Kernel-Level Scheduling for the Nano-Threads Programming Model

Polychronopoulos, E. D., Martorell, X., Nikolopoulos, D., Labarta, J., Papatheodorou, T. S., & Navarro, N. (1998). Kernel-Level Scheduling for the Nano-Threads Programming Model. In *Proceedings of the 12th ACM International Conference on Supercomputing (ICS)* (pp. 337-344). ACM. <https://doi.org/10.1145/277830.277911>

Published in:

Proceedings of the 12th ACM International Conference on Supercomputing (ICS)

Queen's University Belfast - Research Portal:

[Link to publication record in Queen's University Belfast Research Portal](#)

General rights

Copyright for the publications made accessible via the Queen's University Belfast Research Portal is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy

The Research Portal is Queen's institutional repository that provides access to Queen's research output. Every effort has been made to ensure that content in the Research Portal does not infringe any person's rights, or applicable UK laws. If you discover content in the Research Portal that you believe breaches copyright or violates any law, please contact openaccess@qub.ac.uk.

Kernel-Level Scheduling for the Nano-Threads Programming Model

Eleftherios D. Polychronopoulos[†], Xavier Martorell[‡], Dimitrios S. Nikolopoulos[†]
Jesus Labarta[‡], Theodore S. Papatheodorou[†], Nacho Navarro[‡]

[†]High Performance Computing Architectures Laboratory
Department of Computer Engineering And Informatics
University of Patras, Greece

[‡]European Center for Parallelism of Barcelona
Department d'Arquitectura de Computadors
Universitat Politècnica de Catalunya, Spain

e-mail: {edp,dsn,tsp}@hpcclab.ceid.upatras.gr, {xavim,jesus,nacho}@ac.upc.es

Abstract

Multiprocessor systems are increasingly becoming the systems of choice for low and high-end servers, running such diverse tasks as number crunching, large-scale simulations, data base engines and world wide web server applications. With such diverse workloads, system utilization and throughput, as well as execution time become important performance metrics. In this paper we present efficient kernel scheduling policies and propose a new kernel-user interface aiming at supporting efficient parallel execution in diverse workload environments. Our approach relies on support for user level threads which are used to exploit parallelism within applications, and a two-level scheduling policy which coordinates the number of resources allocated by the kernel with the number of threads generated by each application. We compare our scheduling policies with the native gang scheduling policy of the IRIX 6.4 operating system on a Silicon Graphics Origin2000. Our experimental results show substantial performance gains in terms of overall workload execution times, individual application execution times, and cache performance.

1 Introduction

Current shared-memory multiprocessor architectures, including small symmetric multiprocessors and scalable distributed shared memory systems, are widely adopted as a viable and powerful platform for high-performance computing. Among other attractive properties, these machines offer a standard, transparent multiuser/multiprogramming environment to support large numbers of simultaneously running parallel and sequential applications.

Parallel applications that execute on a multiprogramming system often suffer from serious performance degradation, due to the interferences between simultaneously executing programs which contend for system resources. Events like blocking system calls, paging, starting and termination of processes introduce a high variability in the system workload, and can have significant impact in execution time. A major drawback of the vast majority of parallel applications is that their implementation assumes that the programs are executed in a dedicated system, so they setup the desired number of proces-

sors to run on and execute as if the required processors were continuously available to them. This means that parallel applications usually do not know anything about the actual number of processors allocated to them by the operating system, and they are unable to adapt to changes in the system conditions during execution. At the same time, commercial multiprocessor operating systems continue to offer scheduling policies based on the standard UNIX time-sharing mechanism. Several research results have shown that this mechanism is inappropriate for several reasons [9, 18, 19]. Time-sharing of processes belonging to parallel applications can have undesirable effects such as inopportune preemptions, increased bus contention and high cache miss rates.

Both application adaptability and preempted work¹ recovery can be achieved by providing a lightweight communication path between applications and the operating system. In this paper, we present kernel-level mechanisms and scheduling policies which we integrated in a dynamic execution environment that allows parallel applications to execute in tight cooperation with the operating system and adapt to changes of the number of processors available to them at runtime. We are developing the fully integrated execution environment in the context of the NANOS ESPRIT project, following the ideas initially from the definition of the Nano-Threads Programming Model (NPM) [16, 17]. Our current prototype consists of an auto-scheduling compiler, a multithreading runtime library (NthLib [12, 13]), an extended kernel interface and a user-level CPU manager.

We present the definition and describe the functionality of the interface established between applications and the operating system through shared memory. This interface is used as the basic mechanism to implement dynamic application adaptability to the available system resources. As an initial approach, we have implemented the interface in the context of a user-level CPU manager. We also propose a kernel-level scheduling policy that takes into account the processor requirements of the executing applications and the overall system workload. An important aspect of our scheduling policy is that it is a two-level policy. The higher level is in charge of distributing processors among applications, while the lower level uses heuristics to minimize inter-application movement of processors and enhance affinity.

We evaluate our environment on a SGI Origin2000 and compare its performance to that of the native IRIX 6.4 operating system scheduler. The experimental results show that our mechanisms perform significantly better than the native system scheduler for applications which are written and par-

¹Throughout this paper the term *work* refers to a user-level thread executing on top of a kernel thread or process.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICS 98 Melbourne Australia

Copyright ACM 1998 0-89791-998-x/98/ 7...\$5.00

allelized within the NPM.

The rest of the paper is organized as follows: Section 2 overviews the Nano-Threads Programming Model. In Section 3, we describe the functionality of the kernel interface and the basic primitives that compose the interface. Section 4 presents a dynamic two-level scheduling policy, devised for the Nano-Threads execution environment. In Section 5 we compare our approach with other proposals in the literature. Section 6 describes our evaluation framework and Section 7 presents detailed experimental results. We summarize the paper in Section 8.

2 Overview of the NPM

Figure 1: The Nano-Threads Programming Model

Figure 1 outlines the Nano-Threads Programming Model. The integrated compilation and execution environment consists of three main components [17]:

- An *auto-scheduling parallelizing compiler*, which automatically detects and extracts multiple levels of parallelism from application codes, and schedules parallel tasks based on program information.
- A *user-level execution environment* which offers light-weight threads (nano-threads) and a user-level scheduler as the basic execution framework for parallel tasks generated by the compiler. These functionalities are provided by the means of a runtime library.
- An *operating system*, which controls hardware resources and implements a kernel-level scheduling policy that space and time shares the available processors across the active programs, based on the total machine workload and the feedback provided by the programs through a well-established kernel interface.

The *auto-scheduling compiler* (based on Parafrase2 [15] and extended to support Open-MP like directives [3]) performs fine-grained multilevel parallelization and exploits both functional and loop parallelism from standard applications written in FORTRAN. Applications are decomposed in multiple levels of parallel tasks. The compiler uses the Hierarchical Task Graph (HTG) [8] structure to represent the application internally. Management of multiple levels of parallel sections and loops allows the extraction of all the useful parallelism contained in the application.

During the execution of an application, nano-threads are the entities offered by the user-level execution environment to instantiate the nodes of the HTG, with each nano-thread corresponding to a node in the HTG. The *user-level execution environment* controls the creation of nano-threads at run-time, ensuring that the generated parallelism matches the number of processors allocated to the application by the operating system.

The operating system distributes physical processors among running applications. The resulting environment is multi-user and multi-programmed, allowing each user to run parallel and sequential applications. The kernel-level scheduling policy is concerned with the allocation of physical processors to the applications currently running in the system.

The main scheduling objective of NPM is that both application scheduling (nano-threads to virtual processors mapping) and virtual processor scheduling (virtual to physical processors mapping) must be tightly coordinated in order to achieve high performance. In addition, the kernel-level scheduler must avoid any interference with the user-level scheduler, in order to keep runtime scheduling overhead low.

3 Kernel Interface Definition & Functionality

In this section, we describe the kernel interface established between parallel applications and the operating system inside NPM.

3.1 Overview and Design

The interface described in this section is used by applications to cooperate with the operating system in order to avoid certain scheduling decisions that may hurt their performance. The behaviour of such *cooperative applications* is the following:

1. Each application dynamically informs the operating system about its requirements (number of processors) reflecting the actual degree of parallelism that the application wants to exploit at user-level.
2. The operating system distributes processors at a fixed time slice, taking this information into account.
3. The application is informed about the operating system allocation decisions and tries to match the parallelism that it generates to the assigned number of processors.

In more detail, when the operating system decides to reallocate processors, it first decides which applications shall loose processors in the next time slice. Then, it asks these applications for processors to be freed, giving them a certain amount of time (the "grace time") to answer to the request by releasing the processors. If an application does not answer to the request in time, the operating system will forcefully claim back processors through preemption, and inform the application. When some work has been preempted, the application always readapts at user-level when a running virtual processor reaches a "safe" point, by yielding the associated physical processor to a preempted process. A safe point is a user-level dispatching point, where the virtual processor knows that the application and runtime synchronization constraints are satisfied. Ensuring that all preempted virtual processors are stopped at safe points is very important in order to avoid preemption inside critical sections. On the other hand, when the application decides on its own, also at a safe point, to release a physical processor, the associated virtual processor is terminated.

Applications not explicitly coded to cooperate with the kernel (*uncooperative applications*) are also considered by the operating system when deciding the allocation of processors. Any useful information available, such as the number of ready threads limited by the number of physical processors in the system, can be used by the operating system to estimate the requirements of these applications. Therefore, these applications do not escape from kernel control. Space-sharing is also applied to them in a manner which ensures that their performance will be very close to the one achieved using the traditional time-sharing scheduling policies.

We use an upcall mechanism and shared memory to inform applications about kernel-level scheduling decisions. Processor allocation and blocking on I/O are communicated to user-level through upcalls because there is already an available processor to do this work and the overhead is small [2]. The request for processors, processor preemption and unblocking are communicated through shared memory. In this case, the application polls at shared memory at every user-level scheduling point to see if there is any of such events. Fine-grain parallelization helps in that such polling is made often enough to detect most of the operating system requests.

3.2 Kernel Interface

In this subsection we describe the set of services offered by the kernel interface to implement the previously described functionality. It shows the proposed kernel interface to support the execution of the *cooperative applications*.

At any time, applications use the `cpus_request` (`cpus`) primitive to set the number of processors that they want to run on. The operating system will try to assign as many processors as possible without exceeding `ncpus`. The primitive can be used dynamically at runtime to request more or less processors than those currently assigned. The operating system supplies physical processors with the creation of virtual processors. Creation of virtual processors is based on the basic primitives provided by operating systems. Examples of these primitives are the `spoc` call in IRIX and the `thread_create` call in Mach.

The application can release a specific physical processor, as an answer to an operating system request using the `cpus_release_self()` primitive. It is intended to be used when the system requests the application to return some processors.

To allow an application to recover the execution of pre-empted work, the kernel offers the `cpus_processor_handoff(work)` primitive. The virtual processor identified by `work` receives the current physical processor.

The `cpus_requested()`, `cpus_current()` and `cpus_askingfor()` primitives help the application to know at any time which is the status of the processor allocation.

Preempted work recovery is supported through the following primitives: `cpus_preempted()` returns the number of processors which are preempted from the application by the operating system. And `cpus_get_preempted_work()` returns a reference to a preempted context that can be resumed at user-level.

4 Kernel-Level Scheduling Policy

We introduce a two-level scheduling policy aiming at increasing system utilization and efficiency, by effectively space-sharing the machine among the running applications. This is done while the kernel attempts to satisfy the processor requirements of parallel applications which may be executing

simultaneously (subject to the aggregate resource requests of the workload). In this section we present the scheduling algorithm and describe its properties.

At the higher level, *Dynamic Space-Sharing Scheduling* (DSS) partitions the available processors among the executing applications taking into consideration the processor requirements of each application, as well as the overall system workload. The number of processors assigned to each application is, in general, a fraction of the total number of processors requested by that application, and it is a function of the total resource requirement of the entire workload and the number of physical processors. DSS also guarantees that there are no applications waiting to starvation.

Processors assigned to an application remain in the corresponding address space until their time quantum expires. We assume that all processor allocations are done for a base time quantum of 100ms. In [5] we present extensions to DSS which are based on variable number of processors and variable size of time quanta handling also process priorities. Even though processors are allocated to each application as a group of x_i processors (as discussed later in this section), our policy does not require gang scheduling which may involve unnecessary overhead associated with idling in order to accumulate the required number of processors. Instead, DSS is a dynamic policy where the unit of allocation is an idle physical processor. At the upper level, the algorithm simply computes the number (x_i) of processors to be allocated to the next application at the head of the ready queue.

At the lower level, the proposed scheduling policy applies a heuristic called *Selective Scheduling*, which attempts to facilitate processor locality, thus promoting memory locality and cache affinity. We return to the description of Selective Scheduling later in this section.

4.1 The DSS Algorithm

Our approach to scheduling on scalable NUMA multiprocessors aims at providing fair scheduling and thus equitable distribution of the physical resources (processors and memory) to competing parallel applications. Processes that are I/O bound, batch and real-time processes may be excluded and treated at a different level of priority. In a parallel execution environment, this requires accounting (i.e., monitoring of resource utilization) that takes into consideration processor allocation in terms of space (number of processors) and time (length of assignment for each processor).

In our environment, all nano-threaded applications communicate requests to the operating system using the interface described in Section 3. This continuous "updating" helps the kernel to redirect resources to where they are mostly needed. Therefore, it results in improved global system utilization. On the other hand, and in order to avoid starvation of other applications, each processor (or group of processors) assigned to an application is assigned for a pre-determined or computed period of time (time quantum).

Originally, DSS is invoked whenever a new application arrives for execution, when an application changes its request during execution, either by requesting or by releasing one or more processors and whenever an application terminates execution. In the current implementation, DSS is invoked only after the expiration of a time quantum. All ready applications are queued in a logical queue and idle processors are first assigned to the application at the head of the queue. DSS is invoked in two stages: In the first stage, DSS computes the number x_i of processors that are to be assigned to the newly arrived process, and updates some global descriptor variables. In the second stage, DSS applies Selective Scheduling which

is described later in this Section. As mentioned earlier, the number of processors assigned to each application is analogous to the number of processors requested by that application. DSS attempts to be fair to all applications running on the system. However, it favours parallel applications and not necessarily sequential ones with long execution time. This is a result of the fixed time quantum.

More specifically, each new application that enters the system requests from the kernel a number of processors ρ_i , $i = 1, 2, \dots, n$, using the `cpus_request` call of the kernel interface. DSS maintains a workload size index W which is defined by:

$$W = \sum_{i=1}^n \rho_i \quad (1)$$

Although W may not be an ideal metric for workload size, it is an accurate index of the total workload requirement in terms of processor resources, which is needed by our policy. Whenever a new application enters the system, it updates W by adding its resource requirements: $W = W + \rho_i$. Similarly, every time an application completes execution and exits, it updates W by $W = W - \rho_i$. DSS partitions the processors available in the system among the applications according to the number of processors requested by each application (ρ_i) reflected as a percentage of the total number of processors requested by the workload W . If $W \leq P$, where P is the total number of physical processors in the system, then $x_i = \rho_i$, thus assigning to each application the exact number of processors requested. In any other case DSS calculates x_i as follows.

Each application gets $\min(x_i, \rho_i)$ processors. Let,

$$R_i = \left\lfloor \frac{\rho_i \times P}{W} \right\rfloor, \quad (2)$$

we define

$$\delta_i = \begin{cases} 1 & R_i < 1 \\ 0 & R_i \geq 1 \end{cases}$$

Then x_i is given by:

$$x_i = R_i + \delta_i \quad (3)$$

In the current implementation, in order to alleviate problems introduced by roundoffs, we introduce a variable f in which we sum the remainders of the roundoffs, assigning one more processor to the application being serviced if $f \geq 1$. In this case one can trivially show that:

$$\sum_{i=1}^n x_i = P, \text{ for } n \leq P. \quad (4)$$

If the number of applications in the workload is $n > P$, the DSS algorithm is applied in stages, with each stage involving a group of P applications. The key property of DSS is that for a given application j requesting ρ_j processors, the number of processors assigned to it is proportional to ρ_j/W . This ratio characterizes the percentage of the total workload that application j represents. It is clear that in all cases $1 \leq x_j \leq \rho_j$. Under light workload conditions $x_j = \rho_j$, while under heavy workloads the same application will receive $x_j = 1$ processors. Although the two previous scenarios are the extreme cases, any value of x_j between 1 and ρ_j is possible depending on the relative size of ρ_j compared to W (i.e., the relative position of the specific application with respect to the entire workload). The above is a very desirable property for a dynamic kernel scheduler, since any other alternative would result in fragmentation of processor groups, underutilization of

the system, and possible long execution times for some applications.

4.2 The Selective Scheduling Heuristic

During execution, a process establishes affinity in the cache (cache footprint) and local memory, which increases with the length of execution. It is very desirable to be able to assign the same set of physical processors to the same process across different DSS allocation phases.

In this section we describe an enhancement to DSS called Selective Scheduling, which attempts to achieve exactly the above goal. This can be done by “remembering” the processors on which a given application executed in the past, and by trying to reschedule the processes of this application onto the same processors. However, this involves potential trade-offs and potential overhead. For example, we need to consider the case where the processors that executed a given application are all busy executing a different application: do we wait until they become idle, or do we assign the first processors that become idle regardless of execution history?

In this work we present a simple extension with low overhead, which, based on our experimental results, is quite effective. This policy uses execution history information and attempts to first schedule a process on the same processors but “gives up” if these processors are not available. We first define the additional data structures and the type of history information that needs to be recorded and tracked.

We define a global (maintained by the OS) array $\mathcal{A}[1 : P]$ where its k -th entry corresponds to processor k . The use of this array is to store the execution status of the physical processors in the system. If $\mathcal{A}[k] = 0$ then the k -th processor is idle (and thus available), otherwise the k -th processor is assigned to an application. At boot time the entries of array $\mathcal{A}$ are initialized to zero.

In addition to the above structure, we define a bit vector private to each application, which is modifiable by the CPU manager: $V_i[1 : P]$ is the bit vector maintained for application i . Initially, all bits of V are zero. If a processor k is assigned to an application i , then the CPU manager sets $V_i[k] = 1$.

Each time DSS services applications from the CPU manager queue, it first assigns to the application all processors still executing on the same application and then inspects its V_k vector and assigns processor(s) m for which: (1) $V_k[m] = 1$ (i.e. processors previously assigned to the application and are either released by the application or stolen by the kernel), or (2) $\mathcal{A}[m] = 0$ (i.e. processors that are idle).

Several variations of this heuristic are possible and depend on the level of history information that is tracked as well as on the importance the scheduler places on memory locality versus overhead. We first consider the level of execution history that can be recorded in the V vectors.

The implementation of this heuristic works by having the scheduler scan the bit vector V_k of an application k attempting to allocate as many of the x_k processors from the same ones that executed the application in the past, as possible. However, if not all x_k are idle/available, the scheduler proceeds by assigning the remaining from the next immediately available processors (updating the corresponding V_k entries accordingly). A more thorough investigation of the above heuristics as well as experimental evidence of their relative performance appears in [5].

Selective scheduling tries to ensure data locality, if this is attainable, in every scheduling phase of each executing application. Achieving high hit cache rates in scalable-shared memory multiprocessors is one of the most challenging problems of high performance computing.

5 Comparison With Other Scheduling Techniques

In this section, we consider the differences among our approach for kernel scheduling, the widely used gang scheduling [14] and other proposed policies, notably Process Control [18, 19], First-Class Threads [11] and Scheduler Activations [2].

The concept of using shared memory to communicate between the kernel and user processes as well as the notion of scheduling based on proportional processor allocations was first proposed in [17]. Some of our earlier research on refining and evaluating the DSS scheduler and the design and implementation of the specific kernel/user shared memory interface influenced aspects of the scheduler in SGI's IRIX 6.5 [4]. Our collaborative work with the OS group at SGI and research groups at the University of Illinois at Urbana-Champaign continues with new emphasis on enhancing the sensitivity of our kernel scheduler to memory affinity.

We summarize the differences with gang scheduling in four relevant aspects: the first concerns the question about who is in charge of setting the degree of parallelism of an application. It is set by the user in gang scheduling, where the application simply creates a number of processes and the operating system must schedule all of them together to allow the application to make progress. Our approach allows the application to set the maximum number of processors it wants to run on and the operating system can (and will) allocate the most suitable number of processors taking into account the overall system load. This feature is already included in Process Control [19]. Similarly to this approach, we also allow dynamic requests during the lifetime of an application.

The second important difference between our approach and gang scheduling is how blocking interferes with kernel scheduling decisions. When a process of a gang-scheduled application blocks in the kernel, the processor immediately tries to execute some other process. This process may belong to another gang-scheduled application, reclaiming other processors to run in its turn, and motivating more context switches. Although gang scheduling ensures that all processes of a parallel application start executing together at the beginning of a time quantum, it does not ensure that all the required processors will be available to the application till the end of the time quantum. At this point the application performance is degraded because of synchronization issues. Our approach maintains the number and mapping of processors allocated to the application, similarly to Scheduler Activations [2]. Further study of kernel blocking is necessary to know which blocking events may be communicated to the application and which ones should be managed transparently by the operating system.

With respect to the synchronization mechanisms used, gang scheduling applies very often two-level synchronization methods [19, 7]. When a spinning process spends too much time to acquire a spin lock, the run-time package decides to yield the processor to allow the progress of other processes in the system. While this is a good solution, it is not general enough to perform well in all situations. As the operating system knows nothing about the execution at user level, an application may loose a processor which, in the worst case, may be given to another spinning process. This is the third difference with our approach. By now, we are not using two-level synchronization: an application detects, at safe points, when one of its virtual processors has been preempted and then it yields one of its processors to the preempted one.

The last significant difference is the amount and quality of the information shared between an application and the operating system. Process Control maintains a counter of processors

allocated to the application and informs the application of kernel level events through UNIX signals. Scheduler Activations and First-Class Threads inform using upcalls. Our proposal is to use the most efficient mechanism to inform applications of such events. Processor allocation and blocking are easily communicated through upcalls, offering the new processor to the application. Processor preemption and unblocking are most easily informed through shared memory. For this reason, we maintain a shared memory area between the applications and the operating system containing the information explained in subsection 3.2.3.

Feitelson [7] discusses a gang scheduling technique called gang scheduling with malleable preemption. It allows applications to shrink/expand their execution to run on less/more processors. Our approach will allow us to test execution under this scheme. Further analysis on this topic is out of the scope of this paper.

6 Evaluation Framework

In this section we describe the experimentation environment used to evaluate our approach. We outline the platform architecture, the implementation of the kernel interface and scheduling policies, and the workloads used for performance evaluation.

6.1 Experimentation Environment

We experimented on a 32-processor SGI Origin 2000 [10]. Origin2000 is a cache-coherent NUMA multiprocessor. The system uses the MIPS R10000 4-way superscalar microprocessor and it is controlled by the IRIX 6.4 operating system. The basic mechanisms employed by IRIX 6.4 to enhance parallel processing are gang scheduling, exploitation of process affinity for processors and dynamic page migration and replication to maintain memory locality [20].

The kernel interface and scheduling policies were implemented in a user-level CPU manager running on top of IRIX 6.4. Although a user-level implementation lacks flexibility and efficiency compared to an in-kernel implementation, it is sufficient to denote the relative merits of our approaches. The CPU manager implements the kernel part of the interface described in Section 3, with the exception of the two-minute warning mechanism which was still under testing at the time we performed the experiments.

Parallel applications contact the CPU manager, by requesting a number of processors on which they wish to run. The CPU manager is awakened at regular time intervals. In each invocation, the CPU manager applies the processor allocation policy, computes the new allocation of processors and performs only the necessary processor movements among applications. It also informs the applications of the new system conditions, in order to let them dynamically readapt by performing handoffs and suspending kernel threads. The time quantum between subsequent invocations of the CPU manager is set to be equal to the time quantum used by the IRIX gang scheduling policy (100 milliseconds), in order to compare the two approaches under the same conditions.

6.2 Workload

We performed experiments with numerous workloads consisting of several parallel programs with different characteristics and computational requirements. We experimented with workloads that modeled light loads, moderate loads and heavy loads. Due to space limitations, we present results for a workload that models a heavy load. We believe that this workload

Table 1: Experimental workload

Benchmark	Inst.	Request	T_{seq}	$T_{request}$	S
hydro2d	4	8	26.09	10.62	2.46
lu	4	8	29.49	6.14	4.80
jacobi	4	8	39.04	7.96	4.91
diamond	4	8	72.76	37.09	1.96

is representative of real situations in multiprogrammed parallel systems and it is appropriate to make quantitative comparisons. The relative performance of our mechanisms compared to the native IRIX scheduler remained unaltered for all the workloads that we tested.

Table 1 summarizes the workload. The first three columns show the benchmarks, the number of instances of each benchmark in the workload, and the number of processors that each instance requests initially from the operating system. The latter three columns show the sequential execution time, the parallel execution time when the benchmark is executed with all the requested processors in a dedicated system, and the speedup attained in this case.

The first application used in the workload is a parallelized version of the hydro2d benchmark included of the SPECfp95 suite. This benchmark includes a large computation kernel with inherent multilevel and unstructured parallelism. The lu and jacobi benchmarks are application kernels with inherent loop parallelism. Diamond is a synthetic benchmark with a diamond-like task graph. The benchmark goes through a sequence of parallel computation phases. In each phase the program requests a number of processors and executes a constant amount of work on the requested processors. The processor requests start from one, increase up to a maximum number of processors and then decrease back to one at the end of the computation, hence the name of the benchmark. This benchmark is used to model a continuous dynamic variation of the workload. The amount of work in each parallel phase is adjusted so that the benchmark changes its processor request at least once and at most twice in each scheduler time quantum.

The hydro2d benchmark is written in FORTRAN while the other three benchmarks are written in C. All benchmarks are parallelized in the nano-threads environment and linked to NthLib. Parallelization with the nano-threads library favours our mechanisms since the application part of the kernel interface is integrated in NthLib. On the other hand, using the nano-threads library has a negative impact on the native IRIX scheduler with respect to synchronization. The native IRIX synchronization primitives implement a two-phase synchronization algorithm, while the nano-threads library uses pure spinning synchronization instead. Applications parallelized with the SGI MP library use two-level synchronization and they are expected to exhibit better behaviour under gang scheduling in IRIX. In order to make a more fair comparison, we parallelized the hydro2d benchmark with the MIPS FORTRAN compiler and the MP library and executed 8 instances of the benchmark under IRIX gang scheduling. We use this workload to compare our mechanisms with the complete IRIX parallelization environment.

7 Experimental Results

Our analysis is based on the comparison between the native IRIX 6.4 gang scheduling policy and our kernel interface and processor allocation policies. Initially, we use the workload presented in Table 1, to compare the two approaches when all applications are parallelized within the NPM. We then use the workload with 8 instances of the hydro2d benchmark to com-

Figure 2: Workload execution time

pare our environment with the complete SGI parallelization environment. The figures presented are averages of several runs of the workloads in a dedicated system. The variations in execution times between subsequent runs of each workload were statistically insignificant.

We implemented the following set of processor allocation policies in the user-level CPU manager:

- *Batch*: This policy serves applications on a FCFS basis and allocates to each application exactly as many processors as the application has requested. Processors remain pinned to the application until the application terminates. Batch represents an optimal case for processor allocation. Applications exploit the maximum degree of parallelism, processor movements are minimal and there is no cache pollution due to multiprogramming.
- *Round Robin*: This policy is similar to batch with the exception that applications are served in a round-robin manner. Round Robin is expected to perform close to IRIX gang scheduling.
- *Random*: This is also similar to batch, except that at each scheduling interval the policy allocates to each application a random number of processors between zero and the maximum number of processors requested.
- *DSS*: This is the policy described in Section 4.

All policies employ an affinity mechanism. When a processor is allocated to an application, it remains in the same application space, until either the application terminates or the CPU manager decides to move the processor to another application. Selective scheduling is also enabled in all cases to keep track of short-term processor history and further exploit affinity.

Figure 2 plots the normalized execution time of the complete workload of Table 1 under IRIX gang scheduling and our scheduling policies. Normalization is performed with respect to the execution time under IRIX gang scheduling. Several conclusions can be drawn from Figure 2. All scheduling policies that use our kernel interface outperform IRIX gang scheduling. This result indicates that using kernel-application communication to implement dynamic program adaptation is highly beneficial. The batch policy reduces workload execution time by as much as 64% compared to IRIX gang scheduling. DSS performs equally well to batch, a surprisingly good result given that DSS performs much more frequent movements of processors. DSS seems to be an interesting alternative for a real multiprogrammed parallel system. Round robin which has similarities with gang scheduling performs

Figure 3: Applications execution times

also noticeably better than IRIX. Intuitively, this result shows that introducing our kernel interface in the IRIX scheduler should provide noticeable gains in performance. Another interesting result is that random performs close to batch. This proves the good functionality of our kernel interface. Applications are able to adapt even if they are executed under totally unpredictable system conditions and processor allocation is changed radically during their execution.

The performance of IRIX gang scheduling can be justified with the following arguments: IRIX schedules gangs of fixed size, always equal to the maximum number of processors that the application requests. A gang is not accommodated by the scheduler unless the kernel finds enough processors to execute all members of the gang. Therefore, gangs often stall unnecessarily waiting for processors to become free. In our experiments we detected that IRIX had poor behaviour in the presence of blocking. When a process belonging to a gang blocks, the processor is switched to another process, possibly from a different gang. The processor spends some time before returning back to the application and this time is usually long enough to hurt application performance. Moreover, applications are not informed of the actual physical processors on which they are running, neither of processor preemptions in order to be able to readapt.

We examined the performance of individual applications by measuring their average execution time when executed in the workload under multiprogramming conditions. Figure 3 plots the results, normalized to the average execution times obtained with IRIX gang scheduling. The results agree with those in Figure 2. Our mechanisms and scheduling policies give significantly reduced execution times for all applications compared to IRIX. Note that under DSS, execution times are marginally better than under batch scheduling.

Figure 4 illustrates the execution times of the applications, normalized to their pure sequential execution time. With DSS and batch, three out of four applications experience satisfactory speedups under multiprogramming. Lu and jacobi obtain more than a two-fold speedup, while diamond gets almost as much of a speedup as if it were executed in a dedicated machine. IRIX fails to give any speedups and three out of four benchmarks exhibit significant slowdowns. Random and round robin give moderate results, with the exception of hydro2d where surprisingly, random is the only policy that achieves any speedup.

We used the MIPS R10000 hardware counters and the IRIX `perfex` utility to obtain statistics about the memory performance of the different scheduling approaches. Figure 5 shows the accumulated number of cache misses occurred during the execution of the complete workload for the various

Figure 4: Applications speedups

Figure 5: Workload cache misses

scheduling policies. Although IRIX uses advanced techniques to enhance memory locality, our affinity heuristics achieve better results. With DSS and batch, cache misses are reduced by 20% compared to IRIX gang scheduling. DSS causes a negligible 3% increase of cache misses compared to batch which has optimal cache performance.

Figure 6, shows the results from the execution of the hydro2d workload parallelized with the SGI MP library and NPM and executed under IRIX gang scheduling and our CPU manager respectively. The results verify that parallelization with the MP library alleviates several performance bottlenecks of IRIX gang scheduling. The primary reason is that programs parallelized with the MP library exploit the two-level synchronization primitive of IRIX, thus utilizing processors better and reducing waiting times. However, our mechanisms still perform equally or marginally better than IRIX. Although we use pure spining synchronization, our handoff technique improves the synchronization behaviour of nano-threaded applications. We believe that integrating two-level synchronization and the two-minute warning mechanism in our implementation will further favour our approach for kernel-level scheduling.

8 Conclusions and Future Work

We have presented the kernel interface and scheduling policies designed to support the Nano-Threads Programming Model and proposed a two level scheduling policy to distribute processors among applications. Our performance results demonstrated that establishing a lightweight communication path between the kernel and parallel applications improves application behaviour radically under multiprogramming. The Nano-Threads parallelization environment exhibited highly competitive performance compared to the native IRIX envi-

Figure 6: Hydro2d workload execution time

ronment, even though our implementation is purely user-level and several optimizations were not already integrated.

Our current work is towards completing the implementation of the NPM kernel interface and studying several alternatives for kernel-level scheduling. We investigate the possibility of combining dynamic space-sharing with time-sharing and management of priorities, as well as different policies for two-level synchronization and scheduling of processors which are idling at the application level.

Acknowledgements

We are grateful to Constantine Polychronopoulos for the initial discussions about the NPM and his valuable comments on this work. We wish to thank all our colleagues within the NANOS ESPRIT Project too. Special thanks to Toni Cortés, Julita Corbalán (kernel interface), Eduard Ayguadé and Marc González (nano-threads compiler).

This work has been supported by the ESPRIT project NANOS (E21907), the Ministry of Education of Spain under contracts TIC-429/95 and TIC-97-1445-CE and the CEPBA.

References

- [1] M. Acceta, R. Baron, D. Golub, R. Rashid, A. Tevanian and M. Young, *Mach: A New Kernel Foundation for UNIX Development*, Proc. USENIX Conference, July 1986.
- [2] T. Anderson, B. Bershad, E. Lazowska and H. Levy, *Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism*, Proc. 13th ACM SOSP, October 1991.
- [3] E. Ayguadé, X. Martorell, J. Labarta, M. González and N. Navarro, *Exploiting Parallelism Through Directives on the Nano-Threads Programming Model*, Proc. 10th LCPC, Minneapolis, USA, August 1997.
- [4] N. Bitar and C. Polychronopoulos, Personal Communication on OS Scheduling and User Support Issues, OS Group, Silicon Graphics Inc., 1994–1997.
- [5] E. Polychronopoulos, *Process Scheduling for Multiprogramming Operating Systems on Shared Memory Multiprocessors*, Ph.D.Th. in progress, Univ. of Patras, 1997.
- [6] E. Polychronopoulos and T. Papatheodorou, *Dynamic Space-Sharing Scheduling at Kernel Level*, LHPCA Tech. Report TR-010797, Univ. of Patras, July 1997.
- [7] D. Feitelson, *Job Scheduling in Multiprogrammed Parallel Systems*, IBM Research Report 19790, Aug. 1997.
- [8] M. Girkar and C. Polychronopoulos, *Automatic Extraction of Functional Parallelism from Ordinary Programs*, IEEE Transactions on Parallel and Distributed Systems, vol. 3(2), pp. 166–178, March 1992.
- [9] A. Gupta, A. Tucker and S. Urushibara, *The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications*, ACM SIGMETRICS Performance Evaluation Review, Vol. 19(1), May 1991.
- [10] J. Laudon and D. Lenoski, *The SGI Origin: A ccNUMA Highly Scalable Server*, Proceedings of the 24th Annual International Symposium on Computer Architecture, pp. 241–251, Denver, Colorado, June 1997.
- [11] B. Marsh, M. Scott, T. LeBlanc and E. Markatos, *First-Class User-Level Threads*, Proc. 13th ACM SOSP, October 1991.
- [12] X. Martorell, J. Labarta, N. Navarro and E. Ayguadé, *Nano-Threads Library Design, Implementation and Evaluation*, Tech. Report UPC-DAC-1995-33, Sep. 1995.
- [13] X. Martorell, J. Labarta, N. Navarro and E. Ayguadé, *A Library Implementation of the Nano-Threads Programming Model*, Proc. 2nd Euro-Par Conf., Lyon, France, Aug. 1996.
- [14] J. Ousterhout, *Scheduling Techniques for Concurrent Systems*, In 3rd Int. Conf. on Dist. Comp. Systems, 1982.
- [15] C. Polychronopoulos, M. Girkar, M. Haghighat, C. Lee, B. Leung and D. Schouten, *Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing and Scheduling Programs on Multiprocessors*, Proc. 1989 ICPP St. Charles, Illinois, 1989.
- [16] C. Polychronopoulos, *Nano-threads: Compiler driven multithreading*, In 4th International Workshop on Compilers for Parallel Computing, 1993.
- [17] C. Polychronopoulos, N. Bitar and S. Kleiman, *nano-Threads: A User-Level Threads Architecture*, CSRD TR 1297, 1993.
- [18] A. Tucker and A. Gupta, *Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors*, Proc. 12th ACM SOSP, December 1989.
- [19] A. Tucker, *Efficient Scheduling on Multiprogrammed Shared-Memory Multiprocessors*, PhD Thesis, Stanford University, December 1993.
- [20] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum, *Operating System Support for Improving Data Locality on CC-NUMA Compute Servers*, Proc. 7th ASPLOS, October 1996.