Implementing a Dynamic Processor Allocation Policy for Multiprogrammed Parallel Applications in the Solaris$^{\text{TM}}$ Operating System

Kelvin K. Yue
kyue@eng.sun.com
Sun Microsystems Inc.
901 San Antonio Road, MS MPK17-302
Palo Alto, CA 94303

David J. Lilja
lilja@ece.umn.edu
Department of Electrical and Computer Engineering
Minnesota Supercomputing Institute
University of Minnesota
Minneapolis, MN 55455
(612) 625-5007

September 24, 1998

Abstract

Parallel applications typically do not perform well in a multiprogrammed environment that uses time-sharing to allocate processor resources to the applications' parallel threads. Coscheduling related parallel threads, or statically partitioning the system, often can reduce the applications' execution times, but at the expense of reducing the overall system utilization. To address this problem, there has been increasing interest in dynamically allocating processors to applications based on their resource demands and the dynamically varying system load. The *Loop-Level Process Control (LLPC)* policy [16] dynamically adjusts the number of threads an application is allowed to execute based on the application's available parallelism and the overall system load. This study demonstrates the feasibility of incorporating the LLPC strategy into an existing commercial operating system and parallelizing compiler and provides further evidence of the performance improvement that is possible using this dynamic allocation strategy. In this implementation, applications are automatically parallelized and enhanced with the appropriate LLPC hooks so that each application interacts with the modified version of the Solaris operating system. The parallelism of the applications are then dynamically adjusted automatically when they are executed in a multiprogrammed environment so that all applications obtain a fair share of the total processing resources.

Keywords: multiprogramming; process scheduling; processor allocation; operating system; parallelizing compiler; shared-memory multiprocessor; performance measurement.

1 Introduction

Fairly allocating processors to the threads of parallel application programs in a multiprogrammed shared-memory multiprocessor is necessary to minimize the execution time of each simultaneously executing application while ensuring high overall system throughput. Traditional time-sharing, such as that implemented in most Unix-based operating systems, typically does not work well to share the processing resources due to high context switching overhead, poor cache utilization, inefficient locking and synchronization, and other related problems [5, 8, 9, 13, 18].

Coscheduling [9] or gang scheduling can solve some of these problems by allocating processors to threads as a group rather than individually. It has the advantage that it can be implemented on top of the time-sharing used in existing operating systems [3]. Statically dividing the system into multiple partitions is another simple approach for allocating processors to parallel applications. While these approaches can improve the execution time of an individual application, they also tend to reduce the system utilization. Dynamically partitioning the system based on the system load is one of the new approaches that can improve the parallel applications' performance in a multiprogrammed environment while maintaining high system utilization [1, 6, 11, 13, 16].

This paper extends our previous work on dynamic processor allocation [15, 16, 17] by implementing and analyzing a strategy called *Loop-Level Process Control* (LLPC) with Sun Microsystems' Solaris operating system and related parallelizing compiler. This paper addresses the feasibility of incorporating a dynamic allocation strategy into an existing commercial operating system and compiler and discusses the related design tradeoffs and novel techniques used in the implementation. Finally, the performance of this strategy is studied using applications from the SPEC95 benchmark suite.

The remainder of the paper is organized as follows: Section 2 provides further background information on the multiprogrammed multiprocessor scheduling problem and describes existing solutions. Our programming model and system architecture are also described. Section 3 describes the LLPC processor allocation strategy and the implementation details. The experimental results are presented in Section 4. The final section concludes the paper.

2 Background

This section discusses the targeted system architecture and the corresponding programming model. It also provides additional details on the performance issues in multiprogrammed multiprocessor systems and reviews existing strategies.

2.1 System Architecture and Programming Model

The system used in this study is a shared-memory multiprocessor in which the memory is equally accessible to all of the processors. Since these systems have a programming model that is similar to traditional uniprocessor systems, shared-memory multiprocessors have been widely commercialized and are often used as general-purpose high-performance compute servers. In this type of system, all processors run at the same speed with each of the processors executing its instructions independently of the others. Most shared-memory systems also have a local cache memory in the processor module to improve the average memory access time.

One of the most common programming models for shared-memory multiprocessor systems is the loop-level parallelized model. A programmer or compiler parallelizes an application with this model by recognizing the independent sections of the application, which usually are the iterations of a loop, and partitioning them into parallel tasks that can be executed concurrently [7]. With this programming model, the parallelism of many existing application programs can be exploited automatically by a sophisticated compiler without rewriting the sequential code. Although this approach might not be able to exploit the maximum inherent parallelism in an application, it can increase the portability of the code and can lessen the programmer's burden in developing new parallel applications.

Multiple threads are used to execute a loop-level parallelized application. Threads share the application's instructions and most of its data, but each thread has its own program counter and its own stack for storing local data. When a parallelized application begins to execute, a number of threads, usually equal to the number of physical CPUs, are created. During the execution of the sequential section of the application, only one thread is used with the remainder waiting idly. When a parallel section is reached, all of the threads are used with each obtaining its share of the available work according to some parallel loop scheduling algorithm [7]. After the parallel work is completed, all the threads are synchronized and one thread continues to execute the sequential section of the code until the next parallel section is encountered.

2.2 Processor Scheduling

Threads are the scheduling entities in the operating system. With time-sharing in the Unix operating system [2], threads that are ready for execution are put into the *ready queue*. Whenever a processor is idle, it removes the first thread from the ready queue and begins executing the task. If the task is not completed before the *time quantum* expires, the thread is returned to the end of the ready queue. The processor then begins executing the next thread from the beginning of the queue.

The Solaris operating system uses multiple queues to enhance scalability. Moreover, it has a two-level thread model with *user-level threads* and *kernel threads*. Kernel threads are the scheduling entities in the operating system. Multiple user-level threads can *share* a single kernel thread [10]. For the loop-level parallel programming model used in this study, a one-to-one mapping is used in which a single user-level thread is *bound* to a kernel thread.

Although time-sharing produces good system utilization by multiplexing the processors among several applications, it can significantly degrade the performance of parallel applications compared to their execution time on a dedicated system. Studies have shown that the performance is affected due to context switching overhead, cache corruption, inefficient locking and synchronization, and other related effects [5, 8, 9, 13, 18]. For example, running threads might be waiting for results that will be produced by other threads that are in the queue waiting for processors on which to run.

A common solution for this problem is to allocate processors to all the threads of an application at the same time [9]. As a result, the active threads will not be waiting for some queued threads. Instead of scheduling the threads of an application independently whenever there are idle processors, this *coscheduling* strategy schedules the threads as a group. The IRIX™ operating system from Silicon Graphics provides a coscheduling strategy called *gang scheduling* [3]. The operation of this gang scheduling depends on the information contained in a *gang control block*, which is created when the related threads from the same application are created. The gang control blocks of the applications form a *gang queue*. The scheduler, using the information from the gang control blocks, schedules these related threads to execute during the same time quantum.

Another possible solution is to divide the processors into several independent partitions and then execute each application in its own partition. By using only a portion of the system, this type of *space-sharing* or *static partitioning* eliminates the competition between applications for processors. The disadvantage is that idle processors in one partition are not accessible to an application running in a different partition. Both Solaris and IRIX have a feature called *processor set* in which a subset of the processors is statically allocated to the applications.

Coscheduling and space-sharing perform better than time-sharing by minimizing the interference between applications. Another advantage of these strategies is that they can be easily incorporated into existing operating systems. However, the performance of coscheduling can still be affected by excess context switching while space-sharing suffers from poor system utilization [13].

To improve the system utilization while minimizing the number of context switches, the system can be *dynamically partitioned*. With this strategy, the sizes of the partitions on which the applications run are adjusted based on the system load. When there are few applications so that the system load is light, the system will have fewer partitions. When the load is high, the system will have more partitions with fewer processors in each. This strategy has the advantages of static partitioning in that interference between applications is minimized, and it improves the overall system utilization.

Although they are not targeted specifically for loop-level parallelized applications, *scheduler activation* [1] and *process control* [13] are two strategies that provide dynamic partitioning for multithreaded applications. In both of these strategies, the kernel communicates changes in the system load to the user-level thread library. The thread library then reacts to the changes by adjusting the number of runnable threads.

For loop-level parallelized applications, the *Distributed Resource Management System* (DRMS) [6] from IBM, *Automatic Self-Allocating Threads* (ASAT) [11], and *Loop-Level Process Control* (LLPC) [16] are some of the approaches that dynamically adjust the parallelism of the applications based on the system load. DRMS is designed for SPMD Fortran applications executed on distributed-memory architecture systems. It repartitions and redistributes the data of the application based on user or compiler inserted directives when processor availability changes. The job scheduler and various components in the system make the re-partitioning decision and coordinate with the user application on the change.

ASAT and LLPC, on the other hand, are designed for shared-memory architectures and require the applications to make the adjustment decisions. ASAT-enabled applications occasionally measure the time required to synchronize threads, which provides an indirect indication of the system load. The application then creates or suspends threads based on this measurement [11]. LLPC-enabled applications use user-level shared-memory to communicate with each other about the processing resource requirements of each of their parallel loops [16]. Based on this load information, applications create a suitable number of threads for the execution of each parallel loop.

Our previous work has shown that using LLPC to dynamically adjust the number of runnable threads in the system has the advantage of reducing the contention for the processing resources which in turn improves the performance of the applications [15]. The fundamental idea underlying LLPC is that, by controlling the number of threads an application is allowed to create based on the system load and the application's available parallelism, the application will be able to utilize as many processors as possible without overloading the system. When the system load is high, LLPC reduces the context switching rate by allowing the applications to create only a small number of threads instead of the maximum number of threads that they may like to create. As a result, execution can still proceed for all applications while no single application monopolizes all of the processors. When the system load is light, however, a highly parallel application is allowed to utilize all of the idle processors.

In the reminder of this paper, we present our integration of LLPC into the Solaris operating system and parallelizing compiler. We also compare its performance with that of the time-sharing and static partitioning strategies already available in Solaris.

3 LLPC Implementation in Solaris

The implementation of loop-level process control requires both kernel level support and enabling the compiler's microtasking library to use LLPC. This section describes each of these parts in detail.

3.1 Kernel Support

The original LLPC implementation on the SGI Challenge system was done entirely at the user level [16]. LLPC-enabled applications communicated their load information with each other through the use of shared memory. Based on this information, they adjusted the number of threads used to execute each parallel loop. There are several advantages to this type of user-level implementation. First, it is simple and straight-forward. Since the load information is maintained by the applications themselves, no kernel modifications were necessary. Second, the load information is *accurate*. That is, all LLPC-enabled applications know exactly how many LLPC-enabled threads are active in the system at all times.

On the other hand, *accuracy* in this case is a relative term. Since the load information is maintained by the LLPC-enabled applications, non-LLPC applications do not update this information. Therefore, the LLPC-enabled applications do not know the load of the whole system–they know about only the LLPC load. Another issue is that, although the LLPC-enabled applications accurately report the number of threads they create, some of these threads might be blocked and so do not require any processing resources. Thus, the actual system load might be lower than indicated by simply counting the number of threads created.

One way to solve the load information problem is to have a daemon process periodically check the overall system load and update an appropriate data structure. Then the LLPC-enabled applications can use this more accurate information to determine how many threads to activate. In fact, the overall system load information is already being maintained by the Solaris kernel in 1-minute, 5-minute, and 15-minute averages, which can be seen using the Unix `uptime` command. However, this information is too coarse for LLPC. A low overhead mechanism is needed to obtain finer-grained system load information from the kernel.

Fortunately, the latest version of Solaris (version 2.6) has a new feature called *scheduling control* [14]. This feature is designed to provide an efficient mechanism for the kernel and the user-level applications to share scheduling information such as the execution state of a thread and the identification number of the last CPU on which it ran. Threads also can provide some extra information to assist the kernel in making scheduling decisions. One of the primary advantages of this new feature over the usual means of communication, such as system calls and signals, is its low overhead.

Scheduling control uses physical pages as the communication medium to achieve low overhead. When an application invokes scheduling control the first time, a kernel page is mapped into the application's address space and locked into physical memory. When there is a change in the status of a thread, the kernel puts the updated information into this page with a simple store instruction. Similarly, at the user level, the thread obtains this information simply by reading the page. Since the page is locked into the physical memory, page faults will never occur.

The current implementation of scheduling control provides information on only a per-thread basis with no overall system information. However, because of its low overhead and the already-implemented API, we extended scheduling control to include the needed information. When a thread makes a call to the scheduling control routine in our prototype, it obtains not only the scheduling information about itself, but also the total number of threads that are currently running on the CPUs and the total number of threads that are waiting on the run queues. This system load information is maintained with clock tick granularity, although it is updated to the page only when the scheduling control routine is invoked.

3.2 Compiler Support

In our previous work, LLPC calls were manually inserted into the applications [17]. At the beginning of each parallel loop, a call to the LLPC routine was made to determine the number of threads to be used for the execution of that loop. This number was then passed to the run-time library which distributed the parallel loop iterations to the threads. To eliminate the manual work of inserting LLPC routines, and to reduce their execution time overhead, LLPC routines have been incorporated into the run-time library of Sun's Fortran-77 and C parallelizing compilers [4].

This compiler automatically parallelizes Fortran-style DO loops. It uses many well-known techniques to determine which loops of a sequential application can be effectively parallelized and transforms them into tasks that can be executed concurrently by multiple processors. The execution of these tasks is facilitated by the *microtasking* run-time library. This library manages the threads and supports parallel loop scheduling algorithms such as guided self-scheduling and factoring [7]. The default scheduling strategy is static scheduling, in which the loop iterations are evenly distributed to the threads.

As an example, the compiler recognizes the following loop as one that can be executed in parallel.

```
DO I = 1, 1000
     A(I) = B(I) * C(I)
```

```
    END DO
```

To parallelize the loop, it replaces the loop with a subroutine call to the microtasking library and transforms the loop body itself into a parallel subroutine, as shown below.

```
    call dopar(parallel_subroutine_1234,...)

    ...

    subroutine parallel_subroutine_1234(begin_I, end_I)

       DO J = begin_I, end_I
          A(J) = B(J) * C(J)
       END DO
    end subroutine
```

When the application begins executing, the main thread, which is also called the *master thread*, creates a number of *slave threads*. The total number of threads (master plus slaves) is set to the number of physical processors, unless the user specifies a smaller value. When the master thread is executing a sequential section of the application, the slave threads wait idly. When a parallel section is reached, the master thread calls the `dopar` routine, which sets up the necessary data structures and unblocks all of the slave threads. Each thread then calculates its share of work and calls the subroutine that encapsulates the parallel loop body. When a thread finishes its share of iterations, it waits for all of the other threads to complete. At this point, the master thread continues with the execution of the subsequent sequential section while the slave threads return to the idle state.

Our prototype integrates the LLPC algorithm into the microtasking library to manage the slave threads using a call to the scheduling control routine at the beginning of the master thread routine. This call allows the master thread to obtain the latest system load information whenever a parallel loop is initiated. The master thread uses this system load information to determine how many threads should be used to execute that particular parallel loop, using the algorithm described in the next section.

3.3 Loop-Level Process Control Algorithm

The original microtasking library from Sun allows slave threads to be in one of two states, either *running* or *spinning*. In the *running* state, a thread is executing a chunk of work from a parallel loop and is thus using the CPU for useful work. When it is in the *spinning* state, it is executing an idle loop waiting for work. Therefore, the CPU resource is being wasted doing no useful work. The master thread is always in the *running* state either executing the application or managing its slave threads.

The LLPC algorithm incorporated into the microtasking library adds a third possible state for a slave thread–*sleeping*. In this state, the slave thread consumes no CPU resources until it is awakened by the master thread. This additional state provides a mechanism for LLPC-enabled applications to reduce the CPU resources they consume to thereby adjust the load they place on the system. Note that slave threads always start in the *spinning* state when the microtasking library creates them at the beginning of the application's execution.

The reason for adding a sleeping state but not putting the slave threads into this state whenever they are not running is because of the overhead incurred when awakening them. When a parallel loop is encountered and the slave threads are spinning, the slave threads can start executing work from the parallel loop instantly. If all of the slave threads are in the sleeping state, however, they need to be awakened before the parallel execution can begin, which can add a significant amount of time to the execution of the parallel loop.

When an LLPC-enabled application begins executing a parallel loop, its master thread obtains the current load information (`current_load`), which is defined to be the number of simultaneously running threads, through the scheduling control system call. It compares this latest information with the previous system load value. If the load has not changed, the parallel loop will be executed using the same number of threads as the last parallel loop that it executed. No adjustment is made when the system load is the same as the number of physical CPUs.

If the current load information obtained by the master thread is different than the previous value, the master calculates a new adjustment (`NumThreadsAdj`). First, it determines if the number of running threads in the system is higher than the number of CPUs. If it is, the system is overloaded and the master thread calculates how many of its currently spinning slave threads (`NumThreadsSpin`) should be put into the *sleeping* state to reduce the total number of active threads in the system to the number of CPUs.

Finally, if the number of running threads is lower than the number of CPUs, the CPU resources are not fully utilized. In this case, the master thread determines how many its *sleeping* slave threads it can awaken. This LLPC algorithm as implemented in the microtasking library is summarized below:

```
current_load = schedctl();

loadDiff = NumCPUs - current_load;
if ((loadDiff != 0) && (loadDiff != LastLoadDiff)){
        if (loadDiff > 0)   /* load is low */
        {
            /* No. of threads to wake up */
            NumThreadsAdj = NumThreadsCreated - NumThreadsSpin - 1;
            if (NumThreadsAdj > loadDiff)
                 NumThreadsAdj = loadDiff;
        }
        else  /* load is high */
        {
            /* No. of threads to go to sleep */
            if (- loadDiff > NumThreadsSpin)
               NumThreadsAdj = (- NumThreadsSpin);
            else
               NumThreadsAdj = loadDiff;
        }
    }
    LastLoadDiff = loadDiff;
```

To further enhance the performance of LLPC, we developed two new techniques for load adjustment. After the master thread determines the adjustment required, it must either put some slave threads to sleep or awaken some sleeping threads. Instead of applying the adjustment immediately, however, it is delayed. When the system is overloaded and spinning threads must be put to sleep, the master does not suspend the threads immediately. Instead, since the spinning slave threads are already ready to run, they are allowed to participate in the execution of the current parallel loop. A flag is set to notify the selected slave threads to go to sleep *after* completing their share of the parallel work.

On the other hand, when the system is underutilized, the master thread does not wait for sleeping threads to awaken before starting the parallel loop execution. Rather, it sends a signal to awaken the sleeping threads and then starts the parallel loop execution with the threads that are already in the spinning state. This approach masks the thread wake-up time with the execution of useful work. The newly awakened threads then go into the spinning state to later participate in the next parallel loop execution.

3.4 Example Execution Trace

To provide a better idea of how the various LLPC pieces work together, an example is shown in Figure 1. This diagram shows the execution of an application on a 4-CPU system using the LLPC strategy. When the application begins its execution, a master

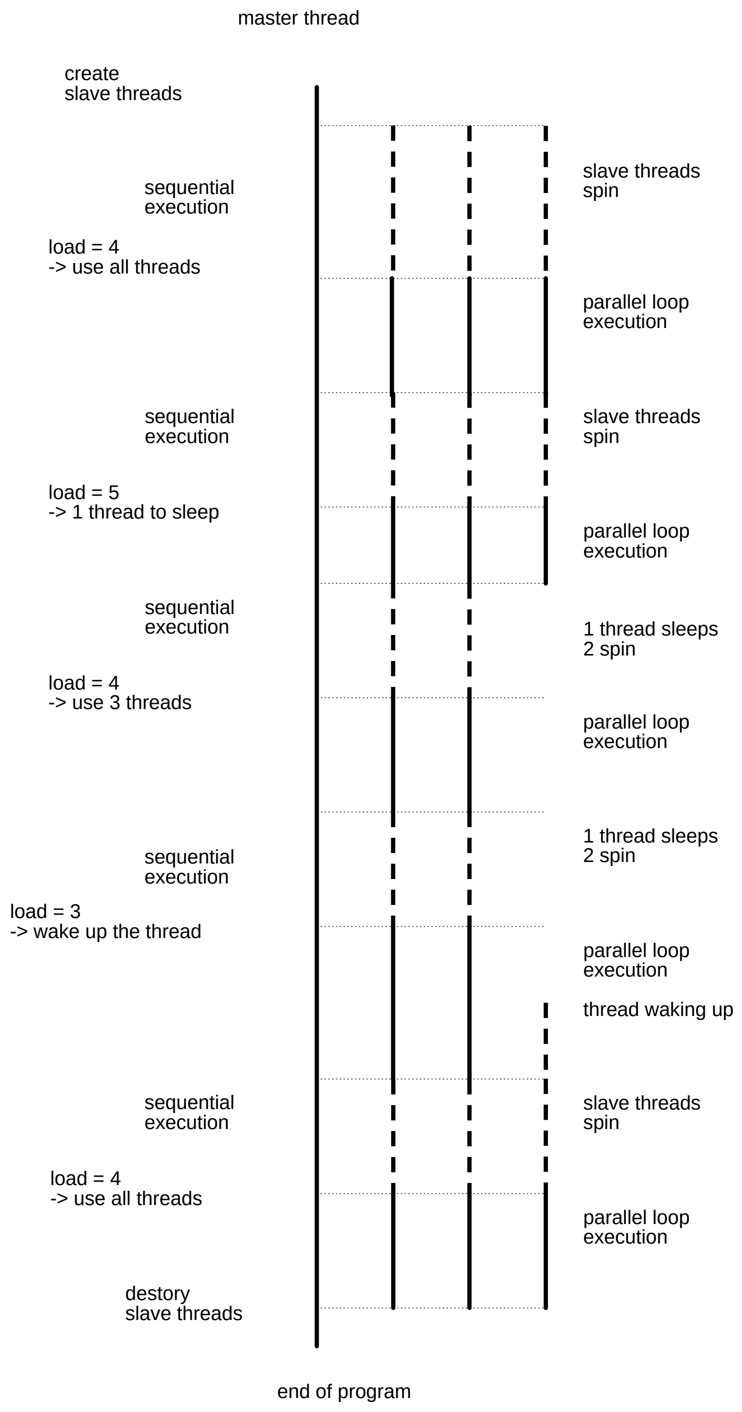


Figure 1: An example showing how LLPC changes the number of threads it uses to execute the parallel loops of an application program.

and three slave threads are created and the slave threads are put into the spinning state. At the beginning of the first parallel loop, the master thread determines that it is the only application using the system. Consequently, it uses all of the slave threads to execute the parallel loop. After the parallel execution, the slave threads return to the *spinning* state.

When the second parallel loop is reached, the master thread finds that the system load has increased due to other applications sharing the CPUs. The system load is at five, which is one more than the number of CPUs, so it decides to put one of its slave threads to sleep. The second parallel loop is still executed by four threads (one master and three slaves) but after the loop is executed, one of the slave threads goes into the *sleeping* state. At the third parallel loop, the system load remains unchanged, so only the slave threads that are in the *spinning* state are used for this loop execution.

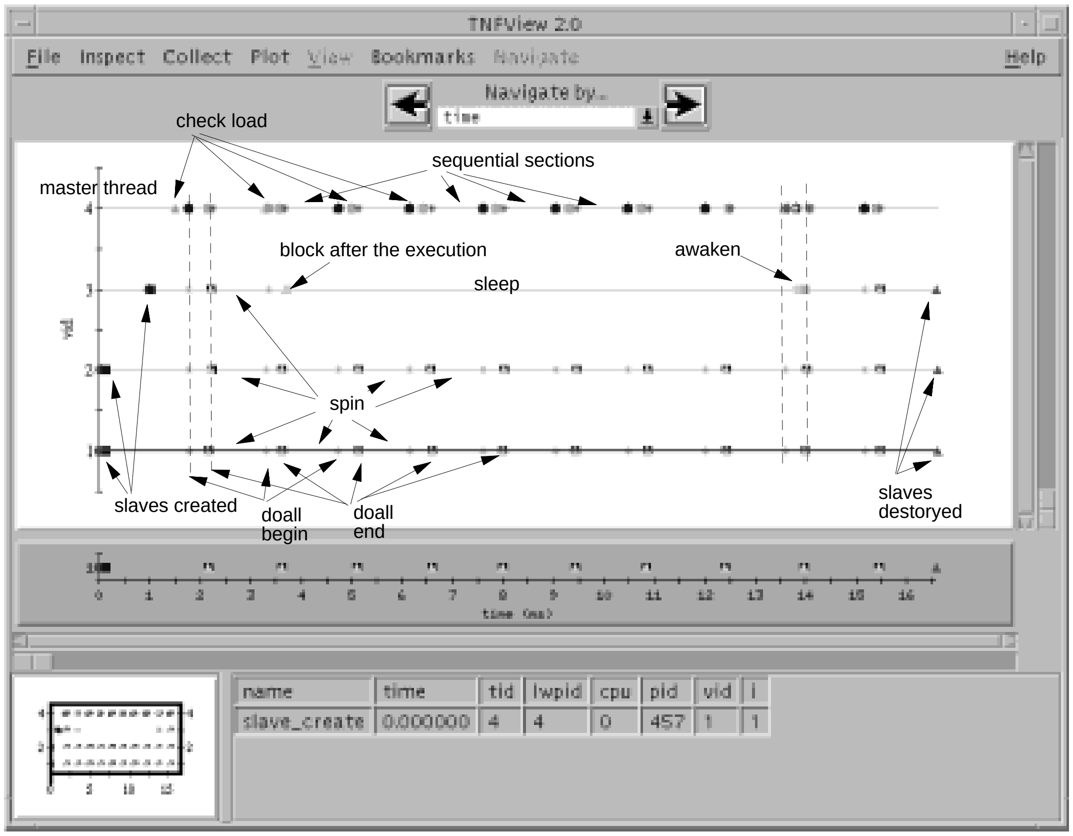
In the meantime, the other application terminates. When the master thread checks the load again at the beginning of the fourth loop, it decides that it can awaken one of its sleeping threads. After sending a signal to the sleeping thread, the master and the two spinning threads execute the fourth parallel loop. The sleeping thread eventually wakes up and goes into the *spinning* state. It can then participate in the execution of the next parallel loop.

4 Performance Evaluation

In this section, we compare the performance of our implementation of LLPC on Solaris with time-sharing and static partitioning using applications from the SPEC95 benchmark suite. First, however, we used the TNF (Trace Normal Form) tracing tool in Solaris to to verify that an LLPC-enabled application controls its slave threads as expected. TNF probes were inserted into the microtasking library. A synthetic benchmark consisting of ten sequential loops and ten parallel loops was then compiled and executed with this instrumented library. Figure 2, which is an annotated screen shot of the tracing output, shows that the LLPC-enabled application works as expected.

4.1 Experimental Environment

The system we used for this performance evaluation is a Sun Ultra Enterprise 6000 Server [12]. This single-bus shared-memory architecture system was equipped with four UltraSPARC processors running at a clock rate of 167 MHz. Each processing unit had 16 Kbytes of on-chip data cache and 16 Kbytes of on-chip instruction cache. The secondary cache was 512 Kbytes of unified instruction and data cache. The system had a total of 256 Mbytes of physical memory.



name	time	tid	lwpid	cpu	pid	vid	i
slave_create	0.000000	4	4	0	457	1	1

Figure 2: The execution trace of the synthetic benchmark program showing the state changes of the LLPC-controlled slave threads.

4.2 Benchmarks

The benchmarks we used for the performance comparison are four applications from the SPEC95 floating-point suite that have similar parallel run-times. The `mgrid` program is a multigrid solver in a 3D potential field while the `swim` program is a shallow water model using a 1024 x 1024 grid. The `su2cor` program computes the masses of elementary particles and `hydro2d` computes galactic jets by solving the hydrodynamical Navier-Stokes equations. All of the benchmark programs were parallelized using the standard Fortran compiler and linked with our modified microtasking library.

Table 1 summarizes the characteristics of these benchmarks. The first column shows the total number of loops in these programs. The second column is the number of these loops that were parallelized by the compiler, while the third column shows the number of times these parallelized loops were actually executed. The parallel speedup of the entire application, when compiled with the standard compiler, linked with the original microtasking library, and run by itself on the 4-processor system, is shown in the last column.

Table 1: Characteristics of the selected SPEC95 floating-point benchmark programs.

Application	Total no. of loops	No. of parallel loops detected	Dynamic parallel loop counts	Speedup on a dedicated system
swim	21	11	10262	3.51
su2cor	134	35	190932	2.39
hydro2d	150	65	76377	2.97
mgrid	53	10	58225	3.72

4.3 LLPC Overhead

To determine the execution time overhead of adding the LLPC calls to the benchmark programs, they were linked with the enhanced version of the microtasking library and run individually on the test system with the modified kernel. Table 2 shows the average run-time of three executions of the benchmarks. The run-times of the LLPC-enabled benchmarks are slightly higher than the unmodified programs since, when LLPC is enabled, the applications become sensitive to the unavoidable background load caused by the various system daemon processes. However, the overhead effect is quite small. The benchmarks were able to use all of the CPUs for executing their parallel loops most of the time, as shown by the average number of threads used per loop being almost equal to the number of physical CPUs.

Table 2: The overhead due to adding the LLPC calls to the test programs.

Application	original run-time (sec)	LLPC-enabled run-time (sec)	average no. of threads used/loop
swim	172	174.67	3.999
su2cor	166	166.33	3.999
hydro2d	241	241.0	3.996
mgrid	247	249.0	3.999

4.4 LLPC Compared to Time-Sharing

Next we compare the effectiveness of LLPC to time-sharing with a parallelized application and a sequential application executing concurrently. In this experiment, a synthetic sequential application was used that repeatedly executed for 18 seconds then slept for 5 seconds. This benchmark was executed simultaneously with the individual SPEC benchmarks. A corresponding *slowdown factor* for the parallel application was then calculated as follows:

$$\text{slowdown} = \frac{\text{measured run-time with sequential application running}}{\text{parallel run-time on a dedicated system}}.$$

A slowdown of one means that the execution time of the parallel application was not affected by the sequential application while a slowdown factor of two would mean that the parallel application executed for twice as long as it did on a dedicated system.

As shown by the slowdown factors in Table 3, the performance of the LLPC-enabled applications are not affected by the sequential load as much as those executed with time-sharing. With LLPC, the execution time of the parallel applications increases from 15% to 25% while with time-sharing, the execution time increases by at least 46%. For `su2cor`, the run-time with time-sharing is more than twice its stand-alone run-time. This table also shows that the average number of threads used per parallel loop for each benchmark with LLPC has decreased from around four threads per loop in stand-alone execution to about 3.25 threads per loop. This change clearly illustrates that LLPC adjusted its thread usage to compensate for the varying sequential background load.

Table 3: The slowdown factors of the benchmarks when using time-sharing and LLPC to execute one parallel application and one sequential application.

Application	time-sharing	LLPC	threads/loop
swim	1.46	1.20	3.26
su2cor	2.39	1.15	3.27
hydro2d	1.97	1.17	3.25
mgrid	1.73	1.25	3.27

4.5 LLPC Compared to Static Partitioning

To evaluate the effectiveness of LLPC when there are multiple parallel applications sharing the system, we compare LLPC to the Solaris version of static partitioning called *processor set (*`psrset`*)*. The processor set feature allows the system administrator to partition the processors in a system into several subsets. These subsets then run only those applications that are specifically assigned to them, although multiple applications can share a subset of processors using time-sharing.

For the processor set measurements, we partitioned the system into two sets with two CPUs in each. One of the SPEC application programs then was executed in each set. For the LLPC measurements, the SPEC applications were compiled with the modified microtasking library. Pairs of applications were then executed simultaneously on all four processors while they adjusted their processor requirements using the LLPC algorithm described above. Because of the dynamic nature of LLPC, a slight change in the system load can affect the number of processors used to execute a parallel loop, which then produces slightly different total execution times. Therefore, we conducted the LLPC experiment five times for each set of benchmarks and present all five execution times for comparison. We did not compare the performance with time-sharing in these experiments since time-sharing proved to be significantly slower than both LLPC and processor set.

The slowdown factors shown in Figure 3 suggest that, in almost all cases, LLPC reduces the slowdown of the parallel applications compared to the static partitioning of the processor set approach. The reductions in the slowdown factor for LLPC range from a few percent to almost 20% when `su2cor` is run with `mgrid`. Only one benchmark, `swim`, troubles the LLPC approach. This benchmark, as defined by SPEC, consists of a single application that is run twice in succession with two different input data sets. As a result, the start-up costs for LLPC are encountered twice which contributes to the higher overhead of LLPC on this application, as shown in Table 2. Moreover, after the first run finishes and before the second run starts, the other program executing in the system is able to grab all of the processors, leaving the second run of the `swim` benchmark with only a single thread to use. This thereby puts the second run of `swim` at an initial disadvantage.

One of the main advantages of LLPC, however, is that it allows a much more flexible allocation of the processors. Instead of limiting the number of threads used by an application to 2 as in the statically partitioned case, LLPC adjusts the allocation based on the availability of the processors. For instance, when `su2cor` was running concurrently with `hydro2d`, Table 4 shows that `su2cor` used 2.30 threads per parallel loop while `hydro2d` used 2.97. This table shows that all of the applications were able to use an average of more than 2 processors to execute their parallel loops with LLPC. The static partitioning, however, sets a hard limit of at most two processors for the execution of each application's parallel loops. Thus, this flexibility in being able to use otherwise idle processor resources allows LLPC to outperform the statically partitioned processor set strategy.

Table 4: The average number of threads executed per parallel loop when using LLPC to execute two parallel applications simultaneously.

	swim	su2cor	hydro2d	mgrid
swim		2.91	2.47	2.20
su2cor	2.68		2.30	2.52
hydro2d	2.97	2.97		2.66
mgrid	2.86	3.28	2.81	

5 Conclusion

Fairly allocating the processors of a multiprogrammed shared-memory multiprocessor system is necessary to minimize the execution time of individual parallel applications while still maintaining high overall system utilization. Previous research has shown that, for loop-level parallelized applications, techniques that dynamically adjust the parallelism of the applications based on the system load can effectively achieve these contradictory goals [6, 11, 16]. This paper has demonstrated how to incorporate the LLPC dynamic processor allocation strategy [16] into the Solaris production operating system and related parallelizing compiler.

A unique feature of this implementation is the addition of a *sleeping* state for slave threads that allows parallel applications to dynamically adjust how much load they place on the system. Another unique feature is the masking of a sleeping thread's restart time with the execution of parallel loop iterations. Experiments with the SPEC95 benchmark suite show that LLPC allows simultaneously executing parallel applications to exploit more parallelism on average in each parallel loop than static partitioning or time-sharing. As a result, parallel applications executed with LLPC have shorter execution times on multiprogrammed systems that those executed using static partitioning or time-sharing.

6 Acknowledgements

We are indebted to Andrew Tucker for implementing the scheduling control feature that is currently being shipped with Solaris, including the processor set feature. We also would like to thank Michael Sebrée and Dan Stein for their help with various kernel and thread issues. Special thanks go to Vinod Grover for his assistance with the compiler. We also appreciate the help from Seth Strumph and Chris Quenelle from the TNF team.

Solaris is a trademark of Sun Microsystems, Inc. *IRIX* is a trademark of Silicon Graphics, Inc.

References

[1] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler activations: Effective kernel support for user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53 – 79, feb. 1992.

[2] M. J. Bach. *The Design of the Unix Operating System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[3] J. Barton and N. Bitar. A scalable multi-discipline, multiple-processor scheduling framework for IRIX. In *Workshop on Job Scheduling Strategies for Parallel Processing, International Parallel Processing Symposium*, pages 24 – 40, April 1995.

[4] V. Grover, X. Kong, M. Lai, and J.-Z. Wang. *SunSoft Parallelizing Compiler for Fortran and C*. Internal document, Sun Microsystems Inc., April 1996.

[5] A. Gupta, A. Tucker, and S. Urushibara. The impact of operating system scheduling polices and synchronization methods on the performance of parallel applications. In *ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, volume 19, pages 120–132, 1991.

[6] R. Konuru, J. Moreira, and V. Naik. Application-assisted dynamic scheduling on large-scale multicomputer systems. In *Second International Euro-Par Conference (Euro-Par'96)*, pages II:621–630, 1996.

[7] D. Lilja. Exploiting the parallelism available in loops. *Computer*, 27(2):13 – 26, February 1994.

[8] V. Naik, S. Setia, and M. Squillante. Performance analysis of job scheduling policies in parallel supercomputing environments. In *Supercomputing*, pages 824–833, 1993.

[9] J. Ousterhout. Scheduling techniques for concurrent systems. In *Distributed Computing Systems Conference*, pages 22–30, 1982.

[10] M. Powell, S. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. SunOS multi-thread architecture. In *USENIX Conference Proceedings*, pages 65 – 80, 1991.

[11] C. Severance, R. Enbody, S. Wallach, and B. Funkhouser. Automatic self-allocating threads on the Convex Exemplar. In *International Conference on Parallel Processing*, pages I:24 – 31, 1995.

[12] Sun Microsystems Inc. Ultra enterprise x000 server family: Architecture and implementation. Technical report, Sun Microsystems Inc., Mountain View, California, 1996. `http://www.sun.com/servers/ultra_enterprise/arch.html`.

[13] A. Tucker. *Efficient Scheduling on Multiprogrammed Shared-memory Multiprocessors*. PhD thesis, Department of Computer Science, Stanford University, 1993.

[14] A. Tucker. *Scheduler Activations in Solaris*. Internal document, Sun Microsystems Inc., April 1996.

[15] K. Yue and D. Lilja. Loop-level process control: an effective processor allocation policy for multiprogrammed multiprocessor systems. In D. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 182 – 199. Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949. Also available in *Workshop on Job Scheduling Strategies for Parallel Processing, International Parallel Processing Symposium*, pages 112–121, April 1995.

[16] K. Yue and D. Lilja. Efficient execution of parallel applications in multiprogrammed multiprocessor systems. In *10th International Parallel Processing Symposium*, pages 448–456, April 1996.

[17] K. Yue and D. Lilja. Performance analysis and prediction of processor scheduling strategies in multiprogrammed shared-memory multiprocessors. In *International Conference on Parallel Processing*, pages III 70–78, August 1996.

[18] J. Zahorjan, E. Lazowska, and D. Eager. Spinning versus blocking in parallel systems with uncertainty. In *International Seminar on Performance of Distributed and Parallel Systems*, pages 455–462, December 1988.

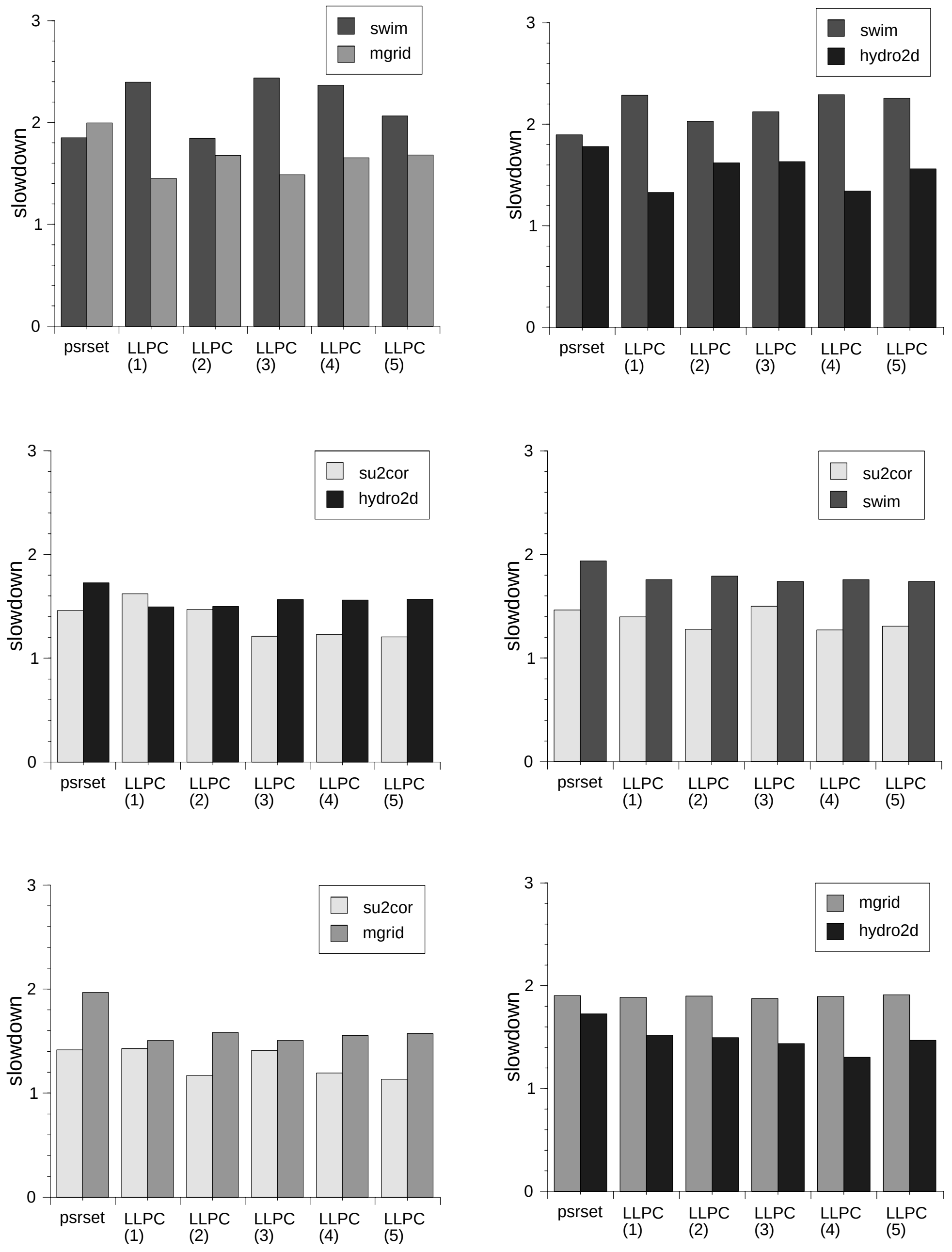


Figure 3: Comparing the slowdown factors of LLPC to the Solaris *processor set (psrset)* static partitioning mechanism when executing two parallel applications simultaneously. Five different runs of the LLPC-enabled applications are shown to demonstrate its potential variability.