An Efficient Kernel-level Scheduling Methodology for Multiprogrammed Shared Memory Multiprocessors

Eleftherios D. Polychronopoulos[1] Dimitrios S. Nikolopoulos[1] Theodore S. Papatheodorou[1]
Xavier Martorell[2] Jesus Labarta[2] Nacho Navarro[2]

[1] High Performance Information Systems Laboratory
Department of Computer Engineering and Informatics
University of Patras
Rion, 26 500, Patras, Greece

[2] Department d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Jordi Girona 1–3, Campus Nord, Modul C6, 08034,
Barcelona, Spain

Abstract

In this work we present an innovative kernel-level scheduling methodology designed for multiprogrammed shared-memory multiprocessors. We propose three scheduling policies equipped with both dynamic space sharing and time sharing, to ensure the scalability of parallel programs under multiprogramming while increasing processor utilization and overall system performance. Our scheduling methodology is designed for multidisciplinary multiprocessor schedulers that need to handle applications with widely different resource requirements and execution characteristics. We implemented the policies on a 64-processor SGI Origin2000 running Cellular IRIX and evaluated them in direct comparison with the native kernel scheduler. Our results demonstrate solid performance improvements over the vendor execution environment.

1 Introduction

Modern shared-memory multiprocessors, including small-scale SMPs and scalable cache-coherent NUMA (cc-NUMA) systems offer general-purpose multiprocessing and multiprogramming environments, that provide a single system view to the users and are able to execute simultaneously parallel jobs with mainstream and desktop applications. The need for providing multidisciplinary scheduling support on these systems motivated the use of standard UNIX schedulers, based on time-sharing and priority decaying to support jobs with diverge characteristics and resource requirements. The typical approach in contemporary operating systems that support symmetric multiprocessing, is to run a standard time-sharing scheduler on each processor possibly enhanced with affinity masks, to exploit the cache footprints of processes [5].

Although time sharing remains the policy of choice for multiprogrammed shared memory multiprocessors, numerous research works in the past have demonstrated that time sharing interacts poorly with parallel programs, primarily because standard time-sharing schedulers preempt threads of parallel programs obliviously, without being aware of synchronization or communication [5]. This research has also demonstrated that dynamic space sharing is a preferable option for the efficient execution of multiple parallel programs on shared memory multiprocessors. Although dynamic space sharing has exemplified its benefits for parallel programs [6], previous work has not addressed adequately the integration of dynamic space sharing in time-sharing schedulers. As a consequence, most modern operating systems either do not support space sharing, or offer space sharing as an option on a per-program basis, rather than as a transparent job scheduling strategy.

This paper presents three dynamic time and space sharing scheduling policies, that improve the performance of parallel programs in multiprogrammed environments and simultaneously increase system throughput and utilization, while having low complexity and runtime overhead.

The rest of this paper is organized as follows: Section 2 discusses related work. Section 3 presents three new time and space sharing policies. Section 4 presents our evaluation framework and experimental results. Section 5 concludes the paper.

2 Related Work

Several research works in multiprocessor scheduling have proposed gang scheduling (or coscheduling) [7] as the policy of choice for parallel programs on multiprogrammed shared memory multiprocessors. Although gang scheduling has proven to be quite effective on distributed memory multicomputers, it has not enjoyed wide acceptance in shared memory multiprocessors due to its implementation complexity and its inability to handle processor fragmenta-

tion and adapt easily to dynamically varying workloads.

Dynamic space sharing partitions the system processors among parallel applications according to some predefined policy and subsequently lets processors migrate from one application to the other, upon changes of the system load. Among various proposals, dynamic equipartition [6] appears to be the most popular dynamic space sharing scheme. Dynamic equipartition can be combined with affinity scheduling [5] to improve memory performance.

A feature shared among all previously proposed dynamic space sharing algorithms is the use of a communication path between parallel programs and the operating system scheduler, to exchange critical scheduling information e.g. the number of processors allocated to each parallel program from the operating system. Earlier proposals relied on heavyweight communication mechanisms, such as signals and upcalls to implement the kernel-user communication interface. More recently, research works with the Solaris and IRIX operating systems [3, 11] proposed the use of a *shared arena* as the most efficient communication medium. A shared arena is a memory region pinned to physical memory and accessible by both the programs and the kernel. Using a shared arena, the kernel and user programs are able to exchange scheduling information simply with loads and stores in shared memory.

One of the drawbacks of previous works on dynamic space sharing is that they lacked integration with general-purpose time-sharing schedulers. Although integrating time sharing and space sharing in a unified multidisciplinary scheduler is of tremendous importance for modern shared-memory multiprocessors, this issue has not attracted considerable interest in the literature. The works presented in [2, 3] for Cellular IRIX are notable exceptions.

Our work differentiates from previous proposals for shared-memory multiprocessor scheduling in two important aspects: First, we employ a different dynamic space sharing algorithm (DSS), which attempts to adapt more effectively to the actual processor requirements of parallel programs. Towards this goal, DSS exploits a shared arena interface which is in many aspects similar to the interface used in the Cellular IRIX operating system [3]. Second, we integrate DSS with simple yet effective time-sharing schemes, which enable our schedulers to meet traditional performance requirements such as increased throughput, increased utilization and reduced response time without however sacrificing parallelism.

3 Kernel-level Scheduling Algorithms

In this section, we describe our dynamic space and time-sharing kernel-level scheduling algorithms. In Section 3.1, we provide a brief overview of DSS, which is our baseline space sharing algorithm. In Section 3.2, we present three new algorithms that integrate DSS with time sharing.

3.1 Dynamic Space Sharing (DSS)

DSS [8] is a two-level scheduling policy. The first level space-shares efficiently the machine resources, processors and memory, among the running applications. The second level attempts to improve the memory performance of simultaneously running parallel programs, by exploiting the affinity of kernel threads for specific processors.

For processor partitioning, the algorithm takes into consideration the processor requirements of each application ρ_i, $i = 1, 2, \ldots, n$ and the overall system workload W. The aggregate system workload is calculated by Equation 1. This information is communicated to the kernel scheduler through a shared arena. The number of processors assigned to each application x_i, is in general a fraction of the total number of processors requested by that application $1 \leq x_j \leq \rho_j$ and it is a function of the total processor requirement of the entire workload and the number of physical processors (P) as shown in Equation 3. DSS guarantees that there are no applications waiting to starvation.

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

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

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

we define

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

Then x_i is given by:

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

At the lower level DSS uses Selective Scheduling, which implements a simple, though efficient heuristic that exploits the memory affinity of kernel threads for specific processors. During execution, a process establishes affinity in the cache and local memory of the processor on which the process runs more often [5]. Selective scheduling maintains some level of history for each kernel thread by tracking the processors on which the kernel thread ran in previous time quanta and attempting to reschedule the kernel thread on the same processors in subsequent time quanta.

3.2 Time–Sharing Mechanisms for DSS

DSS is our core space-sharing algorithm. Our target is to enhance DSS with a time sharing mechanism in order to achieve a complete, efficient and fully dynamic kernel-level scheduler for commercial parallel computers. Towards this direction, we describe next three time-sharing mechanisms for DSS, namely Sliding Window DSS (SW-DSS), Step Sliding Window (SSW-DSS) and Variable Time Quantum DSS(VTQ-DSS).

The idea in all three proposed mechanisms, is to give to each application the same opportunity in terms of system utilization. Namely, to have each application in the workload to consume the same CPU time compared to the other applications present in the system, "ignoring" the parallelism of each application, which is already handled effectively by DSS. The requirement of equally distributing the CPU time among the applications, tries to treat fair the short and serial processes compared to parallel ones, while at the same time preserving all properties of DSS.

The first two mechanisms (SW-DSS and SSW-DSS) are similar but they have different effects on the workload demonstrating therefore different properties. The third one (VTQ-DSS) is a more complex, though more sophisticated and elaborate compared to the other two policies.

3.2.1 Sliding Window DSS

SW-DSS partitions all processes waiting for execution in the ready queue in groups of processes called *gangs*. A gang w_i, is defined as a group of processes requesting a number of processors which is not more than $3P$, P the number of processors in the system. A mathematical definition of a gang, is given by Equation 4 and Inequality 5 bellow. The workload size index W equals the total number of processors requested by all gangs $w_i, i = 1, 2, \ldots, n$ as denoted in Equation 6. Each gang is submitted for execution for a predefined constant time quantum of 100msec. After the expiration of the time quantum the next, neighboring gang of processes in the ready queue is scheduled, and so on. We view this successive assignment of gangs from the ready queue as a moving *window* sliding from the head of the queue to the tail, back and forth, until all processes in the ready queue complete execution. Hereafter, we use the terms window and gang interchangeably.

$$w_i = \sum_{i=k}^{k+m} \rho_i \quad (4)$$

$I_i = \{\rho_1, \rho_2, \ldots, \rho_n\}$,
$I_j = \{\rho_k, \ldots, \rho_{k+m}\}, I_j \subseteq I_i$

$$\max_{I_j} w_i, P < w_i \leq 3P \quad (5)$$

$$W = \sum_{i=1}^{n} w_i \quad (6)$$

P, is the total number of processors in the system and w_i is the number of processors requested by the processes in the ith gang w_i. $w_i \subseteq W$ and $w_i = W$ when $j = i$.

The upper bound of processors requested from the processes in each gang is bound to $3P$ for the following reasons: The first reason is that even in the case of a heavy workload we want to exploit at least some amount of parallelism present in the applications. The second reason is that we have observed through experimentation that setting the upper bound to $3P$ helps most parallel applications in a gang to execute close to their "operating point,, i.e. close to the point where the applications execute on as many processors as they can effectively utilize and beyond which their performance would degrade. An upper bound of $3P$ resulted to the best performance for most of the workloads that we used to evaluate our scheduling algorithms.

At this point, we also need to define two more concepts for the exegesis of SW-DSS. The concept of *scheduling step* and *scheduling phase* or *cycle*. We define as a scheduling step, the scheduling of processes in a window w_i and as scheduling phase all scheduling steps until all gangs in the task queue have been scheduled once.

In each scheduling step, SW-DSS selects to schedule all processes included in the window defined by Inequality 5, as stated previously. The algorithm applies DSS for the selected processes and executes them once for a prespecified time quantum. In the next scheduling step the window moves towards the tail of the ready queue and the next gang of processes is selected to be scheduled with DSS. The algorithm is repeated until all applications in the ready queue have been scheduled. In the next, or in subsequent, scheduling cycles there might be that one or more of the processes in the queue terminate execution; in this case the window is expanded with the next process in the row, if the processor requirements of that process do not violate the constraint of $w_i \leq 3P$. In case the last gang in the tail of the task queue contains less processes that $3P$, the window is extended, by recycling the ready queue pointer and including processes from the head of the queue.

3.2.2 Step Sliding window DSS

The goal of SSW-DSS is to improve cache performance of simultaneously executing parallel programs. SSW-DSS acts as a supplementary to the affinity-conscious scheduling mechanism of DSS. SSW-DSS preserves all definitions that have been earlier specified for SW-DSS except that of the *scheduling step*. In SSW-DSS the scheduling step is not a discrete new window, like in SW-DSS. Rather, it is

exactly the same window as in the previous scheduling step leaving out the first process and replenishing the gang with none, one or more processes depending on what satisfies Inequality 5.

SSW-DSS is expected to improve over SW-DSS with respect to the memory performance of parallel programs, because of the way processes are scheduled. To show that, let us have the kth gang composed out of m processes $w_k = \{\rho_k, \rho_{k+1}, \ldots, \rho_{k+m}\}$. The last process in the kth gang will be scheduled m times repeatedly. This means that in each scheduling step we are going to have all but the last process in the gang to be scheduled again. With the support of DSS, this ensures that in each scheduling step the processors will have cache footprints of almost all the processes that are scheduled in the scheduling step.

3.2.3 Variable Time Qauntum DSS

VTQ is a fully dynamic scheduling policy in all of its layers. In VTQ the whole workload of processes is divided again into gangs of processes. VTQ adopts the notion of a gang as defined for SW-DSS and SSW-DSS. A new element in this policy is that we use in addition to the ready queue a *repository queue*.

Our major effort in this work is to propose a time sharing policy so that the CPU time consumed by each process in the workload is proportional to the size of the process in terms of its execution length. This is indeed what we have already accomplished by the previous two policies, SW-DSS and SSW-DSS. However, this was achieved through a mechanical, though efficient, way by the moving window. In VTQ we propose a formal quantitative analysis enabling us not only us to estimate a fair and effective allocation of CPU time among the processes of the workload, but also to maximize the utilization of system processors.

Our target is to derive a mathematical formula to calculate the time quantum for each process dynamically at run time, in order to allocate to all processes the same CPU time. We adopt the notion of a time quantum q_i as the CPU time allocation unit. For all processes scheduled for the very first time in the system we use a basic, uniform, time quantum of $q_i = 100msec$. The base time quantum is selected through experimentation. The total CPU time for one scheduling step, consumed by each process is the number of processors allocated by DSS to the process, times the assigned quantum for the current scheduling step $x_i q_i$. We define T_i as the accumulated CPU time consumed by the ith process during its execution lifetime. T_i is given by equation 7.

$$T_i = T_i + q_i x_i \tag{7}$$

The aggregate CPU time of all processes in a gang is:

$$T_{w_i} = \sum_{i=k}^{k+m} T_i \tag{8}$$

VTQ equates T_i's among the processes in each gang during execution. The equalization of CPU time among the processes of a gang is accomplished by varying the time quantum for each process, according to the formula in Equation 9. Equation 9 estimates q_i for all processes in each scheduling step.

$$q_i = \frac{T_{w_i}}{n \times q_{i-1}} \tag{9}$$

In VTQ the processes in the tail of the ready queue that do not compose a gang that satisfies Inequality 5, or the last gang of the ready queue in case where all gangs satisfy Inequality 5, are enqueued in the repository queue. Then, VTQ starts scheduling the processes of the first gang assigning initially a uniform time quantum of 100msec to each process in the gang. The first scheduling step in VTQ stands for one time quantum and in the next scheduling step the processes of the next gang in the ready queue are scheduled. However, after the first scheduling phase, the time quanta in all subsequent scheduling phases start varying in each scheduling step. The physical aftereffect of the variable time quantum is that processes belonging in the same gang may terminate execution in different time steps. In this case we define as a scheduling step the longest time quantum in the gang. In order to utilize the CPUs released by processes before the expiration of the longest time quantum of their gang, or before the expiration of their last time quantum, we schedule processes from the repository queue for the remaining time.

4 Evaluation Framework and Results

In this section, we present the experimentation environment used to evaluate our kernel scheduling methodologies, along with performance evaluation results from experiments conducted on a Silicon Graphics Origin2000.

4.1 Experimentation Environment

We performed our experimental evaluation using two parallel compilation and execution environments, the native environment the Cellular IRIX operating system and the NANOS environment[1], in which we implemented our kernel scheduling policies.

IRIX provides an execution framework in which applications are parallelized, either automatically or manually, trying to obtain good processor utilization when running

on a multiprogrammed NUMA multiprocessor. The MIPSpro SGI compilers support automatic detection of parallelism and programmer inserted OpenMP directives that command parallel execution, on top of the SGI MP runtime library. The MP library supports per-application control of the degree of parallelism. An auxiliary thread wakes up every three seconds and suggests an optimal number of kernel threads for the execution of the next parallel region, based on system load information. The application adjusts the number of kernel threads by blocking/unblocking some threads if necessary. Using this mechanism, application parallelism is decreased progressively when the system load is high or when CPU usage is low due to I/O or page faults. The dynamic thread control mechanism in IRIX is used with the standard time-sharing scheduler.

The NANOS automatic parallelization and execution environment is a prototype implementation of the NanoThreads Programming Model (NPM) [1], currently available for Silicon Graphics Origin2000 systems running the Cellular IRIX operating system. The core of the environment consists of a parallelizing compiler, a lightweight multithreading runtime system and a user-level CPU manager which emulates the operating system scheduler. The main components are accompanied with visualization tools for automatic parallelization and performance analysis. More detailed descriptions of NANOS can be found in [1].

The runtime library and the emulated kernel scheduler communicate through a shared arena [3, 8]. The coordination between the applications and the kernel scheduler aims at maintaining consistently a one-to-one mapping between the user-level threads of the application and the number of physical processors granted from the kernel scheduler to the application and ensure that the application will always make progress along its critical path.

The NANOS CPU manager is a multithreaded user-level process, which contains a regular shell for executing arbitrary workloads in the form of csh-like scripts and a scheduling module which emulates various kernel-level scheduling policies.

4.2 Experimental Evaluation

As a baseline for comparisons, we use the performance of the native Silicon Graphics environment, with dynamic thread control enabled for parallel applications. The experiments were performed on a Silicon Graphics Origin2000 with 64 MIPS R10000 processors clocked at 250 MHz, 32 Kilobytes of split L1 instruction and data cache per processor, 4 Megabytes of unified L2 cache per processor and an aggregate of 8 Gigabytes of main memory.

Benchmark	Parallelism	Size
HYDRO2D	functional & loop	20 time steps
SWIM	loop	512×512 grid
TOMCATV	loop	513×513 grid
TURB3D	functional & loop	64×64×64 cubic grid

Table 1: SPECcfp95 benchmarks used in the workloads.

Benchmark	Seq. Time	Exec. Time (oper. point)	
		IRIX	NANOS
HYDRO2D	17.91	6.08(16)	6.94(16)
SWIM	106.35	5.48(16)	5.02(16)
TOMCATV	72.83	8.31(16)	8.05(16)
TURB3D	260.27	28.26(16)	28.76(16)

Table 2: Execution times in seconds and operating points of the SPEC benchmarks used in multiprogrammed workloads.

4.2.1 Workloads and Metrics

We use multiprogrammed workloads consisting of parallelized versions of four benchmarks from the SPEC CFP95 benchmark suite [10], HYRDO2D, SWIM, TOMCATV and TURB3D. The benchmarks, together with their main characteristics are summarized in Table 1. All benchmarks are written in FORTRAN77 and they were parallelized manually, using OpenMP directives. Two parallelized versions for each benchmark were produced, one from the MIPS compiler used to execute in the native IRIX environment and one with the NANOS compiler used to execute in the NANOS environment.

Our methodology for evaluating kernel scheduling policies is to run multiprogrammed workloads with each instance of a benchmark in a workload requesting a number of processors that corresponds to the *operating point* of the benchmark. We computed the operating points of the benchmarks by running them in a dedicated environment. The results are illustrated in Table 2. Using the operating point as a directive for requesting processors, we ensure that each application attempts to run on as many processors as it can effectively utilize and we can isolate the performance impact of multiprogramming and kernel scheduling on parallel applications. Table 2 reveals also that the performance of the two parallelized versions of each benchmark is comparable. Our results with multiprogrammed workloads are therefore expected to reflect directly the relative performance of the kernel schedulers.

The number of instances of each benchmark in the multiprogrammed workload is selected in a way that ensures a fixed minimum degree of multiprogramming M during the execution of the workload. For the purposes of the evaluation we modeled a moderately and a heavily loaded multi-

programmed system, with $M = 2$ and $M = 4$ respectively.

In order to introduce constant variability in the workloads, we added in each workload a background synthetic benchmark with a *diamond* task graph [8]. This benchmark executes through different parallel phases and requests a different number of processors in each phase, for a number of iterations.

We used two types of workloads in the experiments, *homogeneous* and *heterogeneous* workloads. Homogeneous workloads consist of multiple instances of the same benchmark. Heterogeneous workloads consist of multiple instances of multiple benchmarks. Due to space considerations, we present only the results for homogeneous workloads executed on a heavily loaded multiprogrammed system. The rest of the results are available in the expanded version of this paper [9].

We evaluated a six scheduling policies: The native Cellular IRIX time-sharing scheduler (*irix-mp*), running benchmarks parallelized with the SGI MP library and the dynamic process control feature activated, an emulation of a flat gang scheduling scheme [4] in the NANOS user-level CPU manager (*gang*), a dynamic equipartition strategy similar to the one in [6], implemented in the NANOS user-level CPU manager (*equip*), and the sliding window (*sw-dss*), step-sliding window (*ssw-dss*) and variable time quantum DSS (*vtq-dss*) policies described in Section 3. All policies use a base time quantum of 100 milliseconds.

The two primary metrics used in our evaluation are application *turnaround time* and system *throughput*. In order to measure turnaround times, we executed multiprogrammed workloads in *open* system mode. In this mode, all instances of the benchmarks in a workload are fired simultaneously and the turnaround time of each instance is measured independently. For homogeneous workloads, we measure also the *workload completion time*. If an ideal fair scheduler were used, the turnaround time of all instances of the benchmark would be equal to to the workload completion time. Therefore, the difference between the workload completion time and the average turnaround time in homogeneous workloads provides a rough indication of fairness.

In order to compute system throughput, we executed the workloads in *closed system mode*. In this case, several instances of each benchmark are fired again simultaneously. However, upon completion of an instance of a benchmark, another instance of the same benchmark is fired immediately in the place of the completed instance. This is repeated for a number of iterations, and ensures the maintenance of a constant multiprogramming load for a sufficiently long period of time. The first few and last few executions of each instance of a benchmark are discarded and the turnaround times of the intermediate executions are averaged to calculate throughput.

In order to evaluate the schedulers quantitatively with a single value we use the *normalized throughput* of the schedulers. The normalized throughput of a scheduler is defined as the ratio of the throughput of the scheduler to the throughput of an "ideal,, scheduler, or equivalently the ratio of the average turnaround time with an "ideal,, scheduler to the average turnaround time with the scheduler under study. Given a scheduler s, and n benchmarks $i = 1 \ldots n$, executed in a multiprogrammed environment with a multiprogramming degree M, the normalized throughput of s can be defined as:

$$\overline{\lambda}_{s,M} = \frac{1}{n} \sum_{i=1}^{n} \overline{\lambda}_{s,M,i} \tag{10}$$

where:

$$\overline{\lambda}_{s,M,i} = \frac{\lambda_{s,M,i}}{\lambda_{ideal,i}} = \frac{\overline{T}_{ideal,i}}{\overline{T}_{s,i}} \tag{11}$$

In order to circumvent the problem of defining the "ideal,, scheduler we define as "ideal,, a scheduler which is able to execute each benchmark under multiprogramming, in time equal to the time that the benchmark takes on a dedicated system at its operating point.

4.2.2 Experimental Results and Interpretation

Figure 1 illustrates the results from executions of the homogeneous workloads in open system mode. The new dynamic time and space-sharing schedulers (*sw-dss, ssw-dss, vtq-dss*) outperform consistently the IRIX scheduler. For HYDRO2D under the three new schedulers achieve more than a 2-fold speedup of turnaround time compared to *irix-mp*. The speedups of turnaround time for the rest of the benchmarks range from 11% to 40%.

There are several reasons that explain the relative performance of the *dss*-based schedulers. The instantaneous processor allocation in the *dss*-based schedulers is a global decision made by the kernel scheduler, influenced from the overall system load and the relative weight of each job in the workload (i.e. its relative processor requirements). This approach ensures fairness in terms of the number of processors allocated to each application. On the other hand, in the native IRIX environment, the number of processors which the application decides to use for a parallel loop is a local decision of the application, made within the MP library. Although this decision uses also system load information, it does not take into account the relative weight of the application in the system. This approach penalizes some instances of the benchmarks, which detect that the system load is high and decrease the number of processors on which they execute parallel code.

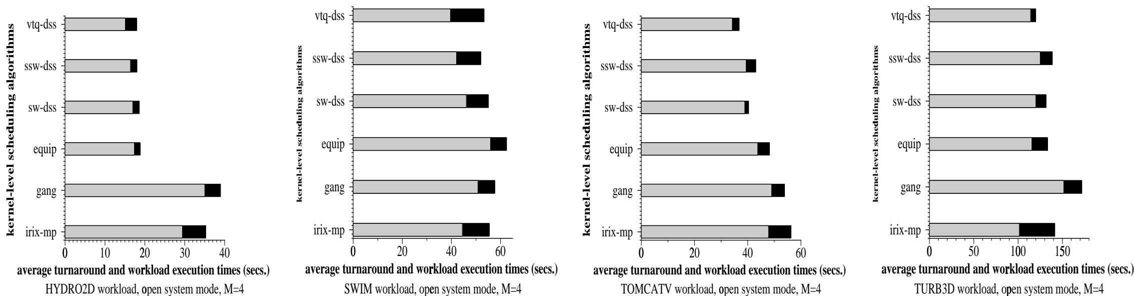

Figure 1: Performance of the kernel-level schedulers with homogeneous workloads in open system mode. The black region corresponds to the difference between the average turnaround time and the workload completion time. The results are averages of three executions.

The shared memory interface used by the *dss*-based schedulers ensures instantaneous response and application adaptability to changes in the number of processors allocated to an application. On the other hand, the MP library uses a polling mechanism with a 3-second polling interval to readapt to changes of the system load. This strategy hurts the performance of fine-grain parallel applications as well applications with frequent transitions between sequential and parallel phases.

The memory performance of the benchmarks under the *dss*-based schedulers is improved. Both the IRIX and our kernel schedulers apply a strong affinity scheduling discipline, that tends to bind each kernel thread to a specific processor node. Our experimental evidence however indicate that IRIX forces more thread migrations, directed primarily from an attempt to trade individual application performance with better processor utilization. We traced this effect using the NANOS performance analysis tools. For memory-intensive workloads reducing kernel thread migrations plays a catalytic role in performance.

Among the three *dss*-based policies, *vtq-dss* seems to be the policy of choice, since it delivers the best performance in terms of turnaround time for 3 out of 4 homogeneous workloads in open system mode. The *ssw-dss* policy seems to be inferior to the other *dss*-based policies. It appears that the longer waiting times experienced by some benchmarks in the workloads overwhelm the benefits of having some benchmarks rescheduled on the same processors for longer time. The equipartition policy performs also well, although its performance is always inferior to at least one of the *dss*-based policies. Gang scheduling seems to be quite effective with a moderate multiprogramming degree, however its performance degrades rapidly as the multiprogramming degree increases.

Concerning fairness, for the *dss*-based schedulers, the difference between the average turnaround and the workload completion times is on average 14.6% with the exception of the SWIM workload, where the difference is 35%. The corresponding value for *irix-mp* is 16%.

Schedulers	Normalized Throughput	
	$\lambda_{s,2}$	$\lambda_{s,4}$
irix-mp	0.34	0.19
gang	0.38	0.19
equip	0.48	0.25
sw-dss	0.50	0.26
ssw-dss	0.49	0.27
vtq-dss	0.52	0.29

Table 3: Normalized throughput of the kernel-level schedulers.

Figure 2 illustrates the average turnaround times of the benchmarks in homogeneous workloads executed in closed system mode. The comparative results are analogous to the results from the experiments in open system mode. However, IRIX demonstrates a solid improvement over the results in open system mode, without outperforming the *dss*-based schedulers though. The IRIX earnings[1]-based scheduling is very effective in dealing with the unbalances incurred from poor initial processor allocation decisions made in the MP library, in the long term.

We use the results from homogeneous workloads in closed system mode, to compute the normalized throughput of the kernel schedulers under different degrees of multiprogramming, using equations 10 and 11. Table 3 illustrates the results. *Vtq-dss* has the best normalized throughput for both values of M, followed by *sw-dss* and *ssw-dss*. *Equip* has competitive throughput compared to the *dss*-based policies, while *irix-mp* is inferior.

5 Conclusions

This paper presented a set of kernel-level scheduling algorithms that combine dynamic space sharing with time-sharing to ensure efficient execution of parallel pro-

[1] The term "earnings" is defined as the accumulated expendable CPU time of a job, relative to the other jobs in the system

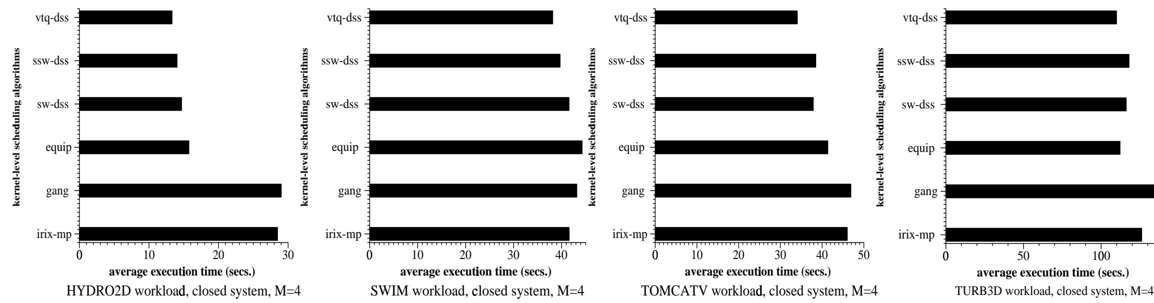

Figure 2: Average turnaround times of the benchmarks in homogeneous workloads executed in closed system mode. The results are averages of three executions of the experiment.

grams in multiprogrammed environments without compromising fairness and global system performance. We presented three algorithms, (sliding-window DSS, step-sliding-window DSS and variable time quantum DSS), of increasing sophistication and complexity and evaluated their performance on a 64-processor Silicon Graphics Origin2000, running the Cellular IRIX operating system. The three new scheduling policies demonstrated a solid improvement over the performance of the native Cellular IRIX environment for multiprogrammed execution of parallel programs, even though our policies were evaluated with user-level emulation of the kernel scheduler.

Our current work is oriented towards integrating priority control mechanisms in our schedulers, as well as porting our schedulers to real operating system kernels. We are also attempting to conduct a more thorough evaluation of the schedulers, using mixtures of sequential and parallel applications, as well as applications parallelized with flat parallel programming models that do not interact with the kernel scheduler to adapt to the available resources.

Acknowledgements

We are grateful to Constantine Polychronopoulos and our partners in the NANOS project. This work is supported by the European Commission, through the ESPRIT Project No. 21907 (NANOS).

References

[1] The NANOS Project Consortium. NANOS: Effective Integration of Fine-Grain Parallelism Exploitation and Multiprogramming. ESPRIT Project No. 21907 (NANOS), http://www.ac.upc.es/NANOS, 1999.

[2] J. Barton and N. Bitar. A Scalable Multidiscipline, Multiprocessor Scheduling Framework for IRIX. *Proc. of the 1st IPPS Workshop on Job Scheduling Strategies for Parallel Programming*, LNCS Vol. 1459, Santa Barbara, CA, 1995.

[3] D. Craig. An Integrated Kernel- and User-Level Paradigm for Efficient Multiprogramming. Master's Thesis, CSRD Technical Report No. 1533, University of Illinois at Urbana-Champaign, 1999.

[4] D. Feitelson. A Survey of Scheduling in Multiprogrammed Parallel Systems. Research Report RC 19790 (87657), IBM T.J. Watson Research Center, Revised Version, 1997.

[5] A. Gupta, A. Tucker and S. Urushibara. The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications. *Proc. of the 1991 ACM SIGMETRICS Conference*, pp. 120–132, San Diego, USA, 1991.

[6] C. McCann, R. Vaswani and J. Zahorjan. A Dynamic Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, Vol. 11(2), pp. 146–178, 1993.

[7] J. Ousterhout. Scheduling Techniques for Concurrent Systems. *Proc. of the Second Distributed Computing Systems Conference*, pp. 22–30, 1982.

[8] E. Polychronopoulos, X. Martorell, D. Nikolopoulos, J. Labarta, T. Papatheodorou and N. Navarro. Kernel-Level Scheduling for the Nano-Threads Programming Model. *Proc. of the 12th ACM International Conference on Supercomputing*, pp. 337–344, Melbourne, Australia, 1998.

[9] E. Polychronopoulos, D. Nikolopoulos, T. Papatheodorou, X. Martorell, N. Navarro and J. Labarta. An Efficient Kernel-Level Scheduling Methodology for Shared-Memory Multiprocessors. Technical Report HPISL-010399, High Performance Information Systems Laboratory, University of Patras, 1999.

[10] Standard Performance Evaluation Corporation. SPEC CPU95 Benchmarks. Available at http://www.spec.org/osg/cpu95, 1995.

[11] K. Yue and D. Lilja. Dynamic Processor Allocation with the Solaris Operating System. *Proc. of the First Merged IEEE IPPS/SPDP Conference*, pp. 392–397, Orlando, FL, 1998.