

# Performance Study and Dynamic Optimization Design for Thread Pool Systems

Dongping Xu  
Department of Electrical and Computer Engineering  
Iowa State University  
Ames, IA 50011

and

Brett Bode  
Scalable Computing Laboratory  
Ames Laboratory  
Iowa State University  
Ames, IA 50011

## 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 paper 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.

**Keywords:** Thread Pool, Analysis, Characterization, Implementation, and Dynamic Optimization.

## 1. INTRODUCTION

While multithreading provides a clean design approach to handling asynchronous requests, the architecture used to implement multithreading still can have a large impact on the computational thread-creation overhead. Two models, including *thread-per-request* and *thread pool*, are widely used in multithreaded programming for server applications. The thread-per-request model spawns a thread for each request, and destroys the thread after finishing the request. In contrast, a thread pool system spawns and maintains a pool of threads. When a request arrives, the application 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 a thread pool model can significantly improve system performance and reduce response time [3][11][10]. Because of its benefits, thread pool systems have been adopted by a large number of popular server applications, such as Apache [1] and Windows IIS [7].

The performances of these server applications rely in part on the throughput which can be delivered by the thread pool. A 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. As the pool size increases, however the overhead of thread pool management will become significant and degrade the system performance eventually. Therefore, handling this tradeoff becomes important for system optimization.

To solve this problem, people have proposed different approaches to tune the thread pool system. The most widely used approach is *based on experience*. In this approach, the system administrators are required to constantly monitor the system performance. Whenever a performance bottleneck is noticed, they will tune the configuration to optimize the performance. This approach is widely adopted by many server applications, including Apache [1] and Microsoft IIS [7], both of which are the most widely-used Web servers. As an example, Apache allows users to control the number of threads deployed by each child process by changing the variable `ThreadsPerChild`. Clearly, such experience-based approaches have serious drawbacks. The performance monitoring job is very time-consuming and inconvenient for a system administrator. In addition, the configuration drawn up through this approach is often inaccurate, especially when the performance varies a lot over time. To solve this problem, it is desirable to have a thread pool which can configure itself based on the current status of the server system.

Some researchers have proposed schemes to predict the optimal thread pool size based on heuristic factors [6]. Unfortunately, the formula usually is very complicated. It is hard, if not impossible, to apply those formulas in practice because of the complexity and overhead. To solve the problems above, we want to construct a new dynamic optimization approach which is simpler to avoid a 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.

In this paper we will develop a set of performance metrics for quantitatively analyzing the thread pool performance. Based on these metrics, we will systematically study the internal characterizations of a thread pool system. Additionally, we will 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.

This paper is organized as follows. In Section 2, we present the metrics used in our system for performance evaluation. The detailed implementations of our thread pool and multithreaded benchmarks are presented in Section 3 and 4, respectively. The experimental results are discussed in Section 5. Finally, Section 6 concludes this paper.

## 2. PERFORMANCE METRICS

Instead of the experience-based approach, we are searching for a quantitative approach which allows us to predict the optimal thread pool size without interference from humans. The performance of our approach relies on a carefully selected set of performance metrics, which must meet the following criteria.

- Measurable
- Low cost
- Complete but not redundant

Since the whole system is composed of three major components: *submitted tasks*, *the thread pool* and *the operating system*. The 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.

The time flow of the submitted task is depicted in Figure 1. 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 1). 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.

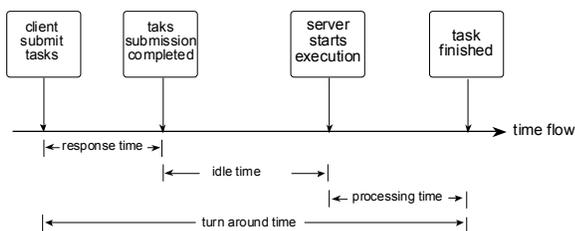


Figure 1. The time flow of a submitted task.

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, including *thread pool throughput*. Throughput refers to the completed tasks per time unit (usually seconds).

## 3. IMPLEMENTATION OF THREAD POOL

### 3.1 Architecture of the Thread Pool System

Our thread pool implementation was written using POSIX C using the Pthreads library to handle threading, which can be easily integrated into the existing applications written in C/C++ [9]. Basically, the software package contains five major modules, which are listed as follows. The relationship among them is shown schematically in Figure 2.

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

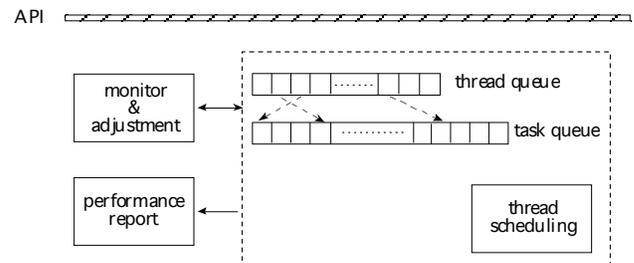


Figure 2. Software organization of thread pool system.

### 3.2 Design of the Thread Queue and Task Queue

*Worker threads* and *tasks* are two major entities in the thread pool system. As we can observe from Figure 2, 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 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.

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.

The thread pool should also allow users to submit tasks for execution. The functionality is provided by *task dispatcher*. Specifically, the task dispatcher will put the submitted task into the task queue (shown in Figure 2) and notify the worker threads which are waiting for new tasks. When the task to be done is placed into a queue, the dispatcher function returns immediately. However, the dispatch function has to be blocked when the task queue becomes full.

### 3.3 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.

### 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 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.

Table 1. The statistics variables used for each task.

Variables	Descriptions
Submitted_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 that can be studied for the thread pool system, we have focused 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 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 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.)

## 4. EXPERIMENTAL ENVIRONMENT

### 4.1 Design of the Benchmark Simulator

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 3.

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*. Our simulation will operate according to the data of input trace files.

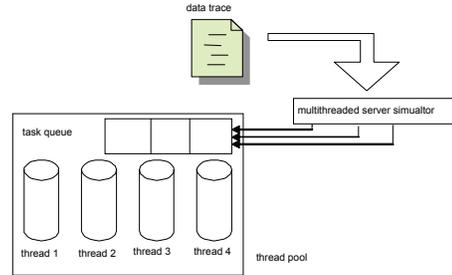


Figure 3. The design of multithreaded simulator

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 reflect the behavior of real world tasks, we defined a new variable, called `free_workload`, in our benchmark. The meaning of this variable is shown in Figure 4.

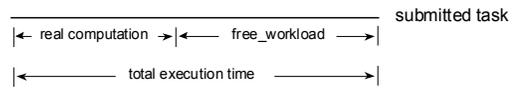


Figure 4. The description of `free_workload`.

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  $\epsilon \mu s$ . 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.

## 4.2 Experimental Configurations

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

## 5. RESULTS

### 5.1 Performance of Thread Pool System

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

Figure 5 shows the relationship between the throughput and the thread pool size. First, we choose `free_workload=100`. 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*. 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. Second, the maintenance overhead brought by increasing pool size might overshadow the benefits obtained by using more threads.

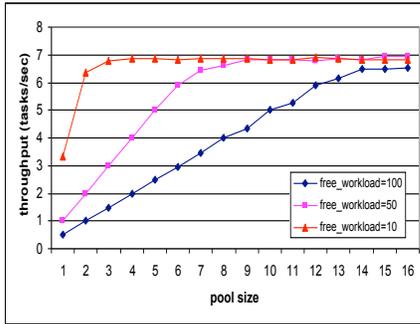


Figure 5. Throughput vs. pool size

To verify our hypothesis, we have performed similar experiments for two different benchmarks. These benchmarks are obtained by varying the parameter `free_workload` (50 and 10 for our experiments). The experimental results are also shown in Figure 5. The results show that the throughput is still proportional to the pool size at the beginning. However, compared with `free_workload=100`, the threshold which can sustain the linear improvement is smaller. 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 Figure 4, 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 and 10, the application becomes more computation intensive. Under this circumstance, OS will be bound to a small number of threads and the throughput will lower.

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 this section.

## 5.2 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.

## 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 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`<sup>1</sup>, 100 and 50, for this study. The experimental results are shown in Figure 6. 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.

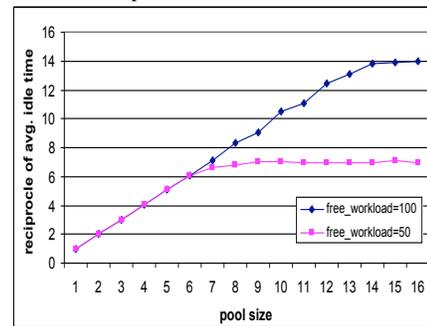


Figure 6. The RAIT vs. thread pool size

By comparing Figure 6 with Figure 5, it is clear that the average idle time (AIT) of tasks has a strong correlation to system throughput. 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.

## 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 study the thread pool management overhead in our experiments.

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 a 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.

<sup>1</sup> The results of `free_workload=1` are not shown here. However, it gives similar results to other sizes.

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 7. 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.)

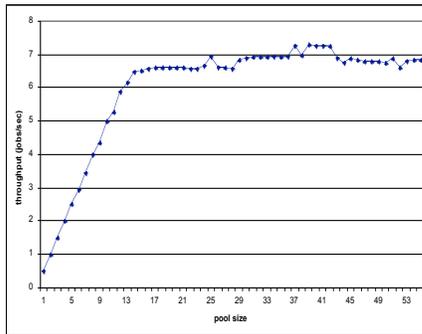


Figure 7. The thread pool management overhead ( $free\_workload=100$ ).

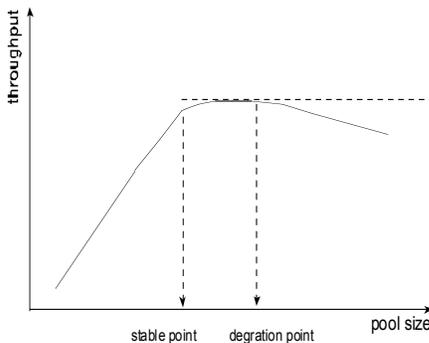


Figure 8. The relationship between throughput and the thread pool size.

The observed behavior is consistent with our expectations. Figure 8 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 the 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.

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.

### 5.3 Dynamic Pool Size Adjustment Algorithm

We have developed a new algorithm, named `dynamicThreadPool`, for thread pool size adjustment. The pseudocode of this algorithm is presented in Figure 9. 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.

#### 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 9. The pseudocode of dynamic thread adjustment.

### Performance of Dynamic Pool Size Adjustment

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 10. 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.

Note that the maximal adjusted pool size is not constant when it reaches the stable area. Instead, it fluctuates around some fixed

value. In a perfect environment, this size is would 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, some fluctuation is acceptable for our algorithm.

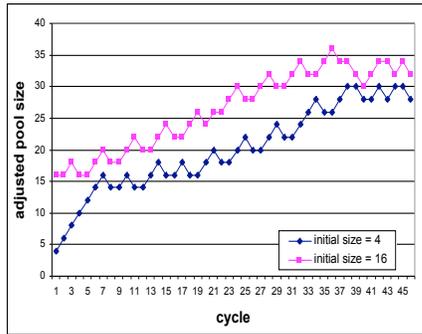


Figure 10. The adjusted thread pool size for two different initial thread pool sizes.

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 11. To compare the performance more clearly, the throughput of the dynamic thread pool is normalized to the throughput of static thread pool.

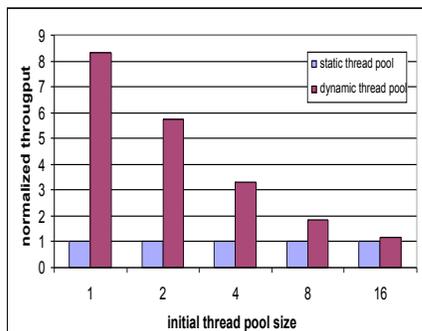


Figure 11. The throughput improvement by using dynamic thread pool.

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 closer to optimal when the initial pool size is large.

## 6. CONCLUSIONS

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.

The experimental results indicate 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. The simulation results show that dynamic optimization for thread pool size is very effective in alleviating the management overhead and improving the overall performance.

## 7. ACKNOWLEDGEMENTS

This work was supported in part by U. S. Department of Energy Scientific Discovery through Advanced Computing (SciDAC) project [4]. Authors want to thank Dr. Daniel Berleant and Dr. Shashi Gadia for their insightful comments on the manuscript. This manuscript has been authored by Iowa State University of Science and Technology under Contract No. W-7405-ENG-82 with the U.S. Department of Energy.

## 8. REFERENCES

- [1] Apache Software Foundation, **Apache HTTP Server Documentation**, Available from <http://httpd.apache.org/docs-project/>.
- [2] Robert D. Blumofe and Charles E. Leiserson, "Scheduling Multithreaded Computations by Work Stealing", **Journal of ACM**, Vol. 46, No. 5, 1999, pp. 720-748.
- [3] John Calcote, "Thread Pools and Server Performance", **Dr. Dobb's Journal**, July 1997, pp. 60-64.
- [4] Al Geist et al, **Scalable System Software Enabling Technology Center**, Available from <http://www.scidac.org/ScalableSystems/proposal.doc>.
- [5] 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)**, Nice, France, April 2003, pp. 22-26.
- [6] Yibei Ling, Tracy Mullen and Xiaola Lin, "Analysis of Optimal Thread Pool Size", **ACM SIGOPS Operating System Review**, Vol. 34, No. 2, 2000, pp. 42-55.
- [7] Microsoft Cooperation, **Maximizing IIS Performance**, Available from <http://www.microsoft.com/technet/prodtechnol/windows2000serv/technologies/iis/maintain/optimize/perflink.mspix>.
- [8] Mike Moore, **Tuning Internet Information Server Performance**, Available from <http://www.microsoft.com/serviceproviders/whitepapers/tuningiis.asp>.
- [9] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell, **Pthreads Programming**, O'Reilly & Associates, Sebastopol, CA, 1996.
- [10] 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)**, Snow Bird, Utah, 2000, pp. 214-222.
- [11] Douglas C. Schmidt, "Evaluating Architecture for Multithreaded Object Request Brokers", **Communication of ACM**, Vol. 4, No. 1, October 1998, pp. 54-60.