The following paper was originally published in the
Proceedings of the 2nd USENIX Windows NT Symposium
Seattle, Washington, August 3–4, 1998

A Thread Performance Comparison: Windows NT and Solaris on A Symmetric Multiprocessor

Fabian Zabatta and Kevin Ying
Brooklyn College and CUNY Graduate School

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org/

A Thread Performance Comparison: Windows NT and Solaris on A Symmetric Multiprocessor

Fabian Zabatta and Kevin Ying
Logic Based Systems Lab
Brooklyn College and CUNY Graduate School
Computer Science Department
2900 Bedford Avenue
Brooklyn, New York 11210
{fabian, kevin}@sci.brooklyn.cuny.edu

Abstract

Manufacturers now have the capability to build high performance multiprocessor machines with common PC components. This has created a new market of low cost multiprocessor machines. However, these machines are handicapped unless they have an operating system that can take advantage of their underlying architectures. Presented is a comparison of two such operating systems, Windows NT and Solaris. By focusing on their implementation of threads, we show each system's ability to exploit multiprocessors. We report our results and interpretations of several experiments that were used to compare the performance of each system. What emerges is a discussion on the performance impact of each implementation and its significance on various types of applications.

1. Introduction

A few years ago, high performance multiprocessor machines had a price tag of $100,000 and up, see [16]. The multiprocessor market consisted of proprietary architectures that demanded a higher cost due to the scale of economics. Intel has helped to change that by bringing high performance computing to the mainstream with its Pentium Pro (PPro) processor. The PPro is a high performance processor with built in support for multiprocessing, see [4]. This coupled with the low cost of components has enabled computer manufacturers to build high performance multiprocessor machines at a relatively low cost. Today a four processor machine costs under $12,000.

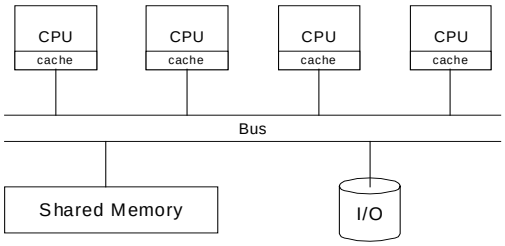

Figure 1: A basic symmetric multiprocessor architecture.

1.1 Symmetric Multiprocessing

Symmetric multiprocessing (SMP) is the primary parallel architecture employed in these low cost machines. An SMP architecture is a tightly coupled multiprocessor system, where processors share a single copy of the operating system (OS) and resources that often include a common bus, memory and an I/O system, see Figure 1. However, the machine is handicapped unless it has an OS that is able to take advantage of its multiprocessors. In the past, the manufacturer of a multiprocessor machine would be responsible for the design and implementation of the machine's OS. It was often the case, the machines could only operate under the OS provided by the manufacturer. The machines described here are built from common PC components and have open architectures. This facilitates an environment where any software developer can design and implement an OS for these machines. Two mainstream examples of such operating systems are Sun's Solaris and Microsoft's Windows NT. They exploit the power of multiprocessors by incorporating *multitasking* and

multithreading architectures. Their implementations are nevertheless very different.

1.2 Objects Of Execution

Classic operating systems, such as UNIX, define a *process* as an object of execution that consists of an address space, program counter, stack, data, files, and other system resources. Processes are individually dispatched and scheduled to execute on a processor by the operating system *kernel*, the essential part of the operating system responsible for managing hardware and basic services. In the classic case, a multitasking operating system divides the available CPU time among the processes of the system. While executing, a process has only one unit of control. Thus, the process can only perform one task at a time.

In order to exploit concurrency and parallelism, operating systems like NT and Solaris further develop the notion of a process. These operating systems break the classical process into smaller sub-objects. These sub-objects are the basic entity to which the OS allocates processor time. Here we will refer them as *kernel-level objects of execution*. Current operating systems allow processes to contain one or more of these sub-objects. Each sub-object has its own *context*[1] yet it shares the same address space and resources, such as open files, timers and signals, with other sub-objects of the same process. The design lets the sub-objects function independently while keeping cohesion among the sub-objects of the same process. This creates the following benefits.

Since each sub-object has its own context each can be separately dispatched and scheduled by the operating system kernel to execute on a processor. Therefore, a process in one of these operating systems can have one or more units of control. This enables a process with multiple sub-objects to overlap processing. For example, one sub-object could continue execution while another is blocked by an I/O request or synchronization lock. This will improve throughput. Furthermore with a multiprocessor machine, a process can have sub-objects execute concurrently on different processors. Thus, a computation can be made parallel to achieve speed-up over its serial counterpart. Another benefit of the design arises from sharing the same address space. This allows sub-objects of the same process to easily communicate by using shared global variables. However, the sharing of data requires synchronization to prevent simultaneous access. This is usually accomplished by using one of the synchronization variables provided by the OS, such as a *mutex*. For general background information on synchronization variables see [14], for information on Solaris's synchronization variables see [1, 5, 12, 13], and [7, 10, 15] for Windows NT's synchronization variables.

[1] This refers to its state, defined by the values of the program counter, machine registers, stacks, and other data.

2. Solaris's and Windows NT's Design

Windows NT and Solaris further develop the basic design by sub-dividing the kernel-level objects of execution into smaller *user-level objects of execution*. These user-level objects are unknown to the operating system kernel and thus are not executable on their own. They are usually scheduled by the application programmer or a system library to execute in the context of a kernel-level object of execution.

Windows NT and Solaris kernel-level objects of execution are similar in several ways. Both operating systems use a priority-based, time-sliced, preemptive multitasking algorithm to schedule their kernel-level objects. Each kernel-level object may be either interleaved on a single processor or execute in parallel on multiprocessors. However, the two operating systems differ on whether user-level or kernel-level objects should be used for parallel and concurrent programming. The differences have implications on the overall systems' performances, as we will see in later sections.

2.1 NT's Threads and Fibers

A *thread* is Windows NT's smallest kernel-level object of execution. Processes in NT can consist of one or more threads. When a process is created, one thread is generated along with it, called the *primary thread*. This object is then scheduled on a system wide basis by the kernel to execute on a processor. After the primary thread has started, it can create other threads that share its address space and system resources but have independent contexts, which include execution stacks and thread specific data. A thread can execute any part of a process' code, including a part currently being executed by another

thread. It is through threads, provided in the Win32 application programmer interface (API), that Windows NT allows programmers to exploit the benefits of concurrency and parallelism.

A *fiber* is NT's smallest user-level object of execution. It executes in the context of a thread and is unknown to the operating system kernel. A thread can consist of one or more fibers as determined by the application programmer. Some literature [1,11] assume that there is a one-to-one mapping of user-level objects to kernel-level objects, this is inaccurate. Windows NT does provide the means for many-to-many scheduling. However, NT's design is poorly documented and the application programmer is responsible for the control of fibers such as allocating memory, scheduling them on threads and preemption. This is different from Solaris's implementation, as we shall see in the next section. See [7, 10] for more details on fibers. An illustrative example of NT's design is shown in Figure 2.

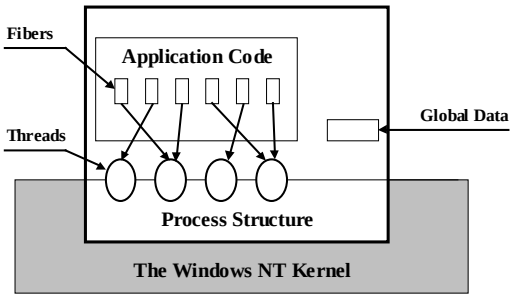

Figure 2: The relationships of a process and its threads and fibers in Windows NT.

2.2 Solaris's LWPs and Threads

A *light weight process* (LWP) is Solaris's smallest kernel-level object of execution. A Solaris process consists of one or more light weight processes. Like NT's thread, each LWP shares its address space and system resources with LWPs of the same process and has its own context. However, unlike NT, Solaris allows programmers to exploit parallelism through a user-level object that is built on light weight processes. In Solaris, a *thread* is the smallest user-level object of execution. Like Windows NT's fiber, they are not executable alone. A Solaris thread must execute in the context of a light weight process. Unlike NT's fibers, which are controlled by the application programmer, Solaris's threads are implemented and controlled by a system library. The library controls the mapping and scheduling of threads onto LWPs automatically. One or more threads can be mapped to a light weight process. The mapping is determined by the library or the application programmer. Since the threads execute in the context of a light weight process, the operating system kernel is unaware of their existence. The kernel is only aware of the LWPs that threads execute on. An illustrative example of this design is shown in Figure 3.

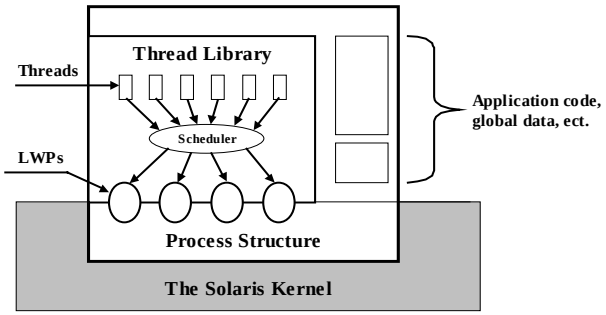

Figure 3: The relationships of a process and its LWPs and threads in Solaris.

Solaris's thread library defines two types of threads according to scheduling. A *bound* thread is one that permanently executes in the context of a light weight process in which no other threads can execute. Consequently, the bound thread is scheduled by the operating system kernel on a system wide basis. This is comparable to an NT thread.

An *unbound* thread is one that can execute in the context of any LWP of the same process. Solaris uses the thread library for the scheduling of these unbound threads. The library works by creating a pool of light weight processes for any requesting process. The initial size of the pool is one. The size can be automatically adjusted by the library or can be defined by the application programmer through a programmatic interface. It is the library's task to increase or decrease the pool size to meet the requirements of an application. Consequently, the pool size determines the concurrency level (CL) of the process. The threads of a process are scheduled on a LWP in the pool, by using a priority based, first-in first-out (FIFO) algorithm. The priority is the primary algorithm and FIFO is the secondary algorithm (within the same priority). In addition, a thread with a lower priority may be preempted from a LWP by higher priority thread or by a library call. Here we will only consider threads of the same priority without preemption. Thus, the scheduling algorithm is solely FIFO. In this case, once a thread is

executing it will execute to completion on a light weight process unless it is blocked or preempted by the user. If blocked, the library will remove the blocked thread from the LWP and schedule the next thread from the queue on that LWP. The new thread will execute on the LWP until it has completed or been blocked. The scheme will then continue in the same manner. For more information on Solaris's design and implementation, see [1, 5, 12, 13].

3. Experiments

Seven experiments were conducted to determine if differences in the implementation, design and scheduling of threads would produce significant differences in performance. None of the experiments used NT's Fibers since they require complete user management and any comparison using them would be subject to our own scheduling algorithms. Furthermore, we wanted to test each system's chosen thread API. Thus we chose to compare the performance of NT's threads to three variations of Solaris's threads (bound, unbound, and restricted concurrency). Although it may seem unfair to compare NT's kernel implementation to Solaris's user implementation, it is fair because Solaris's implementation is not purely user based. Embedded in its design are kernel objects, LWPs. Therefore, like the NT case, the OS kernel is involved in scheduling. Furthermore, the comparison is between each operating system's chosen thread model and thus we are comparing models that each system has specifically documented for multithreading. The models do have fundamental differences, yet they still warrant a comparison to determine how each could effect different aspects of performance. The conducted experiments tried to achieve this by measuring the performance of each system under different conditions. In some cases, the experiments tried to simulate the conditions of real applications such the ones found in client/server computing and parallel processing. The seven experiments were:

1. Measure the maximum number of kernel threads that could be created by each system (Section 3.2).
2. Measure the execution time of thread creation (Section 3.2).
3. Measure the speed of thread creation on a heavily loaded system (Section 3.2).
4. Determine how each operating system's thread implementation would perform on processes with CPU intensive threads that did not require any synchronization (Section 3.3).
5. Determine how each operating system's thread implementation would perform on processes with CPU intensive threads that required extensive synchronization (Section 3.4).
6. Determine the performance of each operating system's implementation on parallel searches implemented with threads (Section 3.5).
7. Determine the performance of each operating system's implementation on processes with threads that had bursty processor requirements (Section 3.6).

To restrict the concurrency of Solaris's threads in the experiments, unbound threads were created and their concurrency was set to the number of processors, noted by (**CL = 4**) in the tables. In theory, this created a LWP for each processor and imposed on the thread library to schedule the unbound threads on the LWPs. To use Solaris unbound threads, threads were created without setting the concurrency level. Thus, only one LWP was made available to the thread library, and the test program could not exploit the multiprocessors. However, the thread library is documented [11, 12, 13] as having the capability to increase or decrease the pool of LWPs automatically. Therefore, any processes using unbound threads, including processes that contain with restricted concurrency, indirectly test the thread library's management of the LWP pool.

Our reported benchmarks are in seconds for an average of 10 trials. In most cases, the standard deviations (σ) for trails were less than 0.15 seconds. We only report σ, when a trial's $\sigma \geq 0.2$ seconds.

3.1 Parameters

We acknowledge the arguments on the stability of benchmarks presented in [2]. Thus, we take every precaution to create a uniform environment. For all experiments, the default priority was used for each system's kernel-level objects of execution. Experiments were performed on the same hardware, a four PPro SMP machine with 512 megabytes of RAM and a four-gigabyte fast and wide SCSI hard drive. At any one time, the machine solely operated under Solaris version 2.6 or Windows NT Server version 4.0. This was implement by a dual boot to facilitate easy switching between the OSs. Each operating system used its native file system. There were no

other users on the machine during experiments. The "same compiler", GNU gcc version 2.8.1, was used to compile the test programs for both operating systems. Originally, this was done to reduce any variances in execution time that could be attributed to different compilers. However, later we compiled the test programs with each system's native compiler (Visual C++ 5.0 and SUN C Compiler 4.2) and found no significant differences in performance. In order to maintain a standard format, we chose to only report the results from the gcc compilations. Note, all test programs were ˝compiled with the options*: -O3 -mpentiumpro -march=pentiumpro*. These options generate the highest level of performance optimizations for a Pentium Pro.

3.2 Thread Creation

The first experiment measured the maximum ˝number of kernel-level objects of execution each operating system could create, since neither system clearly documents the limit. The experiment was performed by repeatedly creating threads (bound in the Solaris case) that suspended after creation. At some given point, the tested operating system would fail trying to create a thread because it had exhausted resources or had reached the preset limit.

Description	NT	Solaris
# of Threads Created	9817	2294
Memory Usage	68MB	19MB
Execution Time (sec.)	24.12	2.68

Table 1: Comparison of the maximum number of threads allowable.

Table 1 shows the test program executing on Solaris failed after 2294 requests to create bound threads. At the time of termination, the process had only used 19 megabytes of memory. The Windows NT test program created 9817 threads before failing. At that time, it had used approximately 68 megabytes of memory. In addition, the table shows the amount of time required to create the threads.

The second experiment, shown in Table 2, measured the speed of thread creation on both systems. The table shows Solaris bound threads and NT threads had similar performances. The similar performance can be attributed to the fact that each OS required system calls for thread creation. In the case of Solaris, this was done indirectly through the ˝thread library. The library was required to make a system call to create an LWP for each bound thread. In addition as expected, Solaris's unbound thread creation outperformed NT's. In this case, Solaris's thread creation required a simple library call, while NT's required a system call. It is also worth noting that the Solaris restricted concurrency case (CL=4) was only marginally slower than the Solaris unbound case. This was because only four LWPs were needed. Thus, only four system calls were required to create the threads.

Threads	NT Time	Solaris Time		
		Bound	CL=4	Unbound
100	0.11	0.08	0.07	0.06
200	0.17	0.15	0.11	0.09
500	0.37	0.37	0.24	0.22
1000	0.74	0.81	0.49	0.45
2000	1.90	2.07	1.12	0.98

Table 2: Comparison of thread creation speed.

The third experiment also involved the creation of threads. The experiment measured the speed of thread creation while other threads were executing CPU intensive tasks. The tasks included several integer operations such as addition, subtraction, and multiplication. This imposed on each OS to create threads on a heavily loaded system. The number of threads created was varied. Table 3 shows how long it took to create a collection of threads in each system.

Threads	NT		Solaris	
	Time	σ	Time	σ
16	3.65	0.29	0.33	0.04
32	5.72	0.34	0.52	0.14
64	12.56	0.43	0.91	0.22
128	146.74	18.77	1.98	0.39

Table 3: Comparison of the performance of processes that create CPU intensive threads.

The experiment showed that the Solaris version of the test program created threads much faster than the NT version. This can be attributed to each systems multi-tasking scheduling algorithm. Although, the algorithms are similar in design, priority differences exist. Solaris's algorithm was fair with respect to newly created LWPs, while NT scheduling algorithm gave priority to currently executing threads. Thus in

the case of NT, requests for thread creation took longer because of the heavily loaded system. We found this to be characteristic of NT's scheduling algorithm. In various cases, executing several CPU-bound threads severely reduced the responsiveness of the system. Microsoft documents this behavior in [7]. Also, in both the Solaris and the NT cases, as the number of threads increased, the thread creation time became less predictable. This was especially true in the NT case, σ = 18.77 seconds when 128 threads were used.

3.3 No Synchronization

The fourth experiment determined how each operating system's thread implementation would perform on processes that created CPU intensive threads (with identical workloads) that did not require any synchronization. The experiment was performed by executing a process that created threads, where each thread had a task to perform that required a processor for approximately 10 consecutive seconds. A thread would perform its task and then terminate. After all the threads terminated, the creating process would terminate. Table 4 shows how long it ˝took processes to complete in each system.

Threads	NT Time	Solaris Time		
		Bound	CL=4	Unbound
1	10.11	10.06	10.06	10.06
4	10.13	10.13	10.12	40.18
8	20.32	20.53	20.26	80.35
16	40.37	40.35	40.52	160.67
32	80.49	80.80	80.73	321.27
64	160.78	161.34	161.49	642.54

Table 4: Comparison of the performance of processes with CPU intensive threads that do not require synchronization.

The experiment showed few differences in performance between NT threads and Solaris bound threads. This suggests that Solaris bound threads are similar to NT threads while performing CPU intensive tasks that did not require synchronization. However, it is worth noting that as the number of CPU intensive threads increased, Windows NT's performance ˝was slightly better.

In Solaris's unbounded and CL=4 cases, the thread library did not increase nor decrease the size of the LWP pool. Therefore, only one LWP was used by the library for the unbounded case. Consequently, the unbound threads took approximately 10*N* time, where *N* was the number of threads used. (Recall each thread performed a 10 second task.) The performance was also reflective of the FIFO algorithm used by library. Another point worth noting is that in Solaris CL=4 case, the performance was equivalent to that of the bound threads, which were optimal. Thus, additional LWPs did not increase the performance. This leads to two observations. First, in the case of equal workloads with no synchronization, peek performance is reached when the amount of LWPs is equal to the number of processors. Second, the time it takes Solaris's thread library to schedule threads on LWP is not a factor in performance.

3.4 Extensive Synchronization

The fifth experiment determined how each operating system's thread implementation would perform on processes that use threads (with identical workloads), which require extensive synchronization. The test program was a slightly altered version of an example from [1] called "array.c". The test program created a variable number of threads that modified a shared data structure for 10000 iterations. Mutual exclusion was required each time a thread needed to modify the shared data. In the general case, this can be implemented with a mutual exclusion object, like a mutex. Both operating systems offer local and global mutual exclusion objects[2]. Windows NT provides two mutual exclusion objects, a *mutex*, which is global, and a *critical section*, which is local. Solaris only provides a *mutex*. However, an argument can be passed to its initialization function, to specify its scope. We thus chose to compare each system's local and global mutual exclusion objects. Tables 5 and 6 shows the execution times for processes to complete in each system.

The results show NT out performs Solaris when using local synchronization objects, while Solaris out performs NT when using global synchronization objects. In addition, the experiment showed large differences in the performance of NT's mutex in comparison to its critical section, and few differences in performance of Solaris local mutex in comparison to its global mutex. The poor performance of NT's mutex was directly attributed to its implementation.

[2] This refers to the scope of the synchronization object, where local refers to a process scope and global refers to a system scope.

NT's mutex is a kernel object that has many security attributes that are used to secure its global status. NT's critical section is a simple user object that only calls the kernel when there is contention and a thread must either wait or awaken. Thus, its stronger performance was due to the elimination of the overhead associated with the global mutex.

The Solaris case CL = 4 outperformed both bound and unbound Solaris cases. This was due to a reduction in contention for a mutex. This reduction was caused by the LWP pool size. Since the pool size was four, only four threads could execute concurrently. Consequently, only four threads could contend for the same mutex. This reduced the time threads spent blocked, waiting for a mutex. Furthermore, when a thread was blocked, the thread library scheduled another thread on the LWP of the blocked thread. This increased the throughput of the process.

Threads	NT Time	Solaris Time		
	Critical Section	Local Mutex		
		Bound	CL=4	Unbound
250	1.04	1.20	1.13	2.69
500	2.49	2.93	2.56	5.98
750	3.76	5.16	4.37	9.63
1000	4.93	8.18	6.43	13.89
2000	9.89	24.85	17.84	35.38

Table 5: Comparison of the performance of processes with threads that required extensive synchron-ization using local/intra-process syncronization ob-jects.

Threads	NT Time	Solaris Time		
	Mutex	Global Mutex		
		Bound	CL=4	Unbound
250	10.84	1.18	1.20	2.68
500	25.58	2.69	2.74	5.94
750	37.78	4.80	4.60	9.59
1000	49.73	7.79	6.95	13.73
2000	99.15	24.98	19.75	34.89

Table 6: Comparison of the performance of processes with threads that require extensive synchronization using global/inter-process synchronization objects.

3.5 Parallel Search

The sixth experiment determined how each operating system's thread implementation would perform on the execution of a parallel search implemented with threads that required limited synchronization. Here we explored the classic symmetric traveling salesman problem (TSP). The problem is defined as follows:

> Given a set of *n* nodes and distances for each pair of nodes, find a roundtrip of minimal total length visiting each node exactly once. The distance from node *i* to node *j* is the same as from node *j* to node *i*.

The problem was modeled with threads to perform a parallel depth-first branch and bound search. For background information on parallel searches, see [6]. The threads were implemented in a work pile paradigm, see Chapter 16 in [5]. The work pile contained equally sized partially expanded branches of the search tree. The threads obtained partially expanded branches from the work pile and fully expanded them in search of a solution. The initial bound of the problem was obtained by a greedy heuristic, see [8]. For testing purposes, the heuristic always returned the optimal solution. Therefore, it was the task of the threads to verify the optimality of the heuristic. Synchronization was required for accessing the work pile and for solution updates. Yet, recall the previous experiment showed that NT's mutex performed poorly when extensive synchronization was required. This leads one to believe that a critical section should be used for NT. However, after thorough testing, it was found that, synchronization occurred infrequently enough that it ˝could be implemented by using mutexes without any loss in performance as compared to a critical section. We still chose to report our results using a critical section for NT. In the case of Solaris, a mutex with local scope was used. The data, gr17.tsp with n = 17, were obtained from the TSPLIB at Rice University [17]. Table 7 shows how long it took to verify optimality using processes in each system.

Threads	NT		Solaris					
			Bound		CL=4		Unbound	
	Time	σ	Time	σ	Time	σ	Time	σ
1	149.86	0.05	152.05	0.01	152.08	0.04	152.08	0.02
2	74.96	0.01	76.06	0.01	76.06	0.01	76.06	0.02
3	50.02	0.01	50.76	0.01	50.74	0.01	76.29	0.22
4	37.59	0.07	38.20	0.04	38.17	0.04	76.37	0.33
8	37.90	0.26	38.36	0.24	38.15	0.02	76.35	0.34
16	38.17	0.24	38.78	0.21	38.18	0.04	76.97	0.29

Table 7: Comparison of the performance of the TSP using threads to perform a parallel depth-first branch and bound search (Data: gr17.tsp, $n = 17$).

The NT version of the TSP slightly outperformed the Solaris version. Both systems were able to achieve an almost linear speed up (3.9+). The Solaris benchmarks again showed that when the LWP pool size was four the performance was equivalent to using four bound threads. Another observation was that when using two or more of Solaris's unbound threads the performance was equal to using two of Solaris's bound threads. This would suggest that the thread library used two LWPs although two LWPs were not requested. This is the only experiment where Solaris's thread library changed the size of the LWP pool.

3.6 Threads With CPU Bursts

The last experiment determined how each operating system's thread implementation would perform on processes that had many threads with CPU bursts. This is analogous to applications that involve any type of input and output (I/O), e.g., networking or client/server applications, such as back end processing on a SQL server. The experiment was performed by executing a process that created many threads. Each thread would repeatedly be idle for one second and then require the CPU for a variable number of seconds. Three burst lengths were explored, one less than the idle time (0.5 sec.), one equal to the idle time (1.0 sec.) and one greater than the idle time (4 sec.). The amount of required CPU time causes the threads to act as if they are I/O-bound, equally-bound, or CPU- bound, respectively. Tables 8 – 10 show how long it took to complete the processes in each system.

Threads	NT		Solaris					
			Bound		CL=4		Unbound	
	Time	σ	Time	σ	Time	σ	Time	σ
1	7.58	0.01	7.53	0.00	4.99	0.04	4.96	0.13
4	7.58	0.01	7.58	0.05	4.97	0.15	5.93	0.12
8	9.17	0.19	8.94	0.10	7.23	0.28	10.82	0.23
16	12.25	0.30	12.90	0.14	10.97	0.24	20.88	0.16
32	21.47	0.03	21.54	0.11	20.93	0.10	41.11	0.11
64	41.70	0.02	41.80	0.05	40.97	0.50	81.57	0.11

Table 8: Comparison of the performance of processes with threads that require the CPU for intervals that are less than their idle time (I/O-Bound).

Threads	NT		Solaris					
			Bound		CL=4		Unbound	
	Time	σ	Time	σ	Time	σ	Time	σ
1	10.08	0.01	10.05	0.02	9.96	0.13	9.94	0.21
4	10.08	0.01	10.13	0.08	9.95	0.12	10.99	0.11
8	14.17	0.18	13.24	0.16	11.11	0.35	20.92	0.05
16	22.27	0.06	22.11	0.25	20.95	0.11	40.99	0.04
32	41.92	0.05	41.58	0.06	41.09	0.28	81.15	0.13
64	81.82	0.03	82.13	0.12	81.81	0.36	162.39	0.20

Table 9: Comparison of the performance of processes with threads that require the CPU for intervals that are equal to their idle time (Equally-Bound).

Threads	NT		Solaris					
			Bound		CL=4		Unbound	
	Time	σ	Time	σ	Time	σ	Time	σ
1	25.15	0.02	25.10	0.01	24.98	0.07	25.00	0.00
4	25.16	0.01	25.33	0.17	24.98	0.08	40.99	0.08
8	43.36	0.25	42.50	0.23	42.06	0.60	81.04	0.11
16	82.25	0.06	82.25	0.31	81.78	0.48	108.84	0.27
32	162.25	0.04	162.63	0.14	162.68	0.39	237.67	0.36
64	322.64	0.01	323.67	0.31	323.81	0.31	431.44	0.44

Table 10: Comparison of the performance of processes with threads that require the CPU for intervals that are greater than their idle time (CPU-Bound).

The experiments showed a few differences in the performance between Solaris's bound threads, Solaris's threads with restricted concurrency and NT's threads. A noticeable difference in performance occurred in the first two cases, shown in Tables 8 and 9, where the threads required the CPU for intervals that were less than or equal to their idle time. In these cases, the Solaris version using restricted concurrency showed a slightly better performance in comparison to NT's threads or Solaris bound and

unbound threads. This can be directly attributed to Solaris's two-tier system. In this case, it was shown that optimal performance could be achieved by setting the concurrency level to the number of CPUs and creating as many unbound ˝threads as needed. This logically creates one LWP for each CPU. Recall the operating system is only aware of the LWPs. This coupled ˝with the FIFO scheduling of Solaris's thread library keeps its bookkeeping to a minimal while maximizing the concurrency.

There were also notable differences in performance in the last case, Table 10, where the CPU intervals were greater than the idle time, CPU-bound. The results of Solaris's bound threads and ˝NT's threads were similar to the fourth experiment, Section 3.3, Table 4. NT's threads out performed Solaris's bound threads as the number of threads increased.

4. Conclusions

Both Windows NT and Solaris were able to utilize multiprocessors. Their performance scaled well with the number of CPUs. However, there is a lack of documentation pertaining to the performance ˝issues of each system. Microsoft and Sun have taken steps in the right direction with the availability of documentation at their respective web sites [7] and [18]. However, little is written on the performance impact of each design. Yet, we found that each implementation can have significant performance implications on various types of applications.

The experiments showed that Windows NT's thread implementation excelled at CPU intensive tasks that had limited synchronization or only intra-process synchronization. Therefore, NT's threads can be greatly exploited on applications such as parallel computations or parallel searches. The experiments also showed that NT's mutex performed poorly compared to Solaris's mutex, when extensive synchronization was required. However, NT's critical section provided significantly better performance than Solaris's mutex. Therefore, for NT, a critical section should be used to implement extensive intra-process synchronization. Another NT observation was that to achieve optimal performance the number of threads used by a process for the execution of a parallel search or computation should be approximately equal to the number of CPUs. Although, it was found that the exact number of threads was dependent on the specific problem, its implementation and the specific data set being used, also see [6]. It is also worth noting, that both systems grew erratic as the number of executing CPU intensive threads grew larger than the number of processors. This was especially true in the NT case. Responsiveness was sluggish on heavily loaded systems and often required dedicated system usage.

Solaris's thread API proved to be more flexible, at the cost of complexity. We found that the exploitation of multiprocessors required a thorough understanding of the underlying OS architecture. However, we also found Solaris's two-tier design to have a positive performance impact on tasks with bursty processor requirements. This suggests that Solaris threads are well suited for applications such back end processing or client/server applications, where a server can create threads to respond to a client's requests. In ˝addition, we found the Solaris thread library's automatic LWP pool size control to be insignificant. We found in most cases, the programmer can achieve desirable performance with unbound threads and a restricted concurrency level that is equal to the number of processors.

In conclusion, the advent of relatively inexpensive multiprocessor machines has placed a critical importance on the design of mainstream operating systems and their implementations of threads. Threads have become important and powerful indispensable programming tools. They give programmers the ability to execute tasks concurrently. When used properly they can dramatically increase performance, even on a uniprocessor machine. However, threads are new to mainstream computing and are at a relatively early stage of development. Thus, arguments exist on how threads should be implemented. Yet, one should remember that differences between implementations are simply tradeoffs. Implementers are constantly trying to balance their implementations by providing facilities they deem the most important at some acceptable cost.

Note there has been a movement to standardize threads. IEEE has defined a thread standard POSIX 1003.1c-1995 that is an extension to the 1003.1 Portable Operating System Interface (POSIX). The standard, called *pthreads,* is a library-based thread API. It allows one to develop thread applications cross platform. However, IEEE does not actually implement the library. It only defines what should be done, the API. This leaves the actual implementation up to the operating system developer. Usually

the pthreads library is built on the developer's own thread implementation. It is simply a wrapper over the developers' own implementation and thus, all features may or may not exist. In the case where the OS does not have a thread implementation, the library is solely user based, and thus can not exploit multiprocessors.

5. Acknowledgements

This research is supported in part by ONR grant N00014-96-1-1057.

References

[1] Berg, D.J.; Lewis, B.: *Threads Primer: A Guide to Multithreaded Programming*, SunSoft Press, 1996

[2] Collins, R.R.: Benchmarks: Fact, Fiction, or Fantasy, *Dr. Dobb's Journal*, March 1998

[3] El-Rewini, H.; Lewis, T.G.: *Introduction to Parallel Computing*, Prentice Hall, 1992

[4] Intel: The Pentium Pro Processor Performance Brief at : http://www.intel.com/procs/perf/highend/

[5] Kleiman, S.; Shah, D.; Smaalders, B.: *Programming With Threads*, SunSoft Press, 1996

[6] Kumar, V.; Grama, A.; Gupta, A.; Karypis, G.: *Introduction to Parallel Computing: Design and Analysis of Algorithms*, Chapter 8, Benjamin Cummings, 1994

[7] Microsoft: Microsoft Developer Network, http://www.microsoft.com/msdn

[8] McAloon, K.; Tretkoff, C.: *Optimization and Computational Logic*, Wiley, 1996

[9] Prasad, S.: Weaving a Thread, *Byte*, October 1996

[10] Richter, J.: *Advanced Windows NT: The Developer's Guide to the Win32 Application Programming Interface*, Microsoft Press, 1994

[11] SunSoft: Multithreaded Implementations and Comparisons, A White Paper, 1996

[12] SunSoft: Multithread Programming Guide, A User Manual, 1994

[13] SunSoft: Solaris SunOS 5.0 Multithread Architecture, A White Paper, 1991

[14] Tanenbaum, A.S.: *Distributed Operating Systems*, Prentice Hall, 1995

[15] Tomlinson, P.: Understanding NT: Multithreaded Programming, *Windows Developer's Journal*, April 1996

[16] Thompson, T.: The World's Fastest Computers, *Byte*, January 1996

[17] TSPLIB: http://softlib.rice.edu/softlib/tsplib/

[18] http://www.sun.com/workshop/threads/