of AIT and our algorithm. Therefore, fluctuation around some values is acceptable for our algorithm.

Our second experiment is to compare the throughputs of the original thread pool (which is called the *static thread pool*) and the dynamic thread pool. The dynamic thread pool is designed to adjust the pool size according to the behavior of multithreaded applications. The ultimate goal is to achieve better performance without introducing too much overhead. Therefore comparing throughput will help to understand the performance improvement brought by using the dynamic thread pool. The experimental results are

shown in Figure 6.8. To compare the performance more clearly, the throughput of the dynamic thread pool is normalized to the throughput of static thread pool.

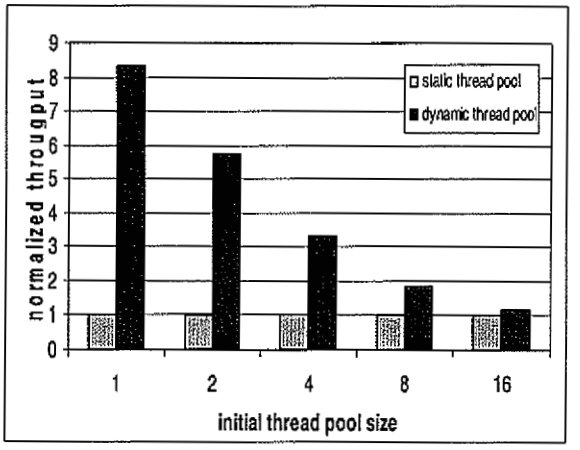

Figure 6.9. The throughput improvement by using dynamic thread pool.

The experimental results clearly show the performance improvement brought about by using a dynamic thread pool. Actually, for a thread pool with initial thread number =1, the throughput of the dynamic thread pool is about 8 times that of the static one. For other initial pool sizes, similar improvements are also observed. Interestingly, the improvement drops gradually when the initial thread number increases. This is because the performance of static thread pool is already close to optimal when the initial pool size is large.

Scalability of thread pools is one important issue to consider. There are two aspects of the scalability issue: the capability to manage many threads simultaneously and the capability to handle a large amount of incoming requests. Since we used a large amount of requests in our experiments, the second issue has been resolved. According to Figures 6.8 and 6.9, our system can maintain good performance improvement even when the pool size is increased to a fairly large number (35 in Figure 6.8). This shows this system has a decent scalability. In our experiments, we also found that the performance of our

computing system drops when the pool size increases. We suspect this problem was caused by the thread management module of our Linux kernel. The problem is expected to be alleviated after Linux is updated to a new pthread library, *Native POSIX Thread Library (NTPL)*. This new library can potentially boost the thread performance dramatically, especially when the thread number is large.

In future applications, it is possible that the number of threads needed will become enormous. How will the method we have developed here scale in such cases? We believe that our system will perform well within the constraint of system capability. The reason is that, based on average idle time, our dynamic adjustment scheme already takes the underlying system limitations into consideration. Such dynamic optimization design should be able to adjust the pool size to an optimal value on-the-fly.

CHAPTER 7. CONCLUSIONS AND FUTURE WORK

Traditional client-server programming incurs significant overhead in spawning a new thread for each incoming request. A thread pool solves this problem by maintaining a pool of pre-created threads and dispatching new tasks to idle threads. Without thread creation overhead, thread pools can deliver better throughput and response time than other approaches. Because of these benefits, thread pools have been widely used in many multithreaded applications.

The performance of the thread pool is determined by the pool size. For different applications, the optimal pool size will be different. The determination of the pool size according to the application behavior is a difficult problem. Server applications usually have to rely on system administrators to tune the thread pool performance based on their experience. To automate this process, in this paper we have developed a set of performance metrics for quantitatively analyzing the thread pool performance. For our experiments, we built a thread pool system which provides a general framework for thread pool research. Based on this simulation environment, we have studied the performance impact brought by the thread pool to different multithreaded applications. Additionally, the correlations between internal characterizations and the throughput were also studied.

Based on our experiments, we observed that the average task idle time has strong correlation with the thread pool throughput. We proposed and evaluated the idea of using a heuristic approach to determine the optimal thread pool size based on the task average idle time. This approach makes a tradeoff between the thread pool performance and the management overhead. Our approach differs than previous research where the thread pool

models are overly complex. The simulation results show that dynamic optimization for thread pool size is very effective in alleviating the management overhead and improving the overall performance. The results imply the potential benefits of using dynamic optimization to replace manual configuration in large multithreaded server applications.

A number of questions arising from this work will be addressed in future work.

- Provide more general thread pool functionality. Some users might have special requirements, such as the submission of jobs that can be executed in a fixed time repeatedly.
- Polish the application programming interfaces (APIs) and maintain a simple, easy-to-use set of APIs.
- Enhance the current statistics mechanism. In addition to the current set of metrics, more performance metrics will be included. These metrics will help to further improve thread pool system performance.

REFERENCES

[1] Apache Software Foundation. *Apache HTTP Server Documentation*. Available from http://httpd.apache.org/docs-project/ as of May 21, 2004.

[2] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of ACM*, 46(5):720-748, 1999.

[3] John Calcote. Thread pools and server performance. *Dr. Dobb's Journal*, 60-64, July 1997.

[4] Earth Simulator Center. *Earth Simulator System Configuration*. Available from http://www.es.jamstec.go.jp/esc/eng/Hardware/system.html as of May 21, 2004.

[5] Al Geist et al. *Scalable System Software Enabling Technology Center*. Available from http://www.scidac.org/ScalableSystems/proposal.doc as of May 21, 2004.

[6] Judith Hippold and Gudula Runger. Task pool teams for implementing irregular algorithms on clusters of SMPs. In *Proceedings of International Parallel and Distributed Processing Symposium* (*IPDPS 2003*), 22-26, Nice, France, April 2003.

[7] IBM Corporation. RS/6000 SP System Management: Easy, Lean and Mean. Technical Report, International Technical Support Organization, June 1995.

[8] Yibei Ling, Tracy Mullen and Xiaola Lin. Analysis of optimal thread pool size. *ACM SIGOPS Operating System Review*, 34(2):42-55, 2000.

[9] Microsoft Cooperation. *Maximizing IIS Performance*. Available from http://www.microsoft.com/technet/prodtechnol/windows2000serv/technologies/iis/maintain/optimize/perflink.mspx as of May 21, 2004.

[10] Mike Moore. Tuning Internet Information Server Performance. Available from http://www.microsoft.com/serviceproviders/whitepapers/tuningiis.asp as of 04/28/2004.

[11] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming.* O'Reilly & Associates, Sebastopol, CA, 1996.

[12] Platform Computing. *LSF (Load Sharing Facility).* Available from http://www.platform.com/platform/platform.nsf/webpage/LSF as of May 21, 2004.

[13] J. Pruyne and M. Livny. Interfacing Condor and PVM to Harness the Cycles of Workstation Clusters. *Journal on Future Generations of Computer Systems*, Vol. 12, 1996.

[14] Irfan Pyarali, Marina Spivak, Ron Cytron and Douglas C. Schmidt. Evaluating and Optimizing Thread Pool Strategies for Real-Time CORBA. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES)*, 214-222, Snow Bird, Utah, 2000.

[15] Douglas C. Schmidt. Evaluating Architecture for Multithreaded Object Request Brokers. *Communication of ACM.* 41(10):54-60, October 1998.

[16] Veridian Systems. *Portable Batch System (PBS).* Available from http://www.OpenPBS.org as of May 21, 2004.

[17] Andrew S. Tanenbaum. *Modern Operating Systems* (2^{nd} *edition*). Prentice Hall, February, 2001.