An OpenMP Implementation for Multiprogrammed SMPs

Vasileios K. Barekas, Panagiotis E. Hadjidoukas,
Eleftherios D. Polychronopoulos and Theodore S. Papatheodorou

High Performance Information Systems Laboratory,
Department of Computer Engineering and Informatics,
University of Patras, Rio 26500, Patras, Grecce
phone:+30-61-993805, fax:+30-61-997706
{bkb,peh,edp,tsp}@hpclab.ceid.upatras.gr
http://www.hpclab.ceid.upatras.gr

Abstract

In this paper, we present an OpenMP implementation designed for multiprogrammed environments on Intel-based SMPs. This implementation consists of a runtime system and a resource manager, while we use the NanosCompiler to transform OpenMP applications into code with calls to our runtime system. The resource manager acts as the operating system scheduler for the applications built with our runtime. The runtime system cooperates with the resource manager in order to adapt each application's generated parallelism to the number of processors allocated to it, according to the resource manager scheduling policy. We use the OpenMP version of the NAS Parallel Benchmark suite in order to evaluate the performance of our implementation. In our experiments we compare the performance of our implementation with that of a commercial OpenMP implementation. The comparison proves that our approach performs better both on a dedicated and on a multiprogrammed environment.

1 Introduction

The OpenMP Application Programming Interface [9] provides a simple and portable model for programming a wide range of parallel applications on parallel plarforms. Everyone is able to use this quite simple standard in order to parallelize applications without paying attention on the underlying architecture. The OpenMP API is portable across a wide range of parallel platforms, including small-scale SMP servers, scalable ccNUMA multiprocessors and clusters of workstations. The simplicity of this model derives from the fact that the programmer does not need to worry about the details of the underlying platform or the operating system mechanisms. The programmer simply inserts directives into its original sequential code in order to annotate loops and sections of code that can be executed in parallel. The OpenMP support is transparent to the user through an OpenMP-capable compiler and the necessary runtime support. The compiler interprets the user-inserted directives into appropriate runtime calls that enable the application to execute in parallel by multiple threads using a fork/join execution model. For every part of the code that has to be executed in parallel, a group of threads is created and a chunk of the total work is assigned to each thread according to a pre-defined scheduling scheme. These threads are scheduled on the available physical processors by the operating system scheduler.

On the other hand, multiprocessor systems are also used as multiprogrammed compute servers, where several users submit parallel and sequential CPU-intensive applications. The threads of all these applications contend for systems's available resources. Since general-purpose operating system schedulers are designed to deal with sequential applications, they fail to achieve good performance when workloads consisting of multiple parallel applications run on the system. This fact is due to the scheduler ignorance about the nature of the parallel applications and their requirements. It is therefore crucial to provide efficient mechanisms that enable parallel applications to achieve robust performance in multiprogrammed environments.

An efficient way to achieve better performance with OpenMP applications that are executed in multiprogrammed environments is the utilization of a sophisticated runtime system. The runtime system enables parallel applications to adapt themselves to the resources made available for them by the operating system. Using this runtime system, parallel applications exploit the amount of parallelism that matches the number of processors that the operating system scheduler allocates to them. This approach requires close collaboration between the operating system scheduler and the runtime system, in order to keep each part informed about the requirements and/or the decisions of the other part. Although the OpenMP standard already makes provision for the use of dynamic parallelism, several OpenMP implementations either do not support it or they support it in a very limited way. In those implementations, the runtime system creates and uses as many threads as each application initially requests, without taking into account the variations of the system workload during the execution time. This is due to the limitations that the operating system scheduler set, and the lack of collaboration between the scheduler and the runtime system.

In this paper, we present an OpenMP implementation consisting of a runtime system and an independent resource manager, which plays the role of the operating system scheduler. The resource manager provides to the runtime system the functionality needed in order to adapt the application parallelism to the available resources. The collaboration between these two parts make our implementation capable to support the execution of OpenMP applications in a multiprogrammed environment. We utilize the NanosCompiler [2] to transform the original OpenMP-coded applications into code that utilizes our runtime system. The runtime system is based on the NTLib multithreaded runtime library [3]. NTLib is a highly optimized library developed for the Windows NT/2000 operating system, which provides lightweight user-level threads and the necessary support to run parallel applications on Intel-based SMP systems. We have extended the runtime library in order to support the code that the compiler injects in the original OpenMP-coded application. We use a portable Resource Manager [4], which is implemented as a loadable kernel-mode driver. The resource manager communicates with the applications through a shared memory area, lying between the operating system kernel space, where the resource manager runs and the user space, where the applications run. The implementation of the resource manager as a loadable kernel module provides, several advantages such as portability and efficiency. So, we can take advantages of an in-kernel implementation avoiding the need to modify or recompile the kernel. The original resource manager was extended in order to adapt its behavior to the peculiarities of OpenMP applications when they are executed in a multiprogrammed environment.

The runtime system in combination with the resource manager provides an environment capable to efficiently execute OpenMP applications on multiprogrammed systems. The functionality of our environment is provided transparently to the programmer of the OpenMP applications. In order to evaluate our approach we have used the OpenMP implementation of the NAS Parallel Benchmarks suite [5]. The experiments on a quad-processor SMP system show that our implementation has a significantly better performance than a commercial OpenMP implementation, both on a dedicated and on a multiprogrammed environment. On a multiprogrammed environment, we also perform better both in the turnaround time of the several applications in the workload executed and the total execution time of the whole workload. Furthermore, in the case of the execution of an application on a dedicated environment, using only our runtime library, we show that it achieves comparable or better speedups to that of a commercial OpenMP compiler.

The rest of this paper is organized as follows: Section 2 describes the features of the runtime system. In Section 3, we present the functionality of the resource manager. Next, in Section 4, we describe how OpenMP applications utilize our environment. In Section 5, we present our experiments that validate the efficiency of our implementation. Finally, in Section 6 we provide our conclusions together with the related work.

2 Runtime System

NTLib is a multithreaded runtime library that provides lightweight user-level threads and the necessary support to exploit, with minimal overhead, the parallelism found in applications. The lightweight threads enable the library to efficiently exploit fine-grain parallelism, while its dependence-driven execution model makes it capable to exploit multiple levels of parallelism. The cost of the user-level thread primitives is very low compared to other thread packages. The runtime library exports a set of functions that supports the nanothreads programming model [7]. These functions are responsible for the thread management, the handling of the ready thread queues, the control of the dependencies between the threads and the initialization of the runtime environment. More details about the implementation and the functionality of the NTLib runtime library can be found in [3]. The set of the exported functions and the functionality of the library was extended in order to support the compiler-generated code that corresponds to the original OpenMP application code. Additionally, support was added to enable the use of the runtime functionality with applications written in the Fortran programming language.

The NanosCompiler utilizes user-level threads to express multiple levels of parallel loops and parallel sections of code. However, in order to express the parallelism of single-level loops the compiler uses the work descriptor structure. We implement in our runtime system the work descriptor structure and the necessary support functions, based on the proposed implementation in [8] with the appropriate modifications to fit in our underlying architecture. A work descriptor structure consists of a pointer to the function that encapsulates the work to be executed and its arguments. Work descriptors provide a very efficient mean to distribute work among the participating processors. For each single-level parallel loop, the master processor creates a work descriptor and initializes it for the specific loop. Then, the master processor distributes the work descriptor to all slave processors that participate in the execution of the specific loop, by copying it to a specific memory location for each processor. Each slave processor, checks its memory location in order to find a work descriptor, and execute it. The part of the work descriptor that each processor executes, is determined by the function arguments, the number of processors participating in the execution and the processor-own identifier.

The runtime library executes the created threads based on a dependence-driven execution model, in order to ensure the correct execution order of the created threads and to allow the efficient exploitation of multiple levels of parallelism. For each thread, we keep information about its dependencies with other threads. When a thread finishes its execution, it satisfies one input dependency on each one of its successors. When a thread creates a new thread, the dependencies of the creator thread must be preserved. For this reason, we must increase by one the creator thread dependencies and declare it as the successor of the new thread. By this way, we can create multiple levels of nested threads making thus easy the representation of multiple levels of parallelism found in a wide range of applications.

The main responsibility of the runtime library is to control the generated parallelism, ensuring that it matches the number of processors allocated by the operating system to each application. In other words, the runtime library implements dynamic program adaptability to the available resources by adjusting the amount of the generated parallelism. The scheduling policy of the operating system is concerned with the allocation of physical processors to the applications currently running in the system. The runtime library cooperates with the operating system scheduler, in order to achieve the desired adaptation. In our implementation, we use an external resource manager to play the role of the operating system scheduler. The operating system provides kernel-level threads to the runtime system, as the kernel abstraction of physical processors on which applications can execute. These kernel-level threads play the role of the virtual processors, which will execute the application's user-level threads.

3 Resource Manager

In this section, we present the functionality provided by the resource manager to the runtime system. The resource manager is implemented as a loadable kernel-mode device driver; this gives us the ability to start and stop its execution at our will. The primary responsibility of the Resource Manager is to keep track of all applications running on the system and to apply for them a user-defined scheduling policy. The scheduling policy mainly determines the number of physical processors that will be allocated to each application. In order to achieve this, a shared memory area that is maintained between the applications and the resource manager, acts as the communication path between them. At its initialization phase, the manager creates a shared memory section, which is mapped in the address space of each application that utilizes the runtime system. For each active application in the system, there is some information in the shared memory area, where both the applications and the manager have access. The information kept in the shared area concerns the actual parallelism of the application each time, the virtual processors identifiers that the application uses and other useful data. The resource manager based on this information calculates the number of physical processors that will be allocated to each application each time, and controls the execution of the virtual processors. When an application starts, it maps the shared memory area in its private address space. Next, the application's main virtual processor creates the rest virtual processors in suspended mode and registers them into the shared area, making them accessible from the resource manager and enabling thus their manipulation from it. After this point, the control of the application's virtual processors has passed exclusively to the resource manager, which is responsible for their execution.

The resource manager, based on the information in the shared memory area, applies a user-defined scheduling policy to distribute the available physical processors across the registered applications. The scheduling policy is executed periodically at a fixed time interval called scheduler quantum. Both the scheduling policy and the scheduler quantum are user-defined and can be changed dynamically at runtime. The scheduling policy determines the number of processors that will be allocated to each application during the next scheduling quantum and which physical processors they will be. The latter is decided based on a per-application history that records the processors on which the application runs during the previous quanta. Additionally, it decides which virtual processors will run on these physical processors. The resource manager applies the scheduling policy decisions on the virtual processors of the applications, by allowing some of them to run by resuming their execution, while preventing some others by suspending them. Eventually, resource manager maintains the total number of running virtual processors equal to the number of physical processors in the system.

The running applications cooperate with the resource manager during their execution period in order to minimize their execution time by avoiding the idling of physical processors while exists some application that could execute useful work on them. Before entering a parallel region, each application informs, through the shared memory area, the resource manager about its requirements reflecting the actual degree of parallelism that it can exploit in this region. The resource manager responds to the application's requirements and allocates physical processors to it according to the current scheduling policy. The application receives, through the shared memory area, the resource manager's decisions and tries to match the parallelism that it will generate to the number of physical processors currently allocated to it. Based on the requirements of all the applications and the scheduling policy, the resource manager decides about physical processor reallocations between the applications. The removal of a physical processor from an application means the blocking of the associated virtual processor, while the assignment of a physical processor to an application means the unblocking of an application's virtual processor and its binding to the specific physical processor. This reallocation procedure can block a virtual processor at an unsafe point, for example while being inside a critical section, preventing the application to make progress on its critical path. In order to eliminate these undesirable preemptions, each virtual processor checks for any non-safely preempted virtual processor whenever it reaches a safe-point, where it is known that the application and runtime synchronization constrains are satisfied. If such a virtual processor is found, the currently running virtual processor yields its physical processor to the preempted one.

The general design of the resource manager allows its collaboration with the native operating system scheduler. The resource manager controls the execution of the virtual processors (kernel threads) of the applications built with our runtime library, by allowing some of them to run, with preventing some others. It does not disturb the kernel threads of the rest applications that are executed in the same time on the system. The virtual processors that the resource manager allows to run contend with the kernel threads of the other applications for the physical processors. The operating system scheduler decides which of the virtual processors that are allowed to run, will actually run on the physical processors. By this way we treat all the running applications with fairness, without violating the operating system scheduling policy primitives. The collaboration of the resource manager with the several applications running on the system and its cooperation with the operating system scheduler is depicted in Figure 1. In this Figure we have three applications, one of them is independent, while the other two are controllable by the resource manager. Resource manager scheduling policy decides to grant two physical processor to each one of its applications. But the operating system scheduler ignores it and schedules on physical processor 1 a kernel-level thread of the independent application. More details about the design and the implementation of the resource manager can be found in [4].

4 OpenMP Application Transformation

In this section, we describe how applications with OpenMP directives use the functionality offered by our environment. We use the NanosCompiler to compile applications that are parallelized with OpenMP directives. NanosCompiler front-end is capable to convert applications with OpenMP directives into equivalent applications that use calls to our runtime library in order to express the annotated parallelism.

Each application, during its initialization informs the resource manager about the maximum parallelism that it is capable to exploit dur-

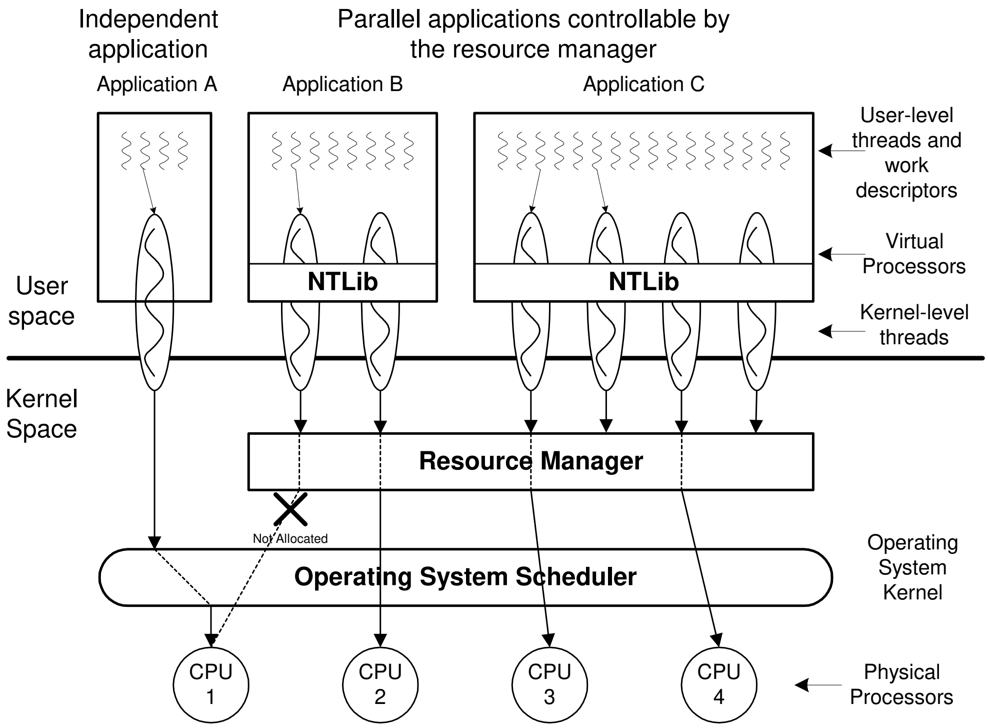

Figure 1: The resource manager cooperation with the applications and the operating system scheduler.

ing its execution. When NanosCompiler reaches a combined parallel work-sharing construct (`OMP PARALLEL DO`), which is the common case in the applications examined in this paper, it generates two functions that substitute the construct and its body. The first function replaces the parallel construct, while the second function contains the code of the parallel loop body. In the first function, the compiler inserts code that consults the resource manager, to find out how many physical processors are allocated to the application in order to execute this region. The number of processors that are allocated to the application is determined through the shared memory area, that the resource manager uses for the communication with the application. The first function is executed exclusively by the master processor, which creates a work descriptor structure that represents the body of the parallel loop. This work descriptor specifies the second compiler-generated function as the one that represents the loop that must be executed in parallel. Next, the master processor distributes the created work descriptor to all the available processors including itself. The work descriptor is distributed by coping it to a per-processor specific memory location. At this point, the master virtual processor suspends the execution of the first function in order to execute its own part of the work descriptor. Each one of the slave processors, whenever it becomes idle, it searches for a new work descriptor into its own memory location. Each processor calculates the work descriptor chunk that it will execute, based on its identifier and the total number of processors participating in the execution of this region. When all processors finish the work descriptor execution, the master virtual processor resumes the execution of the suspended function, while the slave processors remain idle.

When the compiler arrives at a parallel region construct (`OMP PARALLEL`), it again generates two functions that substitute the parallel region code. Additionally, in this case the compiler generates another one function for each work-sharing construct (`OMP DO` or `OMP SECTION`) that exists inside the parallel region. In this case the master processor does not create a simple work descriptor structure that all available virtual processors executes. Instead, it creates as many user-level threads as the number of the allocated physical processors. The created threads are executed on the available processors through their local ready queues. Each one of these threads, execute the compiler-generated functions that represent the work-sharing constructs that resides into the parallel region.

The original implementation of the resource manager was designed to work with general parallel applications, where no distinction is made between the processors of the application, since all of them are considered equal. Some modifications in the resource manager design were required in order to work properly with the fork/join model of the OpenMP applications. This model defines a master processor, which executes different code than the other processors. Since this code belongs to the critical path of the application the resource manager must treats the master processor with more responsibility. This means that the master processor must be always running, independently of the number of physical processors assigned to the application at any time. So, in the extreme case under multiprogramming, where a single physical processor is shared among several OpenMP applications, the only virtual processor that each application executes on this physical processor, each time, is its master processor. So, each application executes through its critical path, if later another physical processor becomes available to the application some of the other slave processors will be assigned for execution on this physical processor.

5 Performance Evaluation

In this section we present the experiments that we have conducted in order to evaluate the performance of our implementation. In our experiments we used the NAS Parallel Benchmarks suite, parallelized with the OpenMP programming model. The NAS Parallel Benchmarks is a very popular benchmarking suite, written in Fortran 77, which is often used for the performance evaluation on multiprocessors systems. In all experiments we used the class W problem sizes for all the applications. All the experiments were performed on a quad-processor 200 MHz Pentium Pro system equipped with 512 MB of main memory, running the Windows 2000 operating system.

We compare the performance obtained by the applications when we built them using our environment (the NanosCompiler, the runtime library and the resource manager) against the performance obtained when we built them with the PGI Workstation compiler version 3.2 [11]. The PGI Workstation is a commercial compiler product which supports the OpenMP standard. It takes as input either C or Fortran code parallelized with OpenMP directives and produces the final executable. In order to produce the final executable for the intermediate code that the NanosCompiler front-end outputs, we linked it with the NTLib library using the Intel Fortran compiler version 4.5 [6]. The NAS Parallel Benchmarks suite consists of seven applications, each one of them solving a different problem. In our experiments we run all of them, except the LU application that the PGI compiler did not build. We performed two kind of experiments, in the first we executed a single instance of an application on a dedicated system. In the second experiment, we ran each application on a multiprogrammed environment, which we simulated by running multiple instances of the same application. For each application, we built two versions, one with the PGI compiler and one with our implementation and we compared their performance in the two experiments.

In our first experiment, we measured the execution time for each version of the applications on a dedicated system with 1, 2, and 4 processors. We executed each application 5 times, the average execution times are presented in Table 1, while the equivalent speedups of the applications for each case are depicted in Figure 2.

As it is shown in Table 1, both versions of all the applications scale similarly when they are executed with up to two processors. On the four-processor execution three of the six application built with the PGI compiler failed to scale as well as the Nanos versions. More specifically, for the execution of BT, MG and SP on four processors there is a slowdown over the sequential execution and the variation between the execution times is quite high. We believe that this is due to the incorrect stack alignment of the data into the code that the PGI compiler produces or a problem into the runtime library that the PGI compiler uses. The other three applications CG, EP and FT does not suffer from this problem and they scale quite good on every processor count. On the contrary, the Nanos version of all the applications achieve good speedup in all cases. Even though the Nanos versions achieve better execution time in all the cases, their speedup is comparable to the PGI version. The conclusions from this experiment is that our implementation achieves comparable or even better speedups to that of a commercial OpenMP compiler, on a dedicated execution environment. Additionally, our implementation is more reliable and more stable than the PGI compiler, achieving robust performance with every application we tested.

In our second experiment, we compare the performance of our implementation on a multiprogrammed environment. In order to simulate a multiprogrammed environment in our system we ran several workloads, each one consisting of multiple instances of the same application. In each workload, all the instances were started simultaneously, at the same time. Each instance of the application assuming that it is running on a dedicated

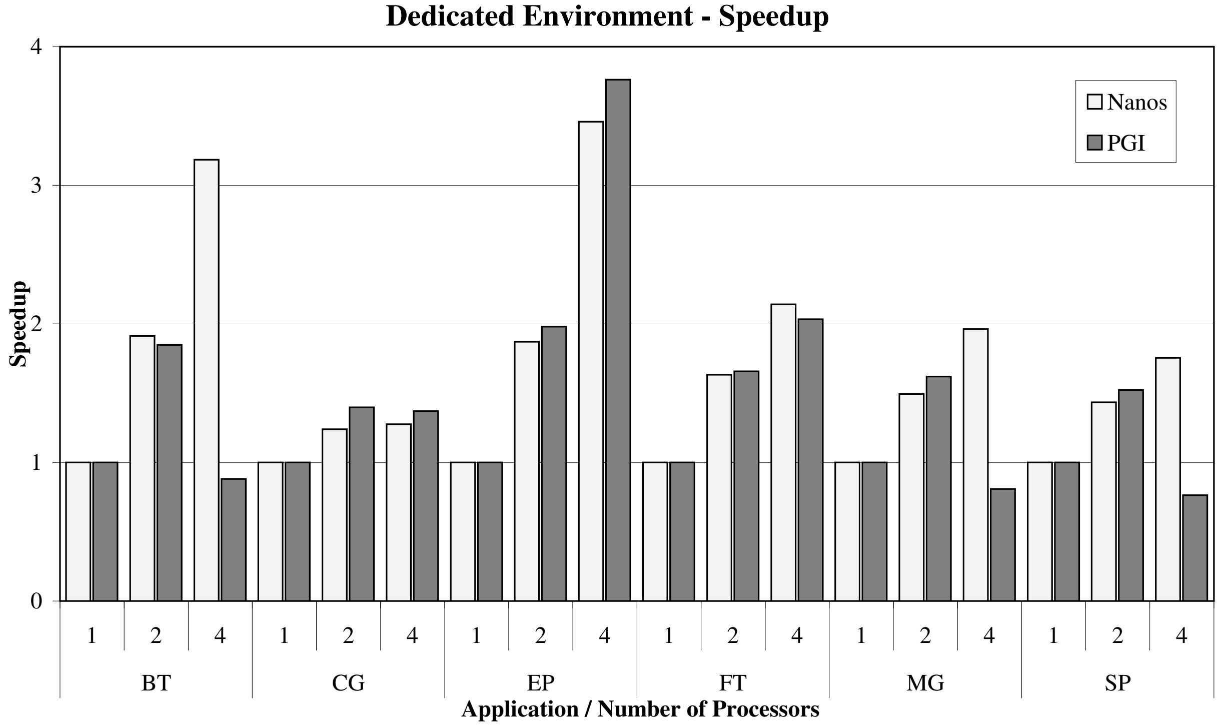

Figure 2: The speedup that the applications achieve on a dedicated execution environment.

# proc.	BT Nanos	BT PGI	CG Nanos	CG PGI	EP Nanos	EP PGI	FT Nanos	FT PGI	MG Nanos	MG PGI	SP Nanos	SP PGI
1	102.9	107.7	16.6	18.6	52.3	62.0	9.0	11.4	16.2	16.2	272.8	309.7
2	53.8	58.3	13.4	13.3	27.9	31.3	5.5	6.9	10.8	10.0	190.1	203.4
4	32.3	122.2	13.0	13.5	15.1	16.4	4.2	5.6	8.2	20.0	155.5	406.1

Table 1: The average execution times of the applications on a dedicated execution environment.

machine requested four processors from the system. The number of instances that we run in each workload determines the degree of multiprogramming that we want to achieve. We run workloads with multiprogramming degree of 1, 2, 4, and 8 for each application. For our implementation, we executed the workloads while the resource manager was running in the system in order to schedule the running applications on the available physical processors. The resource manager ran with a time-sharing version of the DSS scheduling policy [10] and a scheduling quantum of 120 msec, which matches the quantum used by the operating system scheduler for the kernel-level threads. For the PGI version, we executed the workloads without the resource manager. In this case the operating system scheduler was responsible for the scheduling of the applications on the available processors. We ran each workload 5 times, and we measured the average turnaround time for all the instances in each workload. The measured times for each case are presented in Table 2. In Figure 3 the measured times are depicted using a logarithmic scale for the time axis.

As shown in Table 2 and Figure 3 the workloads with the Nanos version applications achieves much lower turnaround times. More specifically, when we use the resource manager to coordinate the execution of the multiprogramming workloads, the turnaround time scales linearly to the degree of multiprogramming. This means that when we run a workload with multiprogramming degree 2 it requires approximately twice the time of a uniprogramming workload. In the PGI case, the workloads fail to scale well, because the operating system scheduler doesn't schedule them efficiently, since it has no knowledge about the requirements of their applications. The workloads of the three PGI-compiled applications that didn't scale well on the dedicated execution, now achieve superlinear scaling as the multiprogramming degree increases. This happens because the extra

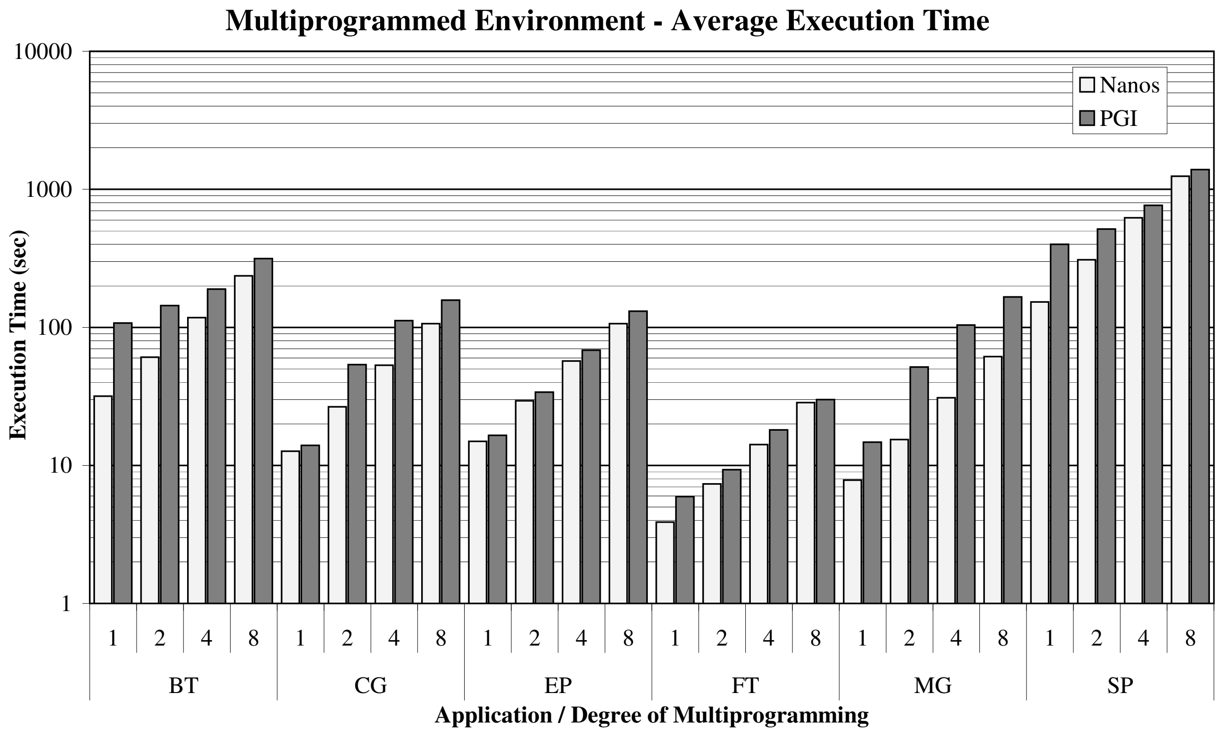

Figure 3: The average turnaround times of the applications in each workload on a multiprogrammed execution environment.

Inst.	BT		CG		EP		FT		MG		SP	
	Nanos	PGI	Nanos	PGI	Nanos	PGI	Nanos	PGI	Nanos	PGI	Nanos	PGI
1	31.7	107.3	12.6	13.9	14.9	16.5	3.8	5.9	7.8	14.7	152.5	399.3
2	60.7	143.7	26.5	53.7	29.3	33.9	7.3	9.3	15.3	51.5	308.5	516.2
4	117.5	189.2	53.1	111.7	57.0	68.3	14.1	18.0	30.9	104.0	621.2	764.3
8	235.8	315.2	106.4	157.4	106.3	131.1	28.5	30.0	61.4	166.1	1245.8	1387.9

Table 2: The average turnaround times of the applications in each workload on a multiprogrammed execution environment.

instances fill in the time intervals that the processors otherwise would be left idle, since each instance does not manage to fully utilize all of its processors. However, in any case our implementation achieves better turnaround times for the applications participating in the workloads. The improvement ranges from 6% to 63% less time that the PGI case. Additionally, in the Nanos case the deviation of the turnaround time between the instances in each workload as well as the total execution time of all the workloads is also smaller than in the PGI case. This second experiment shows the performance advantages and the stability of our implementation when it is used in a heavily multiprogrammed environment, where several applications compete for the system resources.

6 Releated Work and Conclusions

As far as we know, there is only one related work in this subject, and none for the Windows operating system. In [1], the authors compare the performance of a runtime system and a modified Linux kernel with that of OmniMP, a research OpenMP compiler. They achieve similar results to us, since the OmniMP compiler is totally unaware about multiprogramming. Our work differs in that we use an independent resource manager, implemented as a kernel loadable module, to play the role of the operating system scheduler, instead of modifying the kernel. Moreover, our approach is more flexible without losing the performance advantages and more portable since it can be easily installed in new systems, by simply loading our module.

As we have stated, several OpenMP implementations does not efficiently support application execution on a multiprogrammed environment. In these implementation, the OpenMP feature for dynamic parallelism has been ignored or not efficiently implemented. Our results confirm this statement, since our implementation performs better that the commercial PGI implementation on a multiprogrammed environment. Furthermore, our experiments shows that our implementation achieves better performance even when it is used on a dedicated system. The immaturity of the OpenMP implementations is also shown from the fact that 3 of the NAS Benchmarks fail to scale with more that 2 processors, when compiled with the PGI compiler and run on a dedicated system. The PGI Workstation compiler that we use in this work, does not support this feature of the OpenMP standard. But, neither the OpenMP version of the NAS Parallel Benchmarks that we use in this work were written to use this feature. So our results would not have change if we were using an application or a compiler that support the dynamic parallelism feature. Our approach is completely transparent for the programmer of the original OpenMP-coded application, and does not depend on the OpenMP standard feature for dynamic parallelism. In the future, we plan to use this feature in our implementation as a way to give the resource manager a hint, toward a more efficient processor scheduling.

7 Acknowledgements

We would like to thank our colleagues Christos Antonopoulos and Ioannis Venetis for their valuable help in the compilation of the NAS Parallel Benchmarks. This work was partially supported by G.S.R.T. research program 99EΔ-566.

References

[1] C. Antonopoulos, I. Venetis, D. Nikolopoulos and T. Papatheodorou, *Efficient Dynamic Parallelism with OpenMP on Linux SMPs.* In Proc. of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, Nevada, June, 2000.

[2] E. Ayguadé et al., *NanosCompiler: A Research Platform for OpenMP Extensions.* In Proc. of the First European Workshop on OpenMP, pp. 27–31, Lund, Sweden, September, 1999.

[3] V. Barekas, P. Hadjidoukas, E. Polychronopoulos and T. Papatheodorou, *Nanothreads vs. Fibers for the Support of Fine Grain Parallelism in Windows NT/2000 Platforms.* In Proc. of the Third International Symposium on High Performance Computing, pp. 146–159, Tokyo, Japan, October, 2000.

[4] P. Hadjidoukas and V. Barekas and E. Polychronopoulos and T. Papatheodorou, *A Portable Kernel-Mode Resource Manager on Windows 2000 Platforms.* In Proc. of the Twelth IASTED International Conference Parallel and Distributed Computing and Systems, Las Vegas, Nevada, November, 2000.

[5] H. Jin, M. Frumkin and J. Yan, *The OpenMP Implementation of NAS Parallel Benchmarks and its Performance.* Technical Report NAS-99-011, NASA Ames Research Center, October, 1999.

[6] Intel Corporation, Intel Fortran Compiler. Available at: `http://developer.intel.com`.

[7] X. Martorell, J. Labarta, N. Navarro and E. Ayguadé, *A Library Implementation of the Nano-Threads Programming Model.* In Proc. of the 2nd Euro-Par Conference, Lyon, pp. 644–649, August, 1996.

[8] X. Martorell, et al. *Thread Fork/Join Techniques for Multi-level Parallelism Exploitation in NUMA Multiprocessors.* In Proc. of the 13th ACM International Conference on Supercomputing, Rhodes, June, 1999.

[9] OpenMP Architecture Review Board, *OpenMP Specifications.* Available at: `http://www.openmp.org`.

[10] E. Polychronopoulos, et al.*An Efficient Kernel-Level Scheduling Methodology for Multiprogrammed Shared Memory Multiprocessors.* In Proc. of the 12th International Conference on Parallel and Distributed Computing Systems, Fort Lauderdale, Florida, August, 1999.

[11] Portland Group Inc, *Web site.* `http://www.pgroup.com`.