# Analysis of Optimal Thread Pool Size

[1]Yibei Ling, [1]Tracy Mullen, and [2]Xiaola Lin
[1]Telcordia Technologies
445 South Street, Morristown, NJ 07032
{lingy,mullen}@research.telcordia.com
[2]Department of Electrical Engineering
Hong Kong University
xlin@eee.hku.hk

Monday, February 14, 2000

*Abstract*

*The success of e-commerce, messaging middleware, and other Internet-based applications depends in part on the ability of network servers to respond in a timely and reliable manner to simultaneous service requests. Multithreaded systems, due to their efficient use of system resources and the popularity of shared-memory multi-processor architectures, have become the server implementation of choice. However, creating and destroying a thread is far from free, requiring run-time memory allocation and deallocation. These overheads become especially onerous during periods of high load and can be a major factor behind system slowdowns. A thread-pool architecture addresses this problem by prespawning and then managing a pool of threads. Threads in the pool are reused, so that thread creation and destruction overheads are incurred only once per thread, and not once per request. However, efficient thread management for a given system load highly depends on the thread pool size, which is currently determined heuristically. In this paper, we characterize several system resource costs associated with thread pool size. If the thread pool is too large, and threads go unused, then processing and memory resources are wasted maintaining the thread pool. If the thread pool is too small, then additional threads must be created and destroyed on the fly to handle new requests. We analytically determine the optimal thread pool size to maximize the expected gain of using a thread.*

## 1 Introduction

As the Internet continues to expand, web servers receive up to several million hits per day [Mim99]. To handle this demand, high performance servers exploit caching, prefetching, and multiprocessor technologies. Multithreading techniques are another solution to help meet the ever-increasing performance demands. In a multithreaded system, the processor switches between threads so that each request can execute simultaneously and independently. Multithreading improves processor utilization in several ways. For multiprocessors, such as shared memory multiple processors (SMP), multiple threads can be farmed out to different processors to easily distribute the processing load. For long latency operations, where a thread can stall due to slow memory or network I/O, multithreading minimizes overall latency by overlapping processing time between the remaining threads.

While multithreading provides a clean design approach to handling asynchronous requests, the architecture used to implement multithreading can have a large impact on the computational thread-creation overhead. Multithreading architectures range from thread-per-request to thread-pool architectures [Sch98]. The thread-per-request architecture spawns a thread for each client request, then destroys the thread after finishing the request. Thread-pool architectures spawn and maintain a pool of threads. When a request arrives, the server 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 architectures can significantly improve system performance and reduce response time [Age99,Cal97,Sch98,Sun95].

The paper is organized as follows: Section 2 provides an overview of multithreading models and discusses how thread pools support efficient system resource utilization. Section 3 presents a mathematical model characterizing these system costs and derives the optimal thread pool size by maximizing its expected gain. Section 4 illustrates how to find the optimal thread pool size when the probability distribution of requests is given. We present a practical algorithm that determines the request probability distribution from information contained in server logs, as well as an algorithm to derive the optimal thread pool size based on system performance characteristics and the derived probability distribution. In Section 5, we discuss our conclusions and future research directions.

## 2 Related Work

Multithreading is a standard technique used to handle the asynchronous requests commonly encountered by web servers, messaging middleware, and database servers [Ber96,Ric96]. For these kinds of applications, multithreading can increase server responsiveness, scalability and throughput, and enhance process-to-process communication. Due to their constant stream of network requests, web servers and network middleware are particularly apt to fall under the category of I/O intensive applications. The vast majority of their time is spent waiting for I/O operations to complete. Under multithreading, if a thread blocks, the processor simply executes another active thread. Also, for servers running on SMP hardware, multithreading facilitates the server load distribution across processors. Compared to switching processes, multithreading is relatively cheap because threads share the same process space.

However, multithreading is not a panacea. Under a thread-per-request architecture, multithreading incurs a run-time overhead for creating and destroying threads on the fly. One accepted and standardized server implementation of a thread-per-request architecture is the concurrent threading model [Ric96]. In this model, each instance of thread creation and destruction requires allocation and deallocating space (memory or disk) and CPU cycles. For example, to create a thread in the Windows NT or Solaris operating systems requires allocating one megabyte virtual memory (default size) for the thread stack [Lew96,Ric96,Sun95]. A high volume of service requests results in frequent memory allocation and deallocation, and becomes a major factor in performance bottlenecks.

Under a thread-pool architecture, maintaining threads in the pool incurs a run-time overhead for context-switching. Each thread runs on lightweight process (LWP), which can be thought of as a virtual CPU [Lew96]. A context-switch refers to the action of removing an active thread from its LWP, and replacing it with another thread that is waiting to be run. During a context-switch, the currently running thread's registers must be saved and its page tables unloaded. The registers and page tables of a waiting thread must be reloaded. On a SPARC station 10/41, a thread-level context switch requires about 20 microseconds [Lew96]. Additional processing overhead is introduced to safeguard the integrity and consistency of the shared data by various synchronization APIs that ensure the atomicity of thread operations[Age99].

Thread-pool architectures have gained a wide acceptance in computer industry. They are one of the Common Object Request Broker Architecture (CORBA) multithreading architectures used in Object Request Brokers (ORB) implementations [Sch98], and have been adopted by web servers such as Microsoft Internet Information Server (IIS) [Man98,Ric96]. In addition, thread pool APIs are provided both by Microsoft (I/O completion ports) [Man98,Rich96] and by Sun Microsystems in Java SDK [Oak97].

Since a thread-pool architecture prespawns a pool of threads, a thread-per-request architecture can be consider a special case of a thread pool where the number of prespawned threads is zero. Therefore, we consider the general problem of determining the optimal thread-pool size. While the run-time overhead of thread context-switching is cheaper than creating new threads, the extra overhead of maintaining a large thread pool may outweigh these cost savings, consuming system resources in the form of memory and increased scheduling overheads [Vert95]. For example, if a web server only receives a few requests per day, the run-time costs of creating and maintaining a thread pool at all may outweigh the benefits of not having to create and destroy threads on the fly, and the optimal pool size could be zero.

Currently, thread pool size is set via a combination of heuristics and practical experience.
A heuristic is used to set the initial pool size, and then operators are advised to monitor the system load and to modify the pool size if there are performance bottlenecks [Cal97]. One such rule of thumb is that the size of thread pool should be two times of the number of CPUs on the host machine [Ric96]. Another rule, used by Microsoft's IIS, initially allocates 10 threads in the thread pool per CPU, and allows the size of the thread pool to grow based on the number of client requests. The thread pool's maximum size is heuristically set to be two times of the number of megabytes of RAM in the host [Ric96].

Both of these heuristics are based on processor and memory capacities, and provide no theoretical justification. The effect of the system load incurred by client requests is only considered by the second heuristic, and only in an ad-hoc manner. As a result, these heuristics without theoretical justification are likely to create too large or too small of a thread pool, thereby wasting system resources and causing system slowdowns. The main objective of this paper is to establish a principled framework for determining the optimal thread pool size theoretically.

## 3 Theoretical Formulation of Thread Pool Size

### 3.1 Overview

In this section, we provide a mathematical analysis of how to determine the optimal thread pool size. Our primary concern is to arrive at an initial formulation that is tractable and experimentally verifiable, and reasonably realistic. We start with several assumptions that may be relaxed in future work.

The first assumption is that each thread in the pool has the same execution priority and receives an equal share of CPU time. This assumption is reasonable for Internet-based applications where all web requests generally receive uniform treatment. Similar fairness criteria are a common design principle for operating systems, despite their different thread scheduling details [Lew96, Cus93, Ric96].

The second assumption is that one web request is very much like another, and that the memory and I/O resources required by different threads are minimal and do not differ substantially between requests. While this is less justifiable than the first assumption, it is appropriate for sites which handle only one type of web request with a fairly standard, small holding time (the duration of time over which memory blocks will be held for processing the web request). Future extensions to this model may include representing the holding time as a distribution.

The third assumption is that we can aggregate many of the system resource constraints into observable costs. For this initial model, we focus on the easily measurable quantity of processing latency (i.e., elapsed time). Latency is directly or indirectly influenced by many intertwined factors, including contention for physical memory, kernel bookkeeping costs, and system hardware/software configuration. The thread pool overhead costs we discuss in this paper are considered as the weighted sum of such processing latency factors. Obvious future extensions include explicitly considering system memory and processing constraints.

### 3.2 Analysis

We start by defining two overhead costs associated with threads. Let $c_1$ be the overhead for a single thread's creation and destruction and $c_2$ be the overhead of maintaining each thread in a thread pool. We measure these overheads by the amount of elapsed time each activity consumes. The dominant cost for $c_1$ is the elapsed time required for memory allocation. Remember that creating a thread on the Windows NT and Solaris operating systems requires allocating one megabyte virtual memory (default) for the stack. The elapsed time for this activity differs widely depending on the underlying operating system and processor architecture. For example, to create a thread in Windows NT requires a system call, and uses additional kernel resources. On the other hand, Solaris adopts a two-level model, so that thread creation is a user-level activity and does not require using kernel resources [Lew96]. Table 1 provides performance numbers of threads, LWPs, and processes, as measured by creation time on a Solaris operating system [Sun95]:

| Thread Creation | Microseconds |
|---|---|
| Unbound Thread | 101 |
| Bound thread (LWP) | 422 |
| Fork () process | 3,057 |

Table 1 Performance Numbers of Thread, LWP, and Process

Thus, assuming the same system configurations (i.e., similar processing speeds, and available memory resources), we would expect the value of $c_1$ to be greater for Windows NT than Solaris.

The overhead cost for $c_2$ refers to the cost of maintaining and running threads from the thread pool. We assume that the dominant cost is the elapsed time for thread context switching, which differs from platform to platform, and use that to estimate the overall cost. For example, a thread-level context switch is approximately 20 microseconds on a SPARC station 10/41 [Lew96]. In multithreading environment, to get fair share of CPU time, each thread with the same priority is scheduled in round-robin fashion [Vert95]. For a pool size of $n$, the overall overhead of thread pool in context switching is roughly $20 \cdot n$ microseconds, which is proportional to thread pool size.

To simplify our derivation, $c_1$ and $c_2$ can be reasonably assumed to be constants when the platform and server application are fixed. The performance advantage of a thread pool lies in the fact that $c_1 >> c_2$. For instance, creating an unbound thread needs 422 microseconds while the amount of time used for context switching is about 20 microseconds in Solaris operating systems. In Table 2, we summarize the costs associated with a thread pool of size $n$, when the total number of concurrently running threads is $r$. We compare the thread pool costs to the costs of not having a thread pool (i.e., having a thread-per-request architecture). We also define the gain of a thread pool of size $n$ to be the cost of having no thread pool minus the cost of having a thread pool.

|  | Cost of thread pool | Cost of no thread pool | Gain of thread pool |
|---|---|---|---|
| $0 \leq r \leq n$ | $c_2 \cdot n$ | $c_1 \cdot r$ | $c_1 \cdot r - c_2 \cdot n$ |
| $r > n$ | $c_2 \cdot n + c_1 \cdot (r-n)$ | $c_1 \cdot r$ | $c_1 \cdot n - c_2 \cdot n$ |

Table 2 Cost Analysis and Gain of Thread Pool

In Table 2, we consider two cases. The first occurs when the number of running threads is less than thread pool size, (i.e., $0 \leq r < n$). For this case, the thread pool can handle all incoming web requests, and the overhead cost is simply the overhead of maintaining a thread pool size of $n$, or $c_2 \cdot n$. When there is no thread pool, a thread must be created and destroyed to handle all incoming web requests, and the overhead cost is $c_1 \cdot r$. Therefore, the benefit gained by using a thread pool is $c_1 \cdot r - c_2 \cdot n$.

The second case occurs when the number of requests exceeds the thread pool size (i.e., $r > n$). An additional $(r - n)$ requests need to be created on the fly to meet the demand, so that the thread pool cost is $c_2 \cdot n + c_1 \cdot (r - n)$. When there is no thread pool, the overhead cost is once again $c_1 \cdot r$. Thus, the corresponding gain of thread pool is $c_1 \cdot n - c_2 \cdot n$.

In real application environments, the number of the concurrently running threads varies over time, depending on the clients' request patterns, and the server performance characteristics such as the overheads of creating a thread and maintaining a thread in thread pool. To calculate the benefit of using a thread pool, we assume that $r$ is a random variable representing the number of concurrently running threads, with $f(r)$ the probability distribution. Hence the expected gain, or expected utility [Sin99], of having a thread pool can be expressed as:

$$E(n) = \sum_{r=0}^{n} (c_1 \cdot r - c_2 \cdot n) \cdot f(r) + \sum_{r=n+1}^{\infty} (c_1 \cdot n - c_2 \cdot n) \cdot f(r) \qquad [1]$$

To obtain the optimal size of thread pool is equivalent to identifying $n^*$ among all possibilities, which yields the highest gain, i.e.,

$$E(n^*) = \sup_{n \in \mathrm{N}} E(n) \qquad [2]$$

which is equivalent to minimizing the associated costs. For computation simplicity, the discrete probability $f(r)$ can be replaced by $p(r) \cdot dr$, and $p(r)$ is the probability density. Then the expected gain for the thread pool is rewritten as

$$E(n) = \int_0^n (c_1 \cdot r - c_2 \cdot n) \cdot p(r) dr + \int_n^{\infty} (c_1 \cdot n - c_2 \cdot n) \cdot p(r) dr \qquad [3]$$

Using the first-order derivative condition, shown in Eqn (4), we can find the optimal thread pool size $n^*$ that maximizes the expected gain of having a thread pool:

$$\frac{dE}{dn} = -c_2 + c_1 \cdot \int_{n^*}^{\infty} p(r) dr = 0 \qquad [4]$$

The second-order condition, namely that the second derivative of the expected gain with respect to $n$ be nonpositive, holds since:

$$\frac{d^2E}{dn^2} = -c_1 \cdot p(n) \le 0$$

Next we rewrite Eqn (4) by defining the cost ratio of maintaining a thread in thread pool and creating a thread as $\zeta = {}^{c_2}\!/_{c_1}$ :

$$\int_{n^*}^{\infty} p(r)\cdot dr = \frac{c_2}{c_1} = \zeta,\quad \int_0^{n^*} p(r)\cdot dr = 1 - \frac{c_2}{c_1} = 1-\zeta \qquad [5]$$

In fact, since the pool size is an integer, it can be determined by the following equation:

$$\int_0^{\lfloor n^* \rfloor} p(r)\cdot dr \le= 1-\zeta,\quad \int_0^{\lfloor n^*+1 \rfloor} p(r)\cdot dr > 1-\zeta \qquad [6]$$

where $\lfloor x \rfloor$ is the next lowest integer number to $x$. It follows from Eqn (6) that the optimal size of the thread pool is proportional to $\zeta$ (or $c_2/c_1$). This leads to intuitively reasonable results; when thread creation is costly or thread pool maintenance is cheap, the optimal thread pool size will be large. The cost ratio $\zeta$ of Solaris operating systems can be calculated using thread creation values from Table 1, as well as Solaris's thread-level context switch time of 20 microseconds. The performance ratio $\zeta$ is 0.198 when unbound threads are used, and is 0.0474 when bound threads (LWPs) are used.

Eqns (5) and (6) show that the optimal thread pool size is determined not only by the cost ratio of thread creation to thread maintenance, but also by system load. Our derivation characterizes these associated costs and obtains the optimal pool size by maximizing the expected gain.

## 4 Practical Applications of Theoretical Results

In this section, we demonstrate the use of these formulas in two example applications. Example 1 derives the optimal thread pool size when the probability distribution is known. Example 2 illustrates how to obtain an estimated probability distribution from data about client requests. Since the probability distribution, $p(r)$, of client requests is influenced by a wide array of factors such as users' access patterns and server performance characteristics, as well as their interdependence, it may not be analytically tractable. In example 2, we demonstrate how sampling techniques may be used to estimate the distribution.

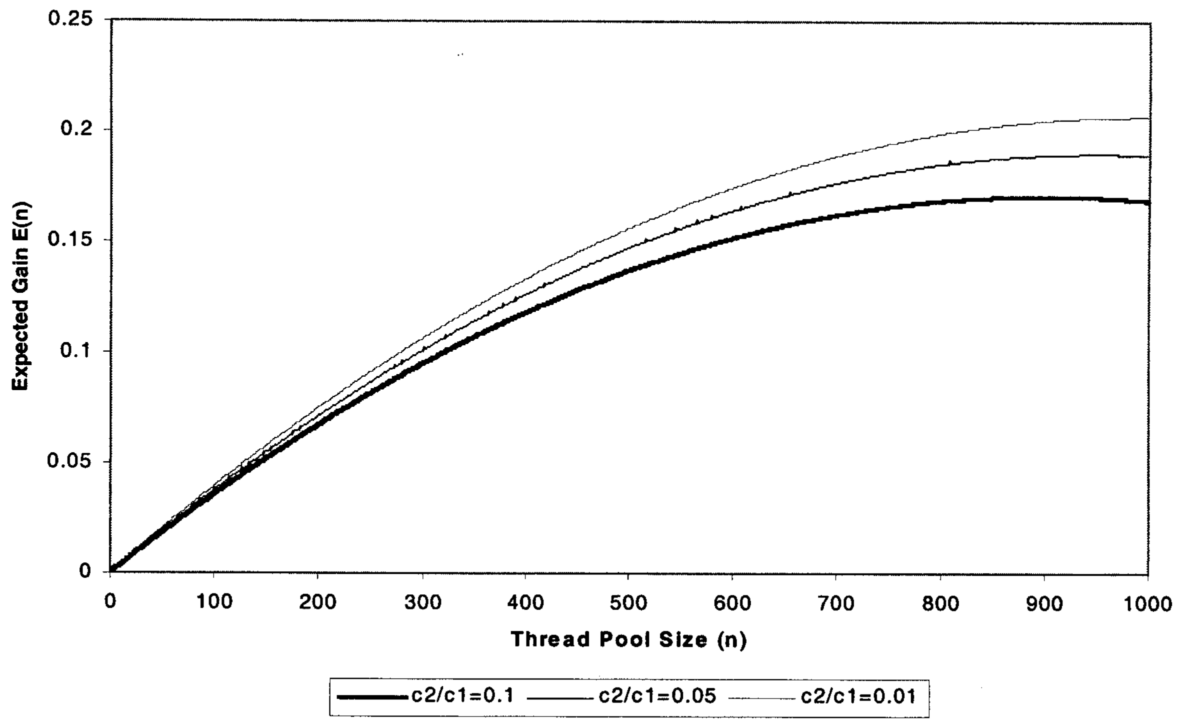

Figure 1 Expected Gain (Seconds) vs. Thread Pool Size ($c_1$ = 422 microseconds)

Example 1: We assume that the probability density $p(r)$ is a uniform distribution $Uniform(0,1000)$. Then, the optimal pool size is determined using Eqn(5).

$$\frac{\lfloor n^* \rfloor}{1000} <= 1-\zeta, \frac{\lfloor n^* \rfloor + 1}{1000} > 1-\zeta$$

if $\zeta = 0.01$, then the optimal pool size is $1000*(1-\zeta) = 990$ megabytes of virtual memory. Figure 1 shows the expected gain of thread pool $E(n)$ for thread pool sizes from 0 to 1000, under different cost ratios $\zeta$ (0.1, 0.05, and 0.01). We also see that the total gain increases with thread pool size up to the optimal size, then starts to decrease. In other words, the marginal gain of adding another thread to the pool is decreasing.

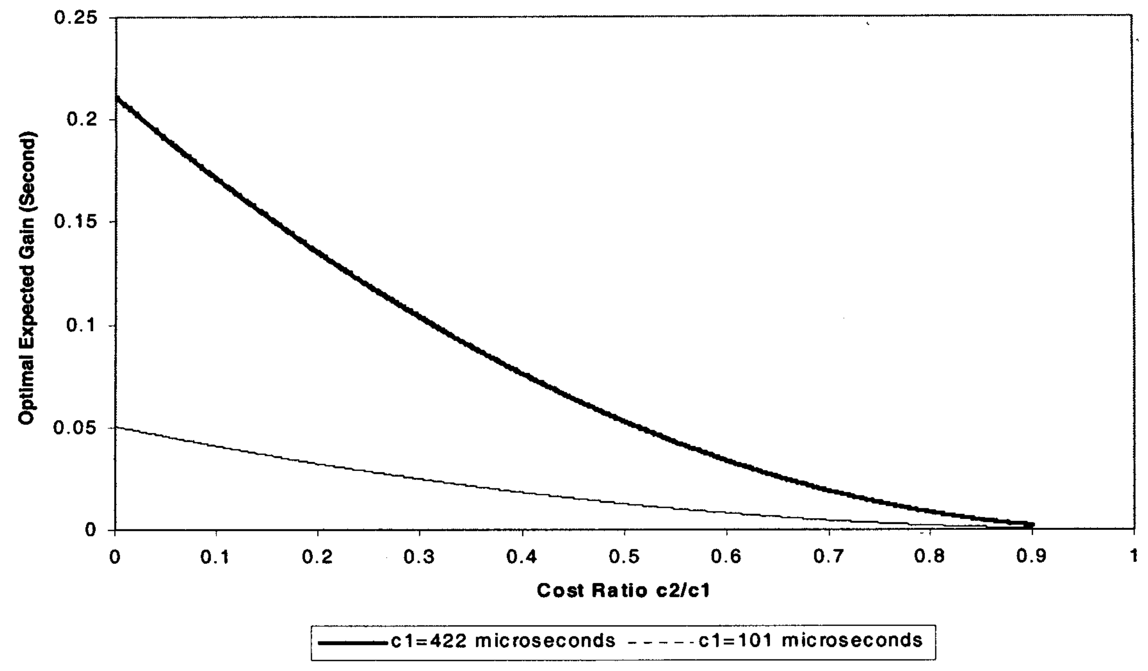

Figure 2 optimal expected gain vs. performance ratio $\zeta$

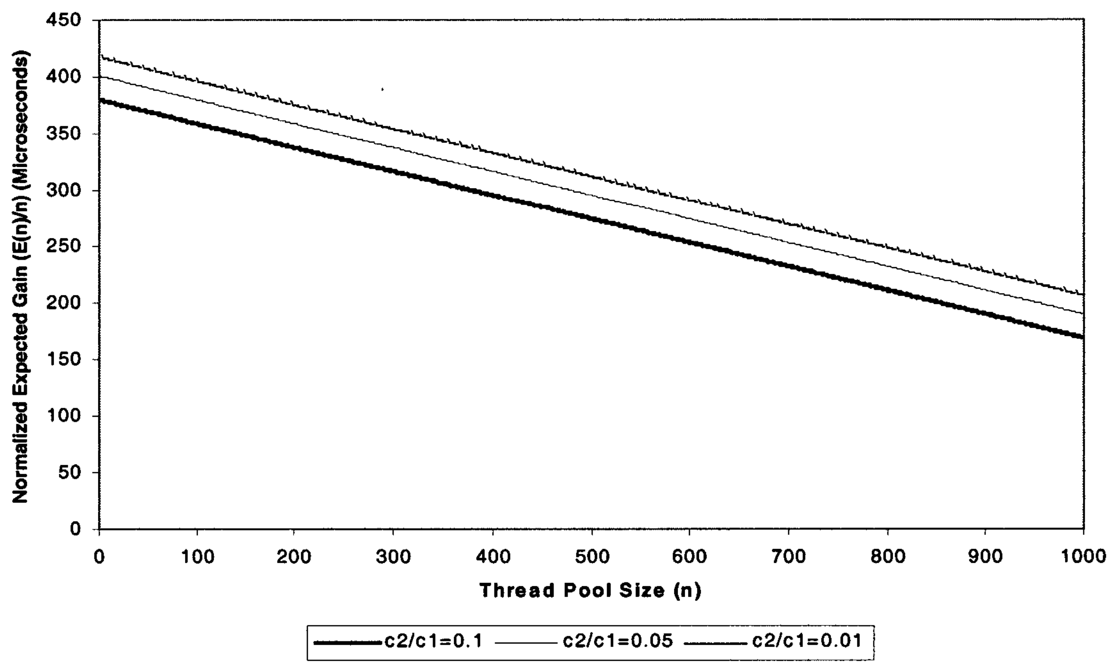

Figure 3 normalized expected gain vs thread pool size ($c_1 = 422$ microseconds)

Not surprisingly, the expected gain of a thread pool is inversely proportional to the cost ratio $\zeta$ as illustrated in Figure 2. We expect a lower gain when $\zeta$ is higher, due to the relatively more expensive thread maintenance. The diminishing gain associated with a high performance ratio $\zeta$ is observed in Figure 2, which is consistent with our intuition.

In Figure 3, we show these same results, but as average gain per thread ($E(n)/n$), rather than the total gain. Each different curve corresponds to a specified cost ratio $\zeta$ (0.1, 0.05, and 0.01). In this graph, we can see clearly how the average expected gain per thread diminishes as the thread pool size increases.

Example 2: In this example, we illustrate how to determine the optimal pool size under real-world conditions. The probability distribution $p(r)$ can be arbitrarily complicated, based on dynamically changing user access patterns and available system resources. However, by sampling real-world data about the current number of running threads, we can avoid analytically characterize these factors.

First, we determine the time period over which we are estimating the probability distribution. For example, a web server might have peak and off-peak time periods that are estimates separately. For this initial model, we assume that the web server's traffic patterns are stable enough that this estimation only needs to occur occasionally, and the cost involved in the estimation are therefore not important.

Next, we divide the time period into $t$ time-slices. We can obtain a good approximation to the probability distribution $p(r)$ when the sample size (or total number of time slices) is sufficiently large, as guaranteed by the central limit theorem [Sic99]. Let the array *Sample[t]* contain the number of concurrently running threads for each time slice. We assume that the number of concurrently running threads is less than *2024*, and assign the array *Estimate_Pro[2024]* to contain the estimated probability distribution after processing. The following algorithm calculates the estimated probability distribution with a running time of $O(t)$. This algorithm also determines the maximum number of concurrently running threads in any one time-slice.

Let the array *Sample[m]* contain the actual sampling and the array *Estimate_Pro[2024]* contain the estimated probability distribution after processing. We assume that the number of concurrently running threads is less than *2024*. Thus the expected utility probability distribution can be derived via the algorithm shown in Figure 4, its running time is of $O(m)$, which $m$ is the sample size. The estimated probability distribution, calculated above in the array *Estimate_Pro*, is shown below in tabular format:

| $r$ | *0* | *1* | 2 | *3* | ... | $K$ | .. |
|---|---|---|---|---|---|---|---|
| $f_i$ | $m_0$ | $m_1$ | $m_2$ | $m_3$ | | $m_k$ | .. |

According to the table above, when *r=2*, there are two concurrently running client requests in a total of $m_2$ out of $m$ sampled time slices. The total number of requests during the entire time period is $m = \sum_{i=0}^{M} m_i$ where $M$ is the maximum number of currently running threads observed in any one time slice (and is calculated in the algorithm above by the variable *maximal_running_threads).*

```
Begin
Input: n, MAX, Array Sample[n];
Output: maximal_number_threads;
Output: Estimate_pro[MAX];
maximal_running_threads=0;
Int i, Total=0;
For ( i = 0  to n )
{
    Total += Sample[ i];
    Estimate_Pro[Sample[i]] ++;
   if (Sample[i] >maximal_running_threads)
         maximal_running_threads = Sample[i];
}
For (  i=0 to maximal_running_threads)
{
     Estimate_Pro[i] = Estimate_Pro [i]/Total;
}
```

Figure 4 Calculation of Estimated Probability Distribution

We can now define the estimated probability $\overline{p_i}$ as:

$$\overline{p_i} = f_i / m = \frac{m_i}{\sum_{i=0}^{M} m_i} \qquad 0 \le i \le M$$

At this point, the optimal pool size can be determined using the estimated probability distribution and the system parameters, $c_1$ and $c_2$, in the algorithm shown in Figure 5. Finally, we show how to determine an adequate sampling frequency. Let $R$ be a random variable representing the number of concurrently running threads observed, where $R$ can take on the values *0, 1, 2... k* with the probabilities $p_0, \cdots, p_k$ respectively. We observe the number of concurrently running threads $m$ times with $R$. For separate web requests, we can reasonably assume that the sequence of $R_0, \cdots, R_m$ are independent, identically distributed random variables, each having the same the distribution. Let $f_i = f_i(R_1,\cdots,R_m)$ be the number of times that $i$ running threads are observed in the samples $R_0,\cdots,R_m$, then $f_i$ has a binomial distribution with parameters $m$ and $p_i$. That is, $\frac{f_i}{m}$ is the random variable with the mean $p_i$ and variance $\frac{p_i\cdot(1-p_i)}{m}$. Using Chebyshev's inequality [Ash65, Ross97], we derive the following equation, indicating that when the sample size $tm$ is large, $\frac{f_i}{m}$ approaches $p_i$ with the probability 1.

$$P(|\frac{f_i}{m}-p_i|\geq\varepsilon)\leq\frac{E[(f_i/m-p_i)^2]}{\varepsilon^2}=\frac{(1-p_i)\cdot p_i}{m\cdot\varepsilon^2}\qquad i=0,\cdots,k \qquad [7]$$

Using Eq [7], we can determine the required sample size $m$ based on the estimate error $\varepsilon$ and the confidence level. Also, we know that $(1-p_i)p_i\leq 1/4$, and for many distributions can be much less. For example, given the estimated error $\varepsilon=0.05$, confidence level $90\%$, then the sample size can be determined as

$$0.1=P(|\frac{f_i}{m}-p_i|\geq\varepsilon)\leq\frac{(1-p_i)\cdot p_i}{m\cdot\varepsilon^2}\leq\frac{1}{4\cdot m\cdot\varepsilon^2}\quad,\quad m\leq\frac{1}{4*0.1*0.05^2}=1{,}000$$

The approach illustrated in this section is of practical significance, and is based on theoretical results derived in this paper. It is widely applicable due to the ability to handle arbitrary probability distributions.

```
Begin
Input: maximal_number_threads;
Input: c1, c2, Estimate_Pro [maximal_number_threads];

Output: optimal_thread_size;
Int Cumulative_probability =0.0;
Int n;
For( n=0 to maximal_running_thread )
{
   Cumulative_probability += Estimate_Pro(n);
   if(Cumulative_probability >=1- c2/c1
      break;
}
optimal_thread_size = n;
End
```

Figure 5 Calculation of Optimal Thread Pool Size

## 5 Conclusion

Thread-pool architectures promise to improve server run-time performance though more efficient system resource utilization. They avoid the overhead incurred by thread-per-request architectures due to repeated thread creation and destruction. However, setting the thread pool size according to various rules of thumb can be expected to produce poor, or at least sub-optimal, server performance. If the thread pool size is too small, some of the cost savings of having a thread pool will be lost. On the other hand, a larger-than-needed thread pool introduces excessive thread context switching, which also wastes system resources. The optimal size of thread pool is a dynamic performance metric that is influenced by the request activity, the system capacity, and the system configuration. To achieve a better performance, the size of thread pool should adjust to reflect this changing flow of client requests.

In this paper, we first formalize this problem by establishing mathematical relationship among optimal thread pool size, system load and the associated costs. We demonstrate that the proposed model is widely applicable due to its ability to adapt to an arbitrary probability distribution. This paper constitutes a significant step toward a better understanding of the interaction between optimal pool size, request activity and associated system resource overheads incurred. Our next step is to experimentally compare the performance of our formulation with those using existing heuristics. Basing our analysis on a single easily measurable performance metric, latency, should allow us to more easily validate our results experimentally.

To make the problem mathematically tractable while capturing its essence, we have considered a system characterized by steady state probabilities. In the future, we expect to extend and refine this model. We will consider the case where the cost proportionality of creating a thread is not constant, but is affected by contention for other system resources and experimentally test it against existing heuristics. We will also look at incorporating slightly different models of thread-pool behavior. For example, revising our model to handle threads that can be created with a certain time-out value, so that if the timer expires before the thread is used again, then the thread kills itself. Finally we will consider the performance of a dynamically changing the thread pool size when the request traffic is bursty and statistically unstable. We believe that incorporating this kind of dynamic, yet analytic, modeling into existing systems can lead to significant performance improvements.

## 6 References

[Ash90] Robert Ash, *Information Theory,* Dover Publications, 1990.

[Age99] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y. S. Ramakrishna, and Derek White, *An Efficient Meta-lock for Implementation Ubiquitous Synchronization*, The SML Technical Reports, Sun Microsystems Laboratories, Sun Mircosystems, Inc., 1999.

[Berg96] Cliff Berg, *How do Threads work and how can I create a general-purpose event*? Dr. Dobb's Journal, Vol. 21 No. 11, pp. 111-117, Nov, 1996.

[But97] David Butenhof, *Programming with POSIX Threads.* Addison Wesley, 1997

[Cal97] John Calcote, *Thread Pools and Server Performance.* Dr. Dobb's Journal, pp. 60-64, July 1997.

[Cus93] Helen Custer, *Inside Window NT*, Microsoft Press, 1993.

[Fro99] Jim Frost, *At your service.* Server Workstation Expert, Vol. 10, no. 6, pp. 44-49, Jun. 1999.

[Lew96] Bil Lewis and Daniel J. Berg, *Threads Primer—A Guide to Multithreading Programming*, SunSoft Press A Prentice Hall Title, 1996.

[Kwa99] Hantak Kwak, Ben Lee, Ali R. Hurson, Suk-Han Yoon, Woo-Jong Hahn, *Effects of Multithreading on Cache Performance*, IEEE Transactions on Computers, Vol 48, no. 2, pp. 176-184, Feb. 1999.

[Man98] Kevin T Manley, *General-purpose Threads with I/O Completion Ports*, C/C++ Users Journal, Vol. 16, No. 4, pp. 75-83, April 1998.

[Mim99] Bob Mims. *Genealogy site overwhelmed by millions of hits.* The Salt Lake Tribune, May 28, 1999.

[Oak97] Scott Oaks, and Henry Wong, *Java Threads.* O'Reilly and Associates, Inc., 1997.

[Ric96] Jeffrey Richter, *Advanced Windows.* Third Edition, Microsoft Press, 1996.

[Ross97] Sheldon Ross, *Introduction to Probability Model*, Academic Press, 1997.

[Sch98] Douglas C Schmidt, *Evaluating Architectures for Multithreaded Object Request Brokers.* Association for Computing Machinery, Communication of the ACM; New York, Vol. 41, no. 10, pp. 54-60, Oct. 1998.

[Sin99] Nozer D. Singpurwalla and Simon P. Wilson, *Statistical Methods in Software Engineering --- Reliability and Risk*, Springer, New York, 1999.

[Sun95] *Frequently Asked Questions*, http://www.sun.com/workshop/threads/faq.html

[Vert95] John Vert, *Writing Scalable Applications for Window NT*, Window NT Base Group, http://msdn.microsoft.com/library.